

Let the Shuffle Fly

Zhouwang Fu¹ and Zhengwei Qi¹

¹Shanghai Jiao Tong University

Abstract

In large-scale data-parallel analytics, *shuffle*, or the cross-network read and aggregation of partitioned data between tasks with data dependencies, usually brings in large network transfer overhead. Due to the dependency constraints, execution of those descendant tasks could be delayed by logy shuffles. To reduce shuffle overhead, we present *SCache*, a plugin system that particularly focuses on shuffle optimization in frameworks defining jobs as *directed acyclic graphs* (DAGs). By extracting and analyzing the DAGs and shuffle dependencies prior to the actual task execution, *SCache* can take full advantage of the system memory to accelerate the shuffle process. Meanwhile, it adopts heuristic-MinHeap scheduling combining with shuffle size prediction to pre-fetch shuffle data and balance the total size of data that will be processed by each descendant task on each node. We have implemented *SCache* and customized Spark to use it as the external shuffle service and co-scheduler. The performance of *SCache* is evaluated with both simulations and testbed experiments on a 50-node Amazon EC2 cluster. Those evaluations have demonstrated that, by incorporating *SCache*, the shuffle overhead of Spark can be reduced by nearly 89%.

1 Introduction

Recent years have witnessed widespread use of sophisticated frameworks such as Dryad[25], Spark[37] and Apache Tez[31]. Tremendous efforts have been paid to improve the speed and of large-scale dataparallel systems computing process, from the the low level storage system to scheduling algorithms.

DAG computing frameworks deriving from MapReduce[18] contains a hard barrier between computing stages. The terminology of this barrier is *shuffle*. Shuffle contains two parts on the connecting stages – *shuffle write* and *shuffle read*. On the side of ancestor stages, *shuffle write* is responsible for writing intermediate results to disk. On the side of descendant stage, *shuffle read* fetches intermediate results from remote disks through network. Although highly optimized

in other factors, the shuffle of framework is still primitive. The coarse design of shuffle introduce a significant performance overhead. For instance, a MapReduce trace analysis from Facebook shows that shuffle accounts for 33% JCT on average, up to 70% in shuffle-heavy jobs[16].

The main defect of current shuffle design is coarse granularity of resource allocation during the task scheduling. Nearly all task scheduling algorithms in DAG frameworks use time slotted model. Specifically, when a task is launched, the framework offers it a bundle of resources (i.e. CPU and memory), which are dedicated to this task during the time in its "slot". But for a task, the resources demand changes during different phases. The computing phase is CPU and memory intensive. The shuffle, instead, is I/O intensive. As shown in the upper part of Figure 1, this "slot" can be released until the map tasks finish *shuffle write* on disk. And the "slot" is occupied when the reduce tasks begin to read shuffle data from remote nodes through network, which is presented as *shuffle read*. This inconsistency between demands and allocation results in a severe resource underutilization, which slow down the framework.

Another drawback of current shuffle is the synchronized shuffle read. When all the reduce tasks are scheduled, the shuffle fetch of each task starts almost simultaneously, which may cause congestion of network and delay the shuffle read. The straight forward way to avoid network burst is to start reduce tasks earlier. Apache Hadoop[2] provides a mechanism that schedules reduce tasks when a certain portion of map tasks completed. So that the shuffle delay can be mitigated. Other publications also purpose solutions to pre-schedule reduce tasks[19, 15, 32]. However this early scheduling of reduce tasks occupies new task slots, which degrades system performance. To this end, we proposed a question for this cross-frameworks issue, *can we efficiently optimize shuffle without manually change every DAG framework?*

In this paper, we introduce S(huffle)Cache, an plugin system to remove shuffle latency for DAG frameworks. *SCache* takes over the management of shuffle and I/O resources to achieve a fine granularity scheduling of tasks. In addition, *SCache* pre-schedules the reduce tasks without launching them and perform shuffle data pre-fetch to

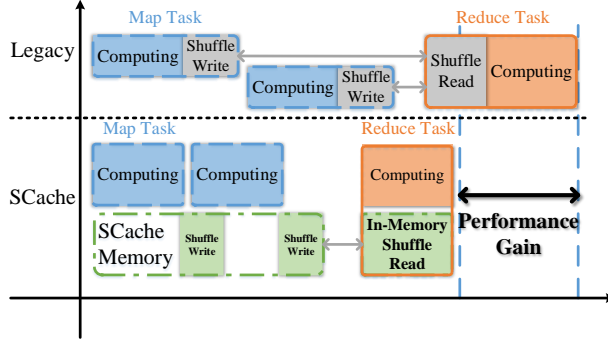


Figure 1: Workflow Comparison between Legacy DAG Computing Frameworks and Frameworks with SCache

break the synchronization of shuffle fetch. In order to provide a general optimization for different DAG frameworks, SCache decouple the shuffle process from computing and provide a cross-frameworks API for shuffle write and read.

The workflow of DAG framework with SCache is presented in Figure 1. In Figure 1, SCache hijacks the intermediate data of a map task from memory space of the slot. The disk operation is skipped and the slot is released after the memory copy. The in-memory intermediate data is immediately shuffled through network to the remote node after heuristic pre-scheduling. By releasing the slot earlier and taking over the I/O operation to start the network transfer ahead of reduce tasks, SCache can help the DAG framework achieve a significant performance gain. A by-product optimization of heuristic pre-scheduling is that SCache can provide a more balanced load for each node. It can further benefit the reduce stage by avoiding data skew.

The main challenge to achieve this optimization is *pre-scheduling reduce tasks*. This challenge is not critical for the simple DAG computing such as Hadoop MapReduce[18]. Unfortunately the complexity of DAG can amplify the defects of naïve pre-scheduling schemes. In particular, randomly assign reduce tasks to nodes can result in a collision of two heavy tasks on one node. This collision can aggravate data skew and hurts the performance of the DAG frameworks. To address this challenge, we propose a heuristic scheme to predict the shuffle output distribution and schedule reduce tasks.

The second challenge is the *limitation of memory space*. To prevent shuffle data touching the disk, SCache leverages extra memory to store the shuffle data. However, the memory is a precious resource for DAG computing, especially for in-memory framework such as Spark[37]. In order to optimize shuffle without hurting the performance of DAG frameworks, SCache only reserves small fraction of memory to store shuffle data. To maximum the performance gain of optimization and memory

utilization, we propose two constraints: all-or-nothing and context-aware. The memory management scheme follows these two constraints to switch shuffle data blocks on and off reserved memory.

We have implemented SCache and customized Apache Spark[4]. The performance of SCache is evaluated with both simulations and testbed experiments on a 50-node Amazon EC2 cluster. We conduct basic test like Group-ByTest. We also evaluate benchmark like Terasort[8] and standard workloads like TPC-DS[9] for multi-tenant modeling. In a nutshell, SCache can eliminate explicit shuffle process by at most 89% in varied application scenarios.

2 Motivation

In this section, we first study the typical shuffle characteristics (2.1), and then spot the opportunities to achieve shuffle optimization (2.2)

2.1 Characteristic of Shuffle

In large scale data parallel computing, enormous datasets are partitioned into pieces to fit into the memory of each node. Meanwhile, complicated application procedures are divided into steps. The succeeding steps take the output of ancestors as computation input. Shuffle occurs when each successor needs part of data from all ancestors' output. It is designed to achieve an all-to-all data blocks transfer among nodes in the cluster. It exists in both MapReduce models and DAG computation models. For clear illustration, we define those computing on each partition of data in one step as a *task*. Those tasks that generate shuffle outputs are called as *map* tasks, and tasks consuming shuffle outputs are called *reduce* tasks.

Overview of shuffle process. As shown in Figure 2. Shuffle mainly contains two phases itself: *data partition* and *data transfer*. For *data partition*, each map task will partition the result data (key, value pair) after execution ("Execution" block in Figure 2) into several buckets according to the partition function. The total number of buckets equals to the number of tasks in the next step. *Data Transfer* can be further divided into two parts: *shuffle write* and *shuffle read*. *Shuffle write* starts after data partition ("Data Partition" block in Figure 2) of map tasks. During *shuffle write*, all the partitioned shuffle output data will be written into local persistent storage for fault tolerance [18, 37]. *Shuffle read* starts at the beginning of reduce tasks. These tasks might fetch the data that belong to their corresponding partitions from both remote nodes and local storage.

Impact of shuffle process. Shuffle process is I/O intensive, which might can introduce a significant latency to the application. Reports show that, 60% of MapReduce

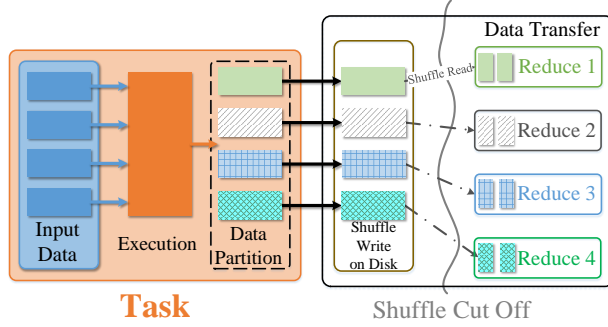


Figure 2: Shuffle Overview

jobs at Yahoo! and 20% at Facebook are shuffle intensive workloads[11]. For those shuffle intensive jobs, the shuffle latency may even dominate Job Completion Time (JCT). For instance, a MapReduce trace analysis from Facebook shows that shuffle accounts for 33% JCT on average, up to 70% in shuffle intensive jobs[16]. 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 Observations

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 typical Spark applications in a 5-node EC2 cluster with `m4.xlarge`. We then measure the CPU utilization, I/O throughput and tasks execution information of each node. Take the trace of one node running Spark *GroupByTest* job in Figure 3 as an example. This job has 2 rounds of tasks for each node. We have marked out the *execution* phase as from the launch time of the first task of this node to the execution finish timestamp of the last one. The *shuffle write* phase is marked from the timestamp of the beginning of the first partitioned data write. The *shuffle read* and *shuffle read and execution* phase is marked from the start of the first reduce launch timestamp.

Figure 3 reveals the performance information of two stages that are connected by shuffle. By analyzing the trace combining with Spark source code[] and reference publications, we propose following observations.

2.2.1 Coarse Granularity Resource Allocation

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 3, the resource of Spark executor will be realised at the ending of *shuffle write*. On the reduce side, though



Figure 3: CPU utilization and I/O throughput of a node during a Spark single shuffle application

in the context of Spark, the reduce task can do computation while fetching data, the transfer of first few blocks may still introduce a explicit I/O delay. On the other hand, shuffle is I/O intensive job. It doesn't involved CPU and application context. Both *shuffle write* and *shuffle read* occupying the slot without using CPU. The current coarse slot — task mapping results in an inconsistency between resource demands and allocation and a low resource utilization. To break this inconsistency, a finer granularity resource allocation scheme must be provided.

2.2.2 Synchronized Shuffle Read

Combining with traces from other nodes, we find that almost all the reduce tasks start *shuffle read* simultaneously (i.e. The rising network utilization during *shuffle read and execution* in Figure 3). The synchronized *shuffle read* requests burst data into network. As shown in Figure 3, the burst data transfer stresses on network bandwidth, which may result in network congestion among the cluster. The bursty demands of network bandwidth might delay the *shuffle read* and hurt the performance of reduce stage. Data reported in previous work[?, 16] also prove that the network transfer can introduce significant overhead in DAG computing.

2.2.3 Multi-round Tasks Execution

Both experience and DAG framework manuals recommend that multi-round execution of each stage will benefit the performance of applications. For example, Hadoop MapReduce Tutorial [3] 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. Spark Configuration also recommends 2-3 tasks per CPU core in the cluster[6]. In Figure 3 we also run two rounds of tasks to process data of about 70GB. As shown in Figure 3, during the map stage, the network is idle (i.e. Network utilization during *execution* and *shuffle write*). Since the shuffle data becomes available as soon as the execution of one map task is finished, if the destination of the shuffle output of each task can be known in priori, the property of multi-round can be leveraged to do *shuffle read* ahead of reduce stage.

2.2.4 Unefficient Persistent Storage Operation

When we look into the detail I/O operation of shuffle, we find that the operation on persistent storage of shuffle is unefficient. There are at least two persistent storage operation for each shuffle data block. At first, Spark will write shuffle data to the persistent storage after map task execution (i.e. *Shuffle Write* in Figure 3). During the *shuffle read*, Spark will then read shuffle data from remote and local persistent storage, which is the second operation. The persistence of shuffle data was designed for fault tolerance. But we believe it's not necessary for today's cluster. Recall that shuffle data only exist in a short time scale. But the mean time to failure (MTTF) for a server is counted in the scale of year [28], which is exponential comparing with the duration of a shuffle. In addition, the capacity of memory and network has been increasing rapidly in recent years. As a result, numbers of memory based distributed storage system have been proposed [?, 28, ?]. On the other hand, the size of shuffle data is relatively small. For example, shuffle size of Spark Terasort [8] is less than 25% of input data. The data reported in [29] also shows that the amount of data shuffled is less than input data, by as much as a factor of 5-10. We argue that removing persistent storage and using memory to achieve shuffle fault tolerance is feasible and efficient.

Based on these observations, it's straightforward to come up with an optimization that uses memory to store the shuffle data and start *shuffle read* ahead of reduce stage to overlap the I/O operations in *multi-round* of DAG computing. To achieve this optimization:

- Shuffle should be taken over to provide a fine granularity scheduling scheme.
- Reduce tasks should be pre-scheduled without launching to achieve shuffle data pre-fetch.
- Shuffle process should be decoupled to provide a cross-framework optimization

In the following section, we elaborate the methodologies to achieve three design goals.

3 Achieve Shuffle Optimization

In this section, we present the detail methodologies to achieve three design goals. We choose Spark as the representative of DAG computing framework to implement our optimization.

3.1 Take Over Shuffle

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

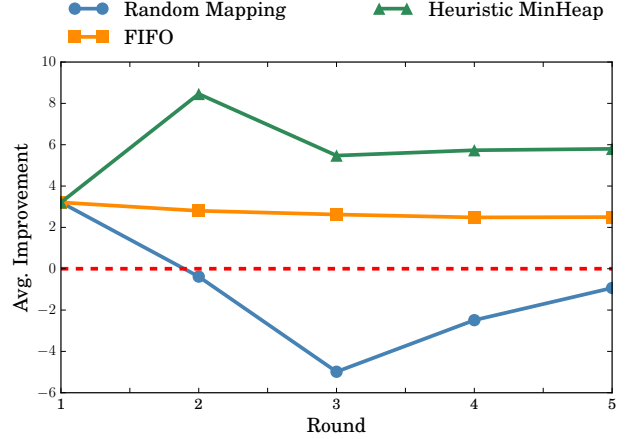


Figure 4: Stage Completion Time Improvement of Open-Cloud Trace

pairs as input. And then it calculates the partitioner number of a key-value pair by applying pre-defined the partition function to the key. At last it put the key-value pair into the corresponding partition. 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 [5], which is big enough in most scenarios according to Section ??.

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 scenarios, and evaluate the performance of pre-scheduling and prediction by calculating the improvement of reduce tasks completion time with trace of OpenCloud [7]. We first emulate the scheduling 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 shuffle data pre-fetch and emulate under different schemes. The result is shown in 4.

Note that since most of the traces from OpenCloud is shuffle-light workload as shown in Figure 5. The aver-

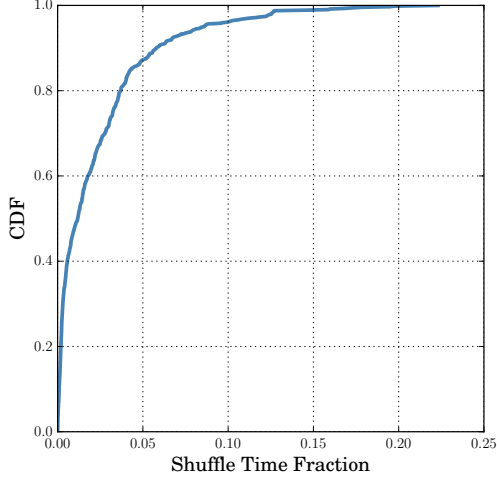


Figure 5: Shuffle Time Fraction CDF of OpenCloud Trace

age shuffle read time is 2.3% of total reduce completion time. So we will only use this trace to evaluate the pre-scheduling.

3.2.1 Random Task-Node Mapping

The simplest way of pre-scheduling is mapping tasks to different nodes evenly. As shown in Figure 4, Random mapping works well when there is only one round of tasks in cluster. But multi-round in cluster is overwhelming according to Section 2.2.3. 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[27, 13, 21] also claim that data-skew is commonly exist in data-parallel computing. When we apply a random mapping, it's probable to assign several 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. In addition, randomly assigned tasks also ignore the data locality between shuffle map output and shuffle reduce input, which can bring extra network traffic in cluster.

3.2.2 Shuffle Output Prediction

The failure of random mapping was obvious caused by application context (e.g. Shuffle input of each task) unawareness, which results in a severe data skew. To avoid the 'bad' schedule results, we have to leverage the application context as assistance. The optimal schedule decision can be made under the awareness of shuffle dependencies number with input size for each task. Unfortunately these data is unavailable when the shuffle-prefetch starts. But the approximate size of each reduce task can be predicted during the initial phase of map tasks with DAG

context, so that the scheduling can approaching a more uniform load for each node.

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 6. '1-1' means it's produced by 'Map Task 1' for 'Reduce Task 1'. For Hadoop MapReduce, the shuffle input for each reduce task can be predicted with decent accuracy[15]. They propose a liner regression model based on Verma et al.[33] 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 6) are picked as observation objects to train the model and than predict the final reduce distribution. But in the more sophisticated DAG computing framework 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 customized partitioner can bring huge inconsistency between observed map output blocks distribution and the final reduce distribution, as we presented in Figure 7. 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 7a, we use a random input dataset with the Hash Partitioner of Spark[5]. In Figure 7b, we use a skew dataset with the Range Partitioner of Spark[5]. 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 distribution 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 choosing k samples from n items, where n is either a very large or unknown number[34]. 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 number of reduce tasks and s is a tunable number. The number of input data partition and reduce tasks can be easily obtained when the from the DAG information. In Figure 7b, we set $s = 3$. After that, the map function is called locally to process the sampled data. As the 'Sampling' part shown in Figure 6, the final sampling map outputs are collected with the size of each partition of input data which is used as weight for each set of sample. For each reduce, the predicted size $reduceSize_i$

$$reduceSize_i = \sum_{j=0}^m partitionSize_j \times \frac{sample_i}{s \times p} \quad (1)$$

(m = partition number of input data)

As we can see in Figure 7b, the prediction result is much better even in a very skew scenario. The variance of the normalized data of sampling prediction is because the standard deviation of the prediction result is relatively small comparing to the average prediction size, which is 0.0015 in this example. Figure 7c further prove that the sampling prediction can provide precise results even in the absolute partition size of reduce tasks. On the opposite, the result of linear regression comes out with huge relative error comparing with the fact of partition size of reduce tasks.

Algorithm 1 Heuristic MinHeap Scheduling for Single Shuffle

```

1: procedure SCHEDULE( $m, h, p\_reduces$ )
2:    $R \leftarrow$  sort  $p\_reduces$  by size
3:    $M \leftarrow$  mapping of host id in  $h$  to reduce id and size
4:    $rid \leftarrow \text{len}(R)$   $\triangleright$  Current scheduled reduce id
5:   while  $rid \geq 0$  do  $\triangleright$  Schedule reduces by MinHeap
6:     Update  $M[0].size$ 
7:     Assign  $R(rid)$  to  $M[0]$ 
8:     sift_down( $M[0]$ )
9:      $\triangleright$  Use min-heap according to size in  $M$ 
10:     $rid \leftarrow rid - 1$ 
11:   $max \leftarrow$  maximum size in  $M$ 
12:   $rid \leftarrow \text{len}(R)$ 
13:  while  $rid \geq 0$  do  $\triangleright$  Heuristic swap by locality
14:     $prob \leftarrow$  max composition portion of  $rid$ 
15:     $nor \leftarrow (prob - 1/m) / (1 - 1/m) / 10$ 
16:     $\triangleright$  Use  $nor$  to limit the performance degradation in
    tasks swap
17:     $t\_h \leftarrow$  host that produces  $prob$  data of  $rid$ 
18:     $c\_h \leftarrow$  current assigned host by MinHeap
19:    if  $t\_h == c\_h$  then
20:      Seal the assignment of  $rid$  in  $M$ 
21:    else
22:      swap_tasks( $rid, c\_h, t\_h, max, nor$ )
23:     $rid \leftarrow rid - 1$ 
24:  return  $M$ 
25: procedure SWAP_TASKS( $rid, c\_h, t\_h, max, nor$ )
26:   $num \leftarrow$  number of reduces
27:  selected from  $t\_h$  that  $total\_size$  won't
28:  make both  $c\_h$  and  $t\_h$  exceed  $(1 + nor) * max$ 
29:  after swapping
30:  if  $num == 0$  then
31:    return
32:  else
33:    # Swap  $num$ s of reduces with  $rid$  between  $c\_h$  and
     $t\_h$ 
34:    # Update size of  $t\_h$  and  $c\_h$ 

```

But the sampling prediction may introduce a extra over-

head in DAG computing process, we will evaluate the overhead in the Section 5. Though in most cases, the overhead is negligible, but we won't use sampling for every reduce prediction. Combing with the DAG context, the sampling prediction will be triggered only when the range partitioner or customized partitioner occurs.

3.2.3 Heuristic MinHeap Scheduling of Single Shuffle

For each predicted reduce size, a percentage array of total data composition among each map output is calculated. The highest percentage and it's corresponding host should be the best choice the dimension of locality. In order to achieve the uniform load on each node while reducing the network traffic and shuffle transmission time. With this composition array and the predicted size of reduce, we present a heuristic MinHeap as the scheduling algorithm for single shuffle.

This algorithm can be divided into two round of scheduling. For input of *schedule*, m is the partition number of input data, h is the array of nodes ID in cluster and $p_reduces$ is the predicted reduce matrix. Each row in $p_reduces$ contains rid as reduce partition ID, $size$ as predicted size of this partition, $prob$ as the maximum composition portion of reduce data, and $host$ as the node ID that produce the maximum portion of reduce data. As for M , it's a matrix consists $hostid$, $size$ (total size of reduce data on this node) and an array of reduce id.

In the first round (i.e. The first while in Algorithm 1), the reduces are first sorted by size. And then, they are assigned to hosts in the descending order of size. For hosts, we use a min-heap to maintain the hosts array according to the scheduled size on each hosts. In other word, the heavy tasks can be distributed evenly in the cluster. After the scheduling, the completion time of reduce stage is close to the optimal. **may need to add math prove between this and optimal.** In the second round, the task-host mapping will be adjusted according to the locality. The closer $prob$ is to $1/m$, the more evenly this reduce is distributed in cluster. For a task which contains at most $prob$ data from $host$, the normalized probability nor is calculated as a bound of performance degradation. This normalization can ensure that the more performance can be traded when the locality level increases. But the degradation of performance will not exceed 10% since the maximum value of nor is 0.1 (in extreme skew scenarios). If the assigned host(c_h in algorithm 1) is not equal to the $host$ (t_h in algorithm 1), than it has the nor probability to trigger a tasks swap between two hosts. To maintain the scheduling mapping of first round, the tasks will only be swapped if the target hosts owns a set of tasks that has similar size totally. We use the OpenCloud[7] trace to evaluate Heuristic MinHeap. Without swapping, the Heuristic MinHeap can achieve a

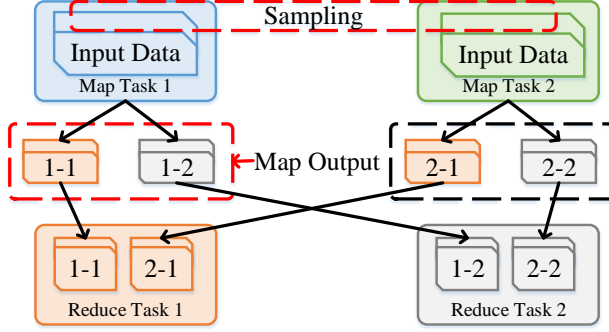


Figure 6: Shuffle Data Prediction

better performance improvement (average 5.7%) than the default Spark FIFO scheduling algorithm (average 2.7%). In the case of extreme skew scenario, such as Figure 7b, Heuristic MinHeap trades about 0.05% percent of stage completion time for 99% reduction of shuffle data transmission through network by heuristically swapping tasks.

3.2.4 Cope with Multiple Shuffles

Unlike Hadoop MapReduce, multiple shuffles commonly exist in DAG computing. The techniques mention in Section 3.2.2 can only predict the ongoing shuffle. For those pending shuffle, it's impossible to predict their size because of lacking either observed map outputs or sampling data. Let all tasks of all shuffle to be scheduled by DAG framework simultaneously can relieve the dilemma. But huge modification in DAG framework should be made by doing this. For example, Spark only supports one stage running at the same time for one application. To avoid this redundant workload, we provide the accumulating scheduling to cope with multiple shuffles.

Algorithm 2 Accumulate Scheduling for Multi-Shuffles

```

1: procedure MSCHEDULE( $m, h, p\_reduces, shuffles$ )
2:    $\triangleright shuffles$  is the previous array of reduce partition
   ID, host ID and size
3:   for all  $r\_id$  in  $p\_reduces$  do
4:      $p\_reduce[r\_id].size \leftarrow p\_reduce[r\_id].size +$ 
        $shuffles[r\_id].size$ 
5:     if  $shuffles[r\_id].size \geq p\_reduce[r\_id].size *$ 
        $p\_reduce[r\_id].prob$  then
6:       Update prob set host to  $shuffles[r\_id].host$ 
7:    $M \leftarrow schedule(m, h, p\_reduces)$ 
8:   for all  $host$  in  $M$  do
9:     for all  $r\_id$  in  $host$  do
10:      if  $host \neq shuffles[r\_id].host$  then
11:        Re-shuffle data to  $host$ 
12:       $shuffles[r\_id].host \leftarrow host$ 
13:   return  $M$ 

```

When a new shuffle start, the $mSchedule$ is called to schedule the new one with previous $shuffles$. The size

of reduce on each node of previous scheduled $shuffles$ are counted. Combined the with the predicted reduces size of the new start shuffle in $p_reduces$, the $size$ of each reduce and its corresponding $prob$ and $host$ are updated. Then the $schedule$ is called to perform the shuffle scheduling. When the new host-reduce mapping is available, for each reduce task, if the new scheduled host in M is not equal to the origin one, the re-shuffle will be triggered to transfer data to new scheduled host for further computing. This re-shuffle can be rare since the previous shuffled data in one reduce contributes a huge composition while doing the accumulate updating. It means in the schedule phase, the $swap-task$ can help revise the scheduling to match the previous mapping in $shuffles$ as much as possible while maintaining the good performance.

4 Implementation

This section overviews the implementation of SCache – a distributed in-memory storage system that caches shuffle data of DAG framework. Here we use Spark as example of DAG framework to illustrate working process of shuffle optimization. We will first present system architecture in Subsection 4.1 while the following two subsections focus on the two constraints on memory management.

4.1 System Architecture

SCache consists mainly two components: A distributed in-memory shuffle data storage system and the daemon inside Spark. As shown in Figure 8, for the in-memory storage system, SCache employs the legacy master-slaves architecture like GFS[20]. The master node of SCache coordinates the shuffle blocks globally with application context from Spark. The coordination provides two guarantees: (a) store data in memory before tasks start and (b) schedule data on-off memory with all-or-nothing property and context-aware-priority constraints to benefit all jobs.

When a Spark job starts, the DAG will be first generated by Spark DAGScheduler[5]. The process starts on the last result stage, and recursively find the dependent stages until the beginning of the DAG. While going forward to the beginning, the DAG computing pipeline will be cut off if a RDD in the stage has one or more shuffle dependencies. These shuffle dependencies among RDDs will then be submitted through RPC call to SCache master by a daemon process in Spark driver. For each shuffle dependency, the shuffle ID (an integer generated by Spark), the type of partitioner, the number of map tasks and the number of reduce tasks are included in the RPC call. The SCache master will store the metadata of one RPC call

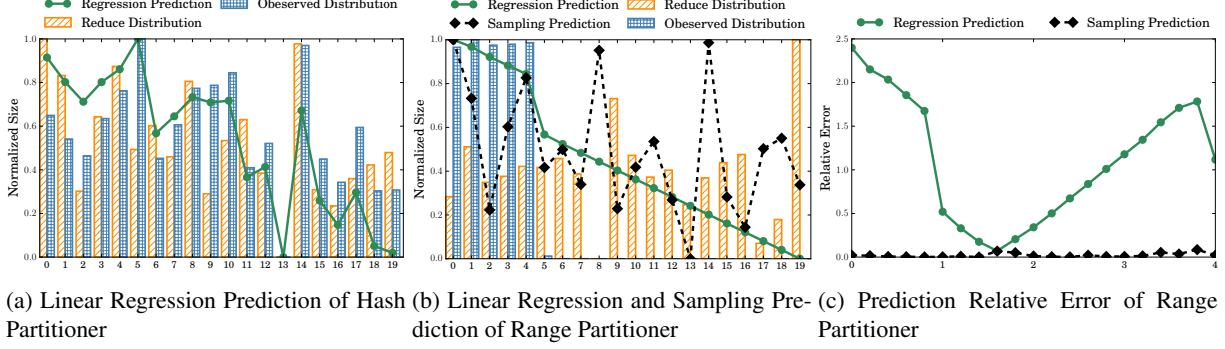


Figure 7: Reduction Distribution Prediction

as a set of multiple shuffles scheduling unit. If there is a specialized partitioner, such as Range Partitioner or a customized partitioner, in the shuffle dependencies, the daemon will insert a sampling program in the host RDD that generates shuffle output using specialized partitioner. The sampling application will be scheduled ahead of that host RDD. We will illustrate the sampling procedure in the Section 4.1.1.

For the hash partitioner, when the map tasks in a stage finish computing on the work nodes, the SCache Worker Daemon process will hijack the shuffle map output in the JVM of each executor of Spark (see Figure 8). Then the data will be transferred into the reserved memory of SCache Worker on each node through memory copy. In the same time, the Spark tasks will end after the memory copy without the disk shuffle output writing, which leads to a reduction of the whole tasks completion time. When the shuffle map output block of a task is stored in the reserved memory, the SCache worker will then notify the master of the block belonging information with the reduce size distribution in this block (see Map Output in Figure 6). When the collected map output data reach the observed ratio of map output, the SCache master will then run the scheduling algorithm 2 (for multiple shuffle dependencies) and 1 (for single shuffle dependency) to get the reduce tasks – nodes mapping. When the scheduling resulted is made, the master will then notify each worker to prepare the memory space for the shuffle data for reduce tasks. The pre-fetch of shuffle data as soon as each worker receives the scheduling results. More specifically, each worker will check the ID of reduce tasks that will be scheduled on itself in the future. When a map task finishes, each node will receive a broadcast message. It will then trigger the pre-fetch process to start fetching shuffle data from the memory of remote SCache worker that just has the map task finished. After all blocks of shuffle map output is transferred, the SCache worker will flush these blocks to disks for saving memory space and maintaining fault tolerance of Spark.

Before the reduce stage starts, Spark DAG Scheduler will first generate a task set for this stage with different locality levels – *PROCESS LOCAL*, *NODE LOCAL*, *NO_PREF*, *RACK LOCAL*, *ANY*. The locality levels are set by finding a cached location of a RDD. For the RDDs that have narrow dependency (opposite of shuffle dependency), the preferred location can also be the same as the dependent RDDs. For those RDDs that have shuffle dependencies, the locality will be set as *NO_PREF* by default. To enforce SCache pre-allocated the reduce tasks, we insert some lines of codes in Spark DAG Scheduler to consult SCache Master to get the preferred node for each tasks. By doing this, the tasks with shuffle dependencies can be set as *NODE LOCAL*. Then the Task Scheduler will schedule tasks according to the task – node mapping from SCache.

When the scheduled reduce tasks start, the shuffle input data is requested. The SCache worker will then pass the requested data through memory copy from the reserved memory to Spark executor JVM memory. As soon as the memory copy finishes, the data in reserved memory will then be flushed to the disk.

4.1.1 Reservoir Sampling

If the submitted shuffle dependencies contain a Range Partitioner or a customized partitioner, the SCache master will send a sampling request to the daemon process in Spark driver. The daemon process will then submit a job on Spark for the current RDD. This sampling job will use a reservoir sampling algorithm[34] on each partition of RDD since the items size of each partition is unknown before sampling. For the sample number, we set the size equals to $3 \times \text{number of partitions}$ for balancing overhead and accuracy (it can be tuned by configuration). The sampling job will then perform a local shuffle with the selected items and partitioner (see Figure 9). At the same time, the size of items is counted as the weight of each partition. These sampling data will be aggregated by *reduce ID* on SCache master. The size for each reduce

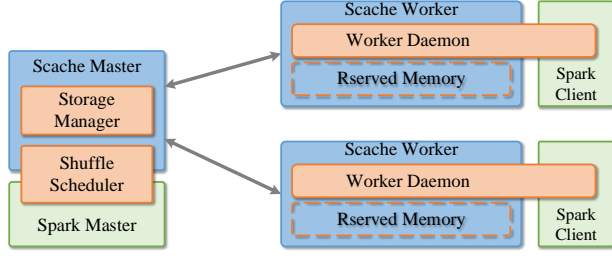


Figure 8: SCache Architecture

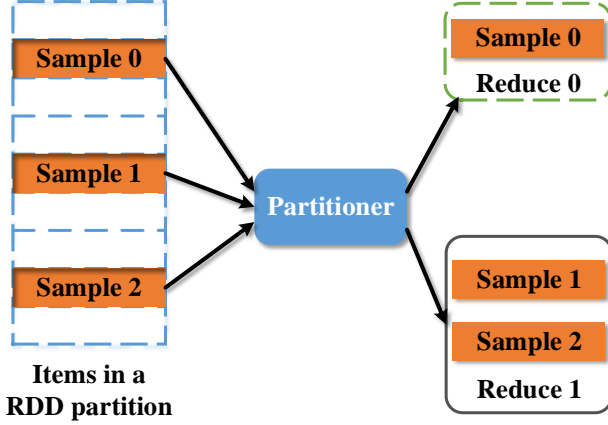


Figure 9: Reservoir Sampling of One Partition

partition can be easily computed by equation 1. After the prediction, master will call algorithm 2 and 1 to do the scheduling.

4.2 Memory Management

As mentioned in section ??, the shuffle size is small enough that can be easily fit in memory. In order to minimize the reserved memory of SCache worker on each node or configure the wrong size by user, the probability of memory exceeded is still exist, especially for multiple existing applications scenarios. When the cached data meet the limitation of reserved memory, SCache flushes some of them to the disk temporarily. And re-fetch them as soon as some cached shuffle blocks is consumed by tasks. To achieve maximum improvement in overall, SCache leverages two constraints to manage the in-memory data – all-or-nothing and context-aware-priority property.

4.2.1 All-or-Nothing Property

Achieving memory cached shuffle data for a reduce task will shorten the task execution time. But this acceleration of single task doesn't speed up the whole stage. Base on the observation that in most cases one single stage contains multi-rounds of tasks from section 2.2.3, the shuffle

cache should at least benefit all tasks in one round. If one of the task misses a memory cache and exceeds the original bottleneck of this round, that task will become the new bottleneck of that round and can further slow down the whole stage. PACMan[12] has proved that for multi-round stage/job, the completion times improves in steps when $n \times \text{number of tasks in one round}$ tasks have data cached in local memory. Therefore, the memory cache of shuffle data need to match at least every round of tasks in a stage. We refer to this as the all-or-nothing property.

According to all-or-nothing property, SCache master leverages the scheduled results to determine the bound of each round of tasks, and then use this as the minimum unit of storage to manage the reserved memory globally. That is, there is one or more storage units for a shuffle schedule unit **add figure**. For those incomplete unit, SCache will mark them as the lowest priority. Following the all-or-nothing constraint can maximum the improvement in stage completion time by using reserved memory efficiently.

4.2.2 Context-Aware-Priority Property

When the size of cached shuffle data exceeds the reserved memory, SCache should decide which of these should be flushed to disk according to the priorities of each storage unit. SCache master first searches if there is an incomplete unit and flush all blocks belonging to the unit to disk cluster-widely.

But what if all the units are completed in the cluster? Traditional cache replacement schemes, such as MIN[14], that only maximize cache hit ratio do not consider the application context in DAG computing thus will easily violate all-or-nothing constrain. In addition, since the cached shuffle blocks will be only read exactly once (without failure), the hit ratio is actually meaningless in this scenario. To decide the priorities among units, SCache makes decision in two dimensions – *inter shuffle units* and *intra shuffle unit*.

- **Inter shuffle units:** SCache master follows the scheduling scheme of the task scheduling schemes of Spark. For a FAIR scheduler, Spark will try to balance the resource of among task sets, which leads to a higher scheduled probability for those has more remaining tasks. The more remaining tasks a stage has, the more storage units exist in the corresponding shuffle unit. Based on this observation, SCache sets priorities from high to low to each shuffle units in descending order of storage units. For a FIFO scheduler, Spark will schedule the task set that is submitted first. So SCache can set the priorities according to the submit time of each shuffle unit and evict the recent ones.
- **Intra shuffle unit:** When a shuffle unit has been marked as the lowest priority, SCache should decide

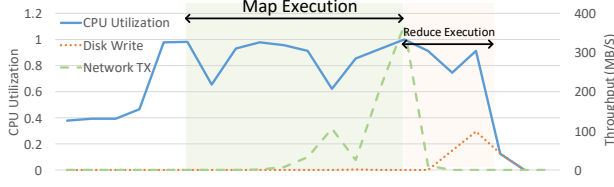


Figure 10: CPU utilization and I/O throughput of a node during a Spark single shuffle application With SCache

the order of eviction among storage units in it. Refer to the task scheduling inside a task set of Spark, the tasks with smaller ID will be scheduled at first under one locality level. Recall that SCache sets all reduce tasks to **NODE LOCAL**, it can easily select the storage unit with larger tasks ID and flush them to disk.

5 Evaluation

In this section, we present the evaluation result of Spark with SCache comparing with the original Spark. We first run simple DAG computing jobs with two stages to analyze the impact of shuffle pre-fetch on the scope of the hardware resources differences of the Spark cluster. The shuffle dependency between two stages contains one shuffle. In addition, we run a shuffle heavy benchmark named Spark Terasort[8] to evaluate the improvement of SCache for each stage. In order to prove the performance gain of SCache with a real production workload, we also evaluate Spark TPC-DS[10] and present the overall performance comparison. At last, we present the overhead of sampling. Because a complex Spark application consists of multiple stages. The completion time of each stage varies under different input data, configurations and different number of stages. This uncertainty leads to the dilemma that dramatic fluctuation in overall performance comparing. To present a straightforward illustration, we limit the scope of most evaluations in single stages.

5.1 Setup

We run our experiments on a 50 m4.xlarge nodes cluster on Amazon EC2[1]. Each node has 16GB memory and 4 CPUs. The network bandwidth is not specifically provided by Amazon. Our evaluation reveals the bandwidth is about 300 Mb/s (see Figure 3).

5.2 Simple DAG Analysis

5.2.1 Differential Runtime Hardware Utilization

We first run the same single shuffle test (GroupByTest from Spark example[5]) as we mentioned in Figure 3. As

shown in Figure 10, the hardware utilization is captured from one node during the job. Note that since the completion time of whole job is about 50% less than Spark without SCache, the duration of Figure 10 is cut in half as well. A overlap between CPU, disk and network can be easily observed in Figure 10. That is, the I/O operation will never cut off the computing process. By running Spark with SCache, the overall CPU utilization of the cluster stays in a high level. The decoupling of shuffle write from map tasks frees the CPU earlier, which leads to a faster map task computation. The shuffle pre-fetch starts the shuffle data transfer in the early stage of map phase shift the network transfer completion time, so that the computation of reduce can start immediately after scheduled. And this is the main performance gain we achieved on the scope of hardware utilization by SCache.

5.2.2 Performance Gain in Detail

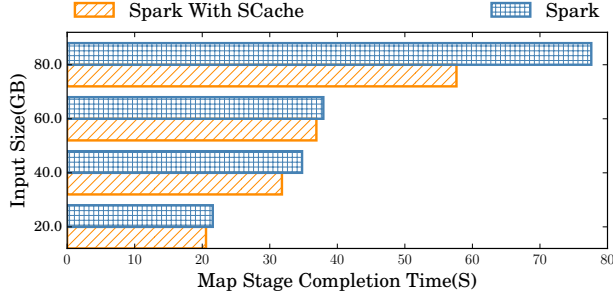
As shown in Figure 11, we run the single shuffle test with different input size in the cluster. For each stage, we run 10 rounds of tasks. The stage completion time is presented separately in Figure 11a and Figure 11b. By running spark with SCache, the completion time of map stage can be reduce 10% on average. For reduce stage, instead, SCache achieves a 75% performance gain in the completion time of whole reduce stage.

Combining with Figure 12, we present a detail analysis into the nutshell of varied overall performance gain on different stages. For each stage, we pick the median task and present in Figure 12. For a single map task, about 40% of shuffle write time can be eliminated by SCache (Figure 12a). Because the serialization of data is CPU intensive[29] and it is inevitable while moving data out of Java heap memory, SCache cannot eliminate the whole phase of shuffle write. This results in a less performance gain in the map stage completion time. On the reduce side, instead, the network transfer contribute significantly latency in shuffle read for a single task (Figure 12b). By doing shuffle data pre-fetch for the reduce tasks in Figure 12b, the shuffle read time decreases 100%, which means shuffle data pre-fetch almost hide all the explicit network transfer in the reduce stage. In addition, heuristic reduce tasks scheduling achieves better load balance in cluster than the Spark default FIFO scheduling which may randomly assign two heavy tasks on a single node. So that we can have a significant performance gain in the completion time of reduce stage.

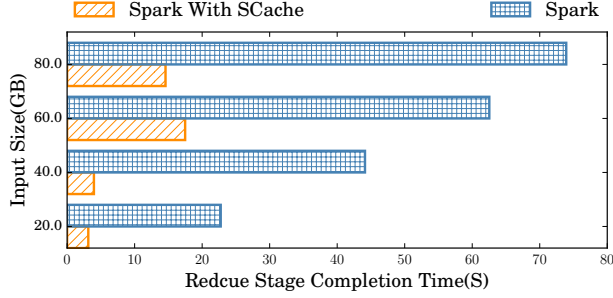
two side of map and reduce, SCache can help Spark decrease about 89% time in the whole shuffle process.

5.3 Terasort

In this part, we evaluate the both TeraSort[8].

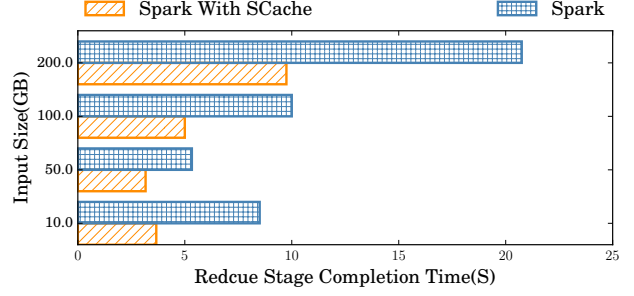


(a) Map Stage Completion Time Comparison

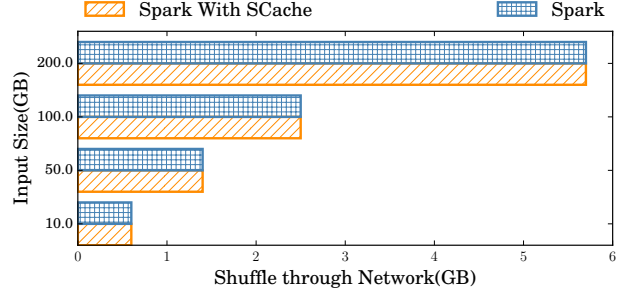


(b) Reduce Stage Completion Time Comparison

Figure 11: Stage Completion Time Comparison of Single Shuffle Test

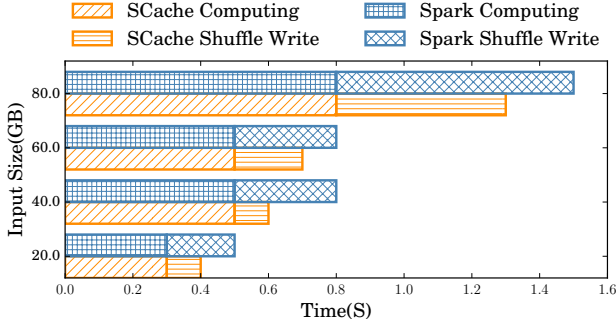


(a) Reduce Stage Completion Time Comparison

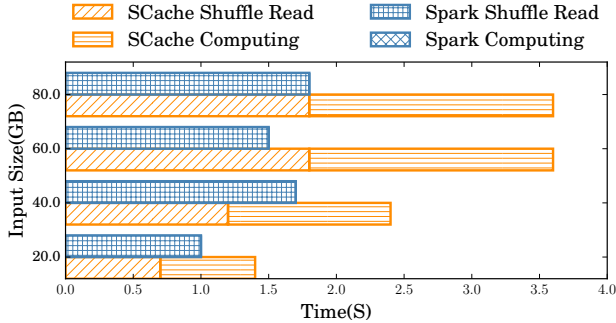


(b) Shuffle Data Size Comparison

Figure 13: Terasort Evaluation



(a) Map Stage Completion Time Comparison



(b) Task Details in Map and Reduce Stages

Figure 12: Stage Completion Time Comparison of Single Shuffle Test

TeraSort[8] is a shuffle intensive benchmark for distributed system analytics. It consists of two consecutive shuffles. The first shuffle reads the input data and uses a custom hash partition function for re-partitioning. The second shuffle partitions the data through a range partitioner and sorts the data respectively. As the range bounds set by the range partitioner almost match the same pattern of the first shuffle, that is, for one reduce task, almost 93% of input data is from one particular map task. It makes the shuffle data transferred through the network extremely small under Spark's locality-preferred task scheduling. So we take the second shuffle as an extreme case to evaluate the scheduling locality for SCache.

As shown in Figure 13a, we present the first shuffle as the evaluation of shuffle optimization. At the same time, we use the second shuffle to evaluate in the dimension of scheduling locality (Figure 13b). For the first shuffle, Spark with SCache runs $2 \times$ faster during the reduce stage in a range from 10GB to 200GB input data. At the same time, the heuristic scheduling algorithms of SCache (Algorithm 1 and Algorithm 2) can obtain the exactly same scheduling result of reduce tasks, which decreases the network traffic. In contrast, Spark delays scheduling reduce tasks with the shuffle data locality to achieve this optimal.

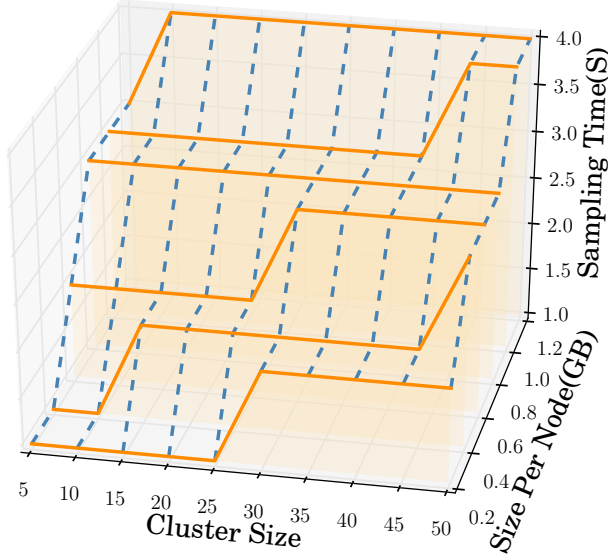


Figure 14: Sampling Overhead

5.4 Production Workload

We also evaluate some shuffle heavy TPC-DS[9] on the cluster. TPC-DS benchmark is designed for modeling multiple users submitting varied queries (e.g. ad-hoc, interactive OLAP, data mining, etc.). TPC-DS contains 99 queries and is considered as the standardized industry benchmark for testing big data systems. We evaluate performance of Spark with SCache by picking some of the TPC-DS queries with shuffle intensive attribute. As shown in Figure 15, on the horizon axis is query number, and on the vertical axis is query completion time. Spark with SCache outperforms the original Spark in almost all the queries in TPC-DS query set. Furthermore, in many-queries, Spark with SCache outperforms original Spark by an order of magnitude. The overall reduction portion of query time that SCache achieved is 40% on average. Since this evaluation presents the overall job completion time of queries, we believe that the optimization is promising.

5.5 Overhead of Sampling

In this part, we evaluate the overhead of sampling with different input data sizes on one node and cluster scales. As shown in 14, the overhead of sampling only grows with the increasing of input size on each node. But it keeps relatively stable when the cluster size scales up. It makes SCache a scalable system in cluster.

6 Related Work

We summarize the shuffle optimization scheduling schemes in this Section. Basically, we categorize related

works in two parts, pre-scheduling and delay scheduling.

Pre-scheduling: Starfish[22] is a self-tuning system in Hadoop. The basic idea is to get sampled data statistics for tuning system parameters (e.g. slowstart, map and reduce slot ratio, etc). However, these parameters cannot be changed once the jobs begin. DynMR[32] dynamically starts reduce tasks in late map stage when there is enough data to be fetched. Thus it reduces the time for reducer to wait for mapper producing outputs. Those two works still left the explicit I/O time wait in both map and reduce phases. iShuffle[15] decouples shuffle from reducers and designs a centralized shuffle controller. The goal is also to find the right time, but it can neither handle multiple shuffles multiple nor schedule multiple rounds of reduce tasks. iHadoop[19] aggressively pre-schedules tasks in multiple successive stages, in order to start fetching data from previous stage earlier. But we have proved that randomly assign tasks may hurt the overall performance in section 3.2.1. Different from these works, SCache pre-schedules reduce tasks without consuming new task slots, whereas all these schemes do.

Delay-scheduling: Delay Scheduling[36] delays tasks assignment to get better data locality, which can reduce the network traffic. ShuffleWatcher[11] delays shuffle fetching when network is saturated. At the same time, it achieves better data locality. Both Quincy[26] and Fair Scheduling[35] can reduce shuffle data by optimizing data locality of map tasks. Even though these kind of schemes can achieve higher data locality, they cannot breach the shuffle cut off between map and reduce stages, whereas SCache does.

6.1 Limitation and Future Work

SCache aims to breach the shuffle cut off between DAG computing stages. And the evaluation results show a promising improvement. But we realize some limitations of SCache.

Fault tolerance of SCache: When a failure happened on the SCache master, the whole system will stop working. To prevent the machine failure leading to inconsistency SCache, the master node will log the meta data of shuffle register and scheduling on the disk. Master can reads logs during recovery. Since we remove the shuffle transfer from the critical path of DAG computing, the disk log will not introduce extra overhead to the DAG frameworks. Note that the master can be implemented with Apache ZooKeeper[24] to provide constantly service to DAG framework. If a failure happens on a worker, the optimization of tasks on that node will fail. It also violates the constraints of all-or-nothing, which means the gain of shuffle data cache maybe negligible. A promising way to solve the failure on worker is to reschedule reduce tasks and retransmit the data. Another solution is

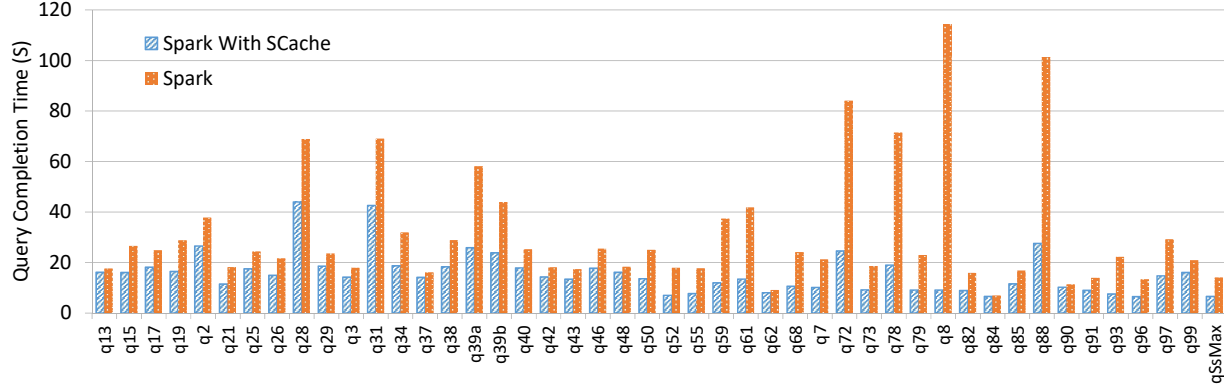


Figure 15: TPC-DS Benchmark Evaluation

selecting some backup nodes to store replications of shuffle data during scheduling to prevent the worker failure. We believe combing the high speed of network and memory is a better choice for fault tolerance. Since the mean time to failure(MTTF) for a server is counted in the scale of year[28]. As for now, fault tolerance is not a crucial goal of SCache, we leave it to the future work.

Scheduling with different frameworks: A cluster for data parallel computing always contains more than one frameworks. Setting priority among jobs submitted from different framework is challenging and complex. However, combining the resource management facilities in data center such as Mesos[23] may be a good direction.

7 Conclusion

In this paper, we present SCache, a shuffle optimization scheme for DAG computing framework. SCache decouples the shuffle from computing pipeline and leverages shuffle data pre-fetch to mitigate I/O overhead of the whole system. By scheduling tasks with application context, SCache bridges the gap among computing stages. Our implementation on Spark and evaluations show that SCache can provide a promising speedup to the DAG framework. We believe that SCache is a simple and efficient plugin system to enhance the performance of most DAG computing frameworks.

References

- [1] Amazon ec2. <https://aws.amazon.com/ec2/>.
- [2] Apache hadoop. <http://hadoop.apache.org/>.
- [3] Apache hadoop tutorial. [http://hadoop.apache.org/docs/](http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html)
- current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.
- [4] Apache spark. <http://spark.apache.org/>.
- [5] Apache spark 1.6 source. <https://github.com/apache/spark/tree/branch-1.6>.
- [6] Apache spark 1.6.2 configuration. <http://spark.apache.org/docs/1.6.2/configuration.html>.
- [7] Opencloud hadoop cluster trace. <http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html>.
- [8] Spark terasort. <https://github.com/ehiggs/spark-terasort>.
- [9] Tpc benchmark ds (tpc-ds): The benchmark standard for decision support solutions including big data. <http://www.tpc.org/tpcds/>.
- [10] Tpc-ds benchmark on spark. <https://github.com/databricks/spark-sql-perf>.
- [11] 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.
- [12] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.
- [13] 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.
- [14] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.
 - [15] 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.
 - [16] 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.
 - [17] 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.
 - [18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
 - [19] E. Elnikety, T. Elsayed, and H. E. Ramadan. ihadoop: asynchronous iterations for mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 81–90. IEEE, 2011.
 - [20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.
 - [21] 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.
 - [22] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Cidr*, volume 11, pages 261–272, 2011.
 - [23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
 - [24] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9, 2010.
 - [25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.
 - [26] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.
 - [27] 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.
 - [28] 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.
 - [29] 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.
 - [30] O. OMalley. Terabyte sort on apache hadoop. *Yahoo*, available online at: [http://sortbenchmark.org/Yahoo-Hadoop.pdf\(May\)](http://sortbenchmark.org/Yahoo-Hadoop.pdf(May)), pages 1–3, 2008.
 - [31] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.
 - [32] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang. Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling. In *Proceedings of the Ninth European Conference on Computer Systems*, page 2. ACM, 2014.
 - [33] 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.
 - [34] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

- [35] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, and X. Li. Preemptive reduced task scheduling for fair and fast job completion. In *ICAC*, pages 279–289, 2013.
- [36] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmelegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.
- [37] 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.