Let the Shuffle Fly

Zhouwang Fu[1] and Zhengwei Qi[1]
[1]*Shanghai 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 constrains, 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 [26], Spark [38] and Apache Tez [32]. Tremendous effeorts 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 framworks deriving from MapReduce [19] 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 resoponsible 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 significiant 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 [18].

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 [20, 16, 33]. 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 acheive a fine granularity scheduling of tasks. In addition, SCache pre-schedules the reduce tasks without launching them and perform shuffle data pre-fetch to break the synchronization of shuffle fetch. In order to

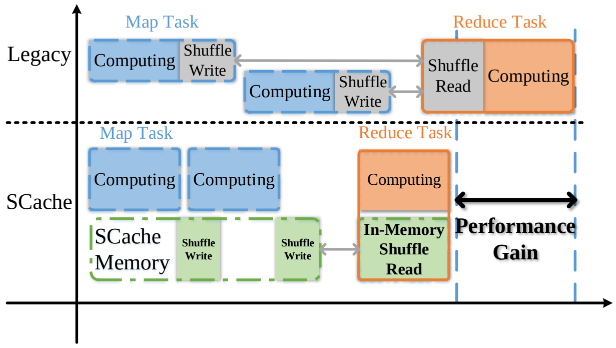


Figure 1: Workflow Comparison between Legacy DAG Computing Frameworks and Frameworks with SCache

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 benifit 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 [19]. 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 [38]. 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 contraints 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 GroupByTest.We also evaluate benchmark like Terasort [9] and standard workloads like TPC-DS [10] for multi-tenant modeling. In a nutshell, SCache can eliminate explict 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 optmization (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 as *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 [19, 38]. *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 jobs at

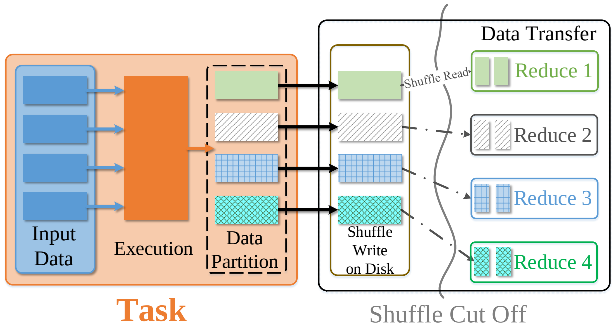


Figure 2: Shuffle Overview

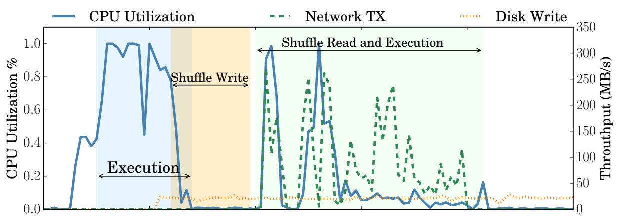


Figure 3: CPU utilization and I/O throughput of a node during a Spark single shuffle application

Yahoo! and 20% at Facebook are shuffle intensive workloads [12]. 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 [18].

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. Here we present the trace of one node running Spark *GroupByTest* 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 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 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 combing with Spark source code [5], we propose the following observations.

2.2.1 Coarse Granularity Resource Allocation

In general, CPU and memory are bounded with a schedule slot in DAG resource scheduler. When a task is scheduled to a slot, it will not release the until the end of the task. In Figure 3, the resource of Spark executor will be released at the ending of *shuffle write*. On the reduce side, though in the context of Spark, the reduce task can do computation while fetching data, the transfer of first few blocks may still introduce an explict I/O delay. On the other hand, shuffle is an I/O intensive job without involving CPU. Both *shuffle write* and *shuffle read* occupy the slot without using CPU. The current coarse slot — task mapping results in an inconsistency between resource demands and slot allocation, which further decrease the 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 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 cause a burst of network traffic. As shown in Figure 3, the data transfer stresses on network bandwidth, which may result in network congestion among the cluster. This bursty demands of network bandwidth might delay the *shuffle read* and hurt the performance of reduce stage. The previous work [17, 18] 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 × no. of nodes × 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 operations of shuffle, we find that the operations on persistent storage of shuffle are unefficient. There are at least two persistent storage operations 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 presistent 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 [29], 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 [7, 29, 30]. On the other hand, the size of shuffle data is relatively small. For example, shuffle size of Spark Terasort [9] is less than 25% of input data. The data reported in [31] also shows that the amount of data shuffled is less than input data, by as much as 10%-20%. We argue that removing persistent storage and using memory to achieve shuffle fault tolerance is feasible and efficient.

Based on these observations, it is 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 framwork 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 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 corresponding partition. The output at last is a set of blocks. Each of them contains the key-value pairs for one partition. At last, they will be flushed to disk. The shuffle takeover starts right here. To prevent the synchronized disk write holding the slot, we use memory copy to hijack shuffle data from Spark executor's JVM space. By doing this, a slot can be released as soon as it finish CPU intensive computing. After that, shuffle data is managed outside the DAG framework. The pre-scheduling can be made to start pre-fetch after enough shuffle data is collected.

On the reduce side, shuffle data is pre-fetched and cached in memory after pre-scheduling. When the reduce tasks start, it can directly read shuffle data from local memory.

To this end, all I/O operations are managed outside the DAG framework, and the slot is occupied only by the CPU intesive phase of task.

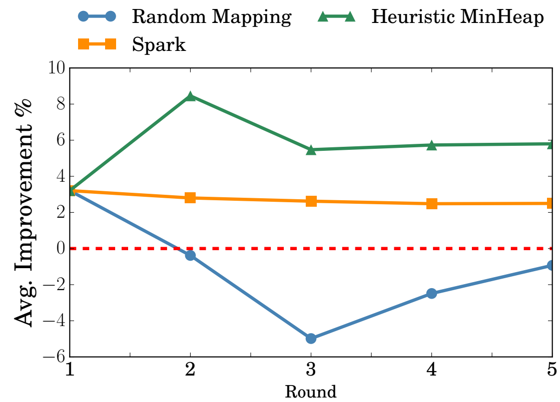


Figure 4: Stage Completion Time Improvement of OpenCloud Trace

3.2 Pre-schedule with Application Context

The main challenge toward the optimization is how to pre-schedule the reduce tasks without launching. The node and tasks mapping is made until they are scheduled by scheduler of DAG framework. But as soon as they are scheduled, slots will be occupied to launch them. On the other hand, shuffle data cannot be pre-fetched without knowing the node and tasks mapping. To get rid of this dilemma, we propose a co-scheduling scheme. That is, the task — node mapping is made ahead of DAG framework scheduler, and then enforce the mapping result to DAG scheduler while doing the real scheduling.

To evaluate the impact of different pre-scheduling schemes, we use trace from OpenCloud [8] for the sim-

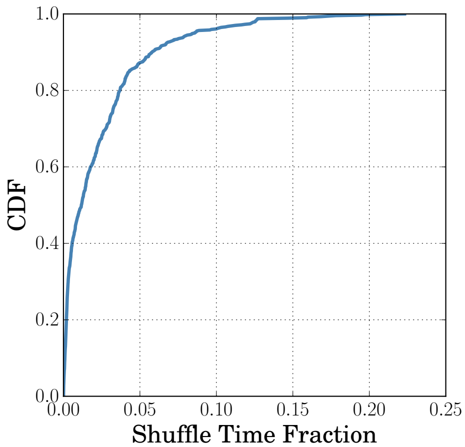


Figure 5: Shuffle Time Fraction CDF of OpenCloud Trace

ulation. The baseline (red dot line in Figure 4) is the stage completion time under Spark default scheduling algorithm. And then we remove the shuffle read time of each task, and do the simulation under three different schemes.

Note that most of the traces from OpenCloud is shuffle-light workload as shown in Figure 5. The average shuffle read time is 2.3% of total reduce completion time.

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. But performance of random mapping collapses as the round number grows. It is because that data-skew is commonly exist in data-parallel computing [28, 14, 22]. Several heavy tasks might be assigned on the same node. This collision than slow down the the whole stage, which make the performance even worse than baseline. In addition, randomly assigned tasks also ignore the data locality between shuffle map output and shuffle reduce input, which might introduce 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 data size) unawareness. To avoid the 'bad' scheduling results, we have to leverage the application context as assistance. The optimal schedule decision can be made under the awareness of shuffle dependencies number, partition number and shuffle size for each partition. The first two of them can be easily extact from DAG information. To acheive a better scheduling result, the shuffle size for each partition should be predicted during the initial phase of map tasks.

According to the DAG computing process, the shuffle size of each reduce task is decided by *input data*, *map task computation* and *hash partitioner*. For each map task, it produces a data block for each reduce task, 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 [16] by liner regression model based on observation that the ratio of map output size (e.g. map output in Figure 6) and input size is invariant given the same job configuration [34].

But the sophisticated DAG computing framework like Spark introduces more uncertainty. For instance, the reduce stage in Spark has more number of tasks than Hadoop MapReduce. More importantly, the customized partitioner can bring huge inconsistency between observed map output blocks distribution and the final reduce input distribution. To find out the connection among three factors, we use different datasets with different partitioners. The result is presented in Figure 7. 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. In Figure 7b, we use a skew dataset with the range partitioner of Spark [5]. The observed map outputs are andomly picked. As we can see, in hash partitioner, the distribution of observed map output is close to the final reduce input distribution(orange boxes). The prediction results also turns out well. However, the huge inconsistency between final reduce distribuion and observed distribution results in a deviation in linear regression model.

To handle this inconsistency, we introduce another methodology named weighted reservoir sampling. The classic reservoir sampling is designed for randomly choosing k samples from n items, where n is either a very large or unknown number [35]. For each partition of map task, 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 (*sampling* in Figure 6). The final sampling outputs are collected with the size of each map partition which is used as weight for each set of sample. For each reduce, the predicted size $reduceSize_i$

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

$$(m = \text{partition number of input data})$$

As we can see in Figure 7b, the result of sampling prediction is much better even in a very skew scenario. The variance of the normalized between sampling prediction and reduce distribution 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 result even in the dimension of absolute shuffle partition size. On the opposite, the result of linear regression comes out with huge relative error.

Algorithm 1 Heuristic MinHeap Scheduling for Single Shuffle

procedure SCHEDULE($m, h, p\_reduces$)
 $R \leftarrow$ sort $p\_reduces$ by size
 $M \leftarrow$ mapping of host id in h to reduce id and size
 $rid \leftarrow$ len(R)   ▷ Current scheduled reduce id
 while $rid \geq 0$ **do**   ▷ Schedule redues by MinHeap
  Update $M\,[0]\,.size$
  Assign $R\,(rid)$ to $M\,[0]$
  sift_down($M\,[0]$)
     ▷ Use min-heap according to size in M
  $rid \leftarrow rid - 1$
 $max \leftarrow$ maximum size in M
 $rid \leftarrow$ len(R)
 while $rid \geq 0$ **do**   ▷ Heuristic swap by locality
  $prob \leftarrow$ max composition portion of rid
  $nor \leftarrow (prob - 1/m)\,/\,(1 - 1/m)\,/10$
    ▷ Use nor to limit the performance degradation in tasks swap
  $t\_h \leftarrow$ host that produces $prob$ data of rid
  $c\_h \leftarrow$ current assigned host by MinHeap
  if $t\_h == c\_h$ **then**
   Seal the assignment of rid in M
  else
   swap_tasks(rid, c_h, t_h, max, nor)
  $rid \leftarrow rid - 1$
 return M
procedure SWAP_TASKS(rid, c_h, t_h, max, nor)
 $num \leftarrow$ number of reduces
 selected from t_h that $total\_size$ won't
 make both c_h and t_h exceed $(1 + nor) * max$
 after swapping
 if $num == 0$ **then**
  return
 else
  # Swap $nums$ of reduces with rid between c_h and t_h
  # Update size of t_h and c_h

However, sampling prediction trade accuracy with extra overhead in DAG computing process. we will evaluate the overhead in the Section 5. Though in most cases, the overhead is acceptable, the sampling prediction will be triggered only when the range partitioner or customized non-hash partitioner occurs.

3.2.3 Heuristic MinHeap Scheduling of Single Shuffle

In order to achieve the uniform load on each node while reducing the network traffic, we present a heuristic Min-

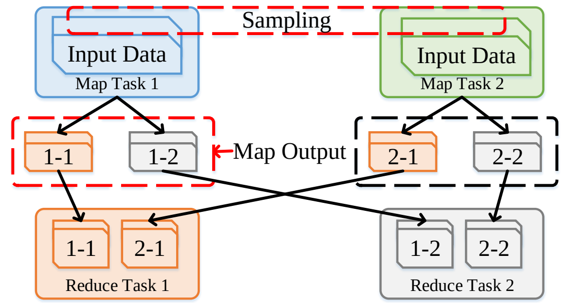
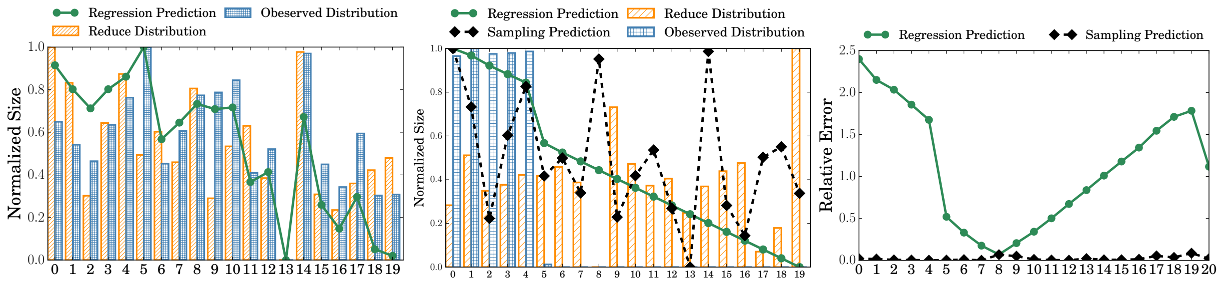


Figure 6: Shuffle Data Prediction

Heap (1) as the scheduling algorithm for single shuffle. It tasks predicted shuffle distribution, locality information and DAG information as input. Unlike the naïve Spark scheduling algorithm, combining these information help the scheduler make a more balanced task — node mapping, which accelerate the reduce stage. This is the by-product optimization harvested from shuffle size prediction.

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 r_id 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.

This algorithm can be divided into two rounds. In the first round (i.e. The first while in Algorithm 1), the reduces are first sorted descendingly by size. For hosts, we use a min-heap to maintain the priority by size of assigned tasks. So that the heavy tasks can be distributed evenly in the cluster. In the second round, the task — node mapping will be adjusted according to the locality. The closer $prob$ is to $1/m$, the more evenly this shuffle partition is produced 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% (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 $swap\_tasks$ will be triggerd. Inside the $swap\_tasks$, tasks will be selected and swapped without exceeding the performance tradeoff threshold ($(1 + nor) * max$). We use the OpenCloud [8] trace to evaluate Heuristic MinHeap. Without swapping, the Heuristic MinHeap can achieve a better performance improvement (average 5.7%) than the default Spark FIFO scheduling algorithm (average 2.7%). The test bed evaluation are presented in Section 5.



(a) Linear Regression Prediction of Hash Partitioner

(b) Linear Regression and Sampling Prediction of Range Partitioner

(c) Prediction Relative Error of Range Partitioner

Figure 7: Reduction Distribution Prediction

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 handle the ongoing shuffle. For those pending shuffle, it's impossible to predict the size. Let all tasks of all shuffle to be scheduled by DAG framework simultaneously can relieve the dilemma. But doing this introduces extream overhead such as redundant extra task serialization. To avoid violating the optimization from framework, we provide the accumulating scheduling to cope with multiple shuffles.

Algorithm 2 Accumulate Scheduling for Multi-Shuffles

procedure MSCHEDULE($m, h, p\_reduces, shuffles$)
  ▷ $shuffles$ is the previous array of reduce partition ID, host ID and size
  for all r_id in $p\_reduces$ **do**
    $p\_reduce\,[r\_id]\,.size \leftarrow p\_reduce\,[r\_id]\,.size + shuffles\,[r\_id]\,.size$
    if $shuffles\,[r\_id]\,.size \geq p\_reduce\,[r\_id]\,.size * p\_reduce\,[r\_id]\,.prob$ **then**
      Update $prob$, set $host$ to $shuffles\,[r\_id]\,.host$
  $M \leftarrow$ schedule($m, h, p\_reducecs$)
  for all $host$ in M **do**
    for all r_id in $host$ **do**
      if $host \neq shuffles\,[r\_id]\,.host$ **then**
        Re-shuffle data to $host$
        $shuffles\,[r\_id]\,.host \leftarrow host$
  return M

The size of reduce on each node of previous scheduled $shuffles$ are counted. When a new shuffle starts, the $mSchedule$ is called to schedule the new one with previous $shuffles$. Combining with the predicted reduces size of the new start shuffle in $p\_reduces$, the $size$ of each reduce and its corresponding $porb$ 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 equle 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. 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 load balance.

4 Implementation

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

4.1 System Overview

SCache consists mainly three components: A distributed in-memory shuffle data storage system, a DAG co-scheduler 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 [21]. 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. The worker node reserves memory to store blocks and bridges the commuication between Spark and SCache. The co-scheduler is dedicated to pre-schedule reduce tasks with DAG information and enforce the scheduling result to Spark.

When a Spark job starts, the DAG will be first generated. Spark DAG scheduler recursively visits the dependencies from last RDD. While going forward to the beginning, the DAG computing pipeline will be cut off if a RDD 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 as a set of shuffles scheduling unit. If there is a specialized partitioner, such as range partitioner, in the shuffle dependencies, the daemon will insert a sampling program in the host RDD. This sampling application will be scheduled ahead of that host RDD. We will elaborate the sampling procedure in the Section 4.1.1.

For the hash partitioner, when a map tasks finishes computing, the SCache daemon process will transfer the shuffle map output from Spark executor to the reserved memory through memory copy. At the same time, the map task will end and the slot will be released after the memory copy without the shuffle map output persistence. When the shuffle map output is received, 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). If the collected map output data reach the observation threshold, the SCache co-scheduler will then run the scheduling algorithm 2 (for multiple shuffle dependencies) and 1 (for single shuffle dependency) to pre-schedule the reduce tasks and then broadcast the scheduling result. The pre-fetching of shuffle data starts as soon as each worker receives the scheduling results. More specifically, each worker will filter the reduce tasks ID that will be launch on itself. When a map task finishes, each node will receive a broadcast message. The pre-fetch process will be triggered to start fetching shuffle data from the remote SCache worker. After the blocks of shuffle map output are transferred, the SCache worker will flush these blocks to disk to memory space and maintain 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*. To enforce SCache prescheduled the tasks — node mapping, we insert some lines of codes in Spark DAG Scheduler. For RDDs with shuffle dependecies, Spark DAG scheduler will consult SCache master to get the preferred node for each partition and set *NODE_LOCAL* locality level on corresponding tasks.

When the scheduled reduce tasks start, the shuffle input data can be fetched from reserved memory in SCache worker through daemon process. As soon as the data is consumed by reduce task, it will be flushed to the disk.

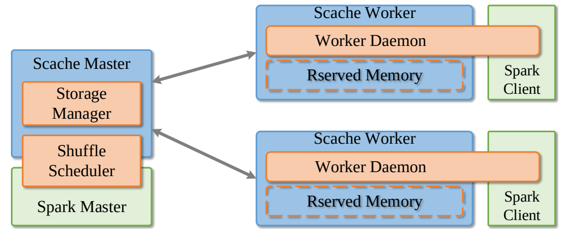


Figure 8: SCache Architecture

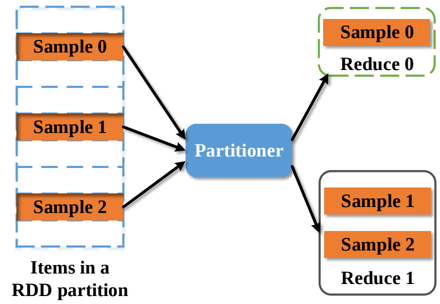


Figure 9: Reservoir Sampling of One Partition

4.1.1 Reservior Sampling

If the submitted shuffle dependencies contain a range partitioner or a customized non-hash partitioner, the SCache master will send a sampling request to daemon in Spark driver. The daemon then insert sampling job with corresponding RDD. This sampling job uses a reservoir sampling algorithm [35] on each partition of RDD. For the sample number, we set the size equals to $3 \times$ $number\ of\ partitions$ for balancing overhead and accuracy (it can be tuned by configuration). The sampling job randomly select some items and performs a local shuffle with partitioner (see Figure 9). At the same time, the items number of this partition is counted as the weight. These sampling data will be aggregated by $reduce\ ID$on SCache master. The prediction of reduce size 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 2.2, despite the shuffle size is small enough to be fitted in memory, unfortunately, the probability of memory exceeded is still exist. 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 or pre-fetched. To achieve maximum overall improvement, SCache leverages two constraints to manage the in-memory data — all-or-nothing and context-aware-priority.

4.2.1 All-or-Nothing Property

Memory cached shuffle can speed up the reduce task execution. But this acceleration of single task is necessary but unsufficient for a shorter stage completion time. Base on the observation that in most cases one single stage contains multi-rounds of tasks from section 2.2.3, if one of the task misses a memory cache and exceeds the original bottleneck of this round, that task will become the new bottleneck and might further slow down the whole stage. PACMan [13] has also proved that for multi-round stage/job, the completion times improves in steps when $n \times number\ of\ tasks\ in\ one\ round$ of tasks have data cached simultaneously. Therefore, the memory cache of shuffle data need to match at least all tasks in a running round. We refer to this as the all-or-noting property.

According to all-or-nothing property, SCache master leverages the pre-scheduled results to determine the bound of each round, and then use this as the minimum unit of storage to manage the reserved memory globally. That is, the storage unit number equals to the number of task round for a shuffle schedule unit. For those incomplete unit, SCache will mark them as the lowest priority.

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 [15], only maximize cache hit ratio without considering the application context in DAG computing. Directly appyling them might easily violate all-or-nothing constraint. In addition, since the cached shuffle blocks are 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 Spark to determine the inter-shuffle priority. For a FAIR scheduler, Spark balances the resource of among task sets, which leads to a higher priority for those with more remaining tasks. More remaining tasks a stage implies more storage units left unconsumed. So SCache sets priorities from high to low to each shuffle units in descending order of remaining storage units. For a FIFO scheduler, Spark schedules the task set that is submitted first. So SCache sets the priorities according to the submit time of each shuffle unit.

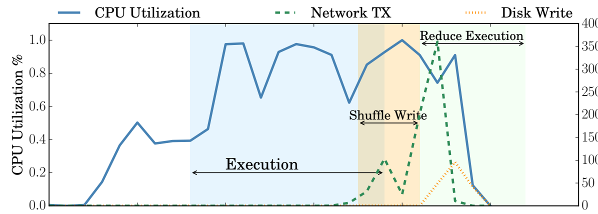


Figure 10: CPU utiliazation and I/O throughput of a node during a Spark single shuffle application With SCache

- Intra-shuffle unit: SCache also needs to decide the priority among storage units inside a shuffle unit. Referring to the task scheduling inside a task set of Spark, the tasks with smaller ID will be scheduled at first under one locality level. Based on this, SCache can assign the storage unit with larger tasks ID with lower priority.

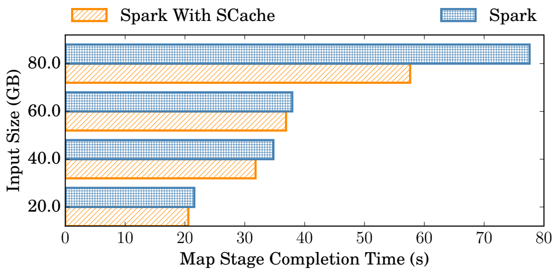
In a word, SCache selects the shuffle unit with lowest priority and evicts the storage units by intra-shuffle priority.

5 Evaluation

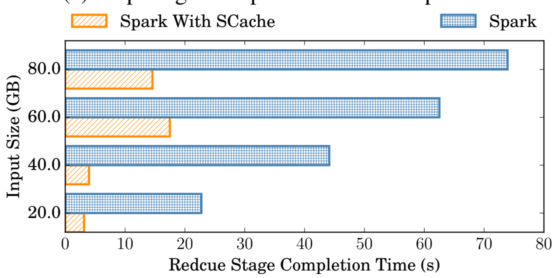
In this section, we present the evaluation result of Spark with SCache comparing with the original Spark. We first run simple DAG computing jobs with two stages to analyze the impact of shuffle optimization on the scope of the hardware resources. The shuffle dependency between two stages contains one shuffle. In addition, we run a shuffle heavy benchmark named Spark Terasort [9] to evaluate the SCache under different input with different partition schemes. In order to prove the performance gain of SCache with a real production workload, we also evaluate Spark TPC-DS [11] and and present the overall performance comparison. At last, we evaluate 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 occurs in overall performance comparison. 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 evaluations reveal the bandwidth is about 300 Mbps(see Figure 3).

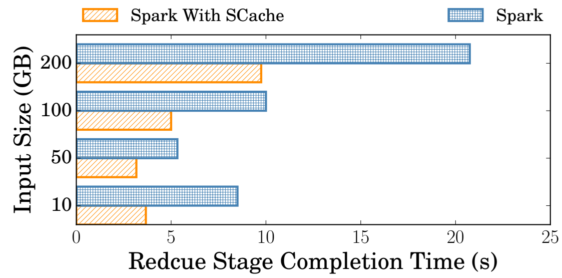


(a) Map Stage Completion Time Comparison

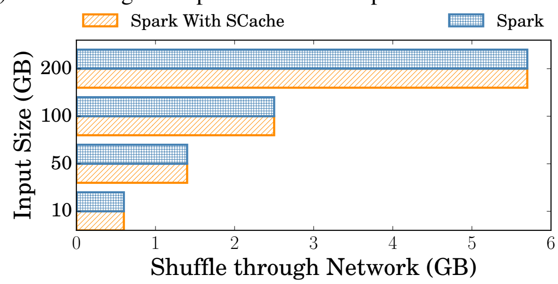


(b) Reduce Stage Completion Time Comparison

Figure 11: Stage Completion Time of Single Shuffle Test



(a) Reduce Stage Completion Time Comparison of First Shuffle



(b) Shuffle Data throuth Network Comparison of Second Shuffle

Figure 12: Terasort Evaluation

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 then Spark without SCache, the duration of Figure 10 is cut in half as well. A overlap between CPU, disk and network can be easily obeserved in Figure 10. That is, the I/O operations will never cut off the computing process with the fine granularity resource allocation. 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 computaion of reduce can start immediately afther scheduled. And this is the main performance gain we achieved on the scope of hardware utilization by SCache.

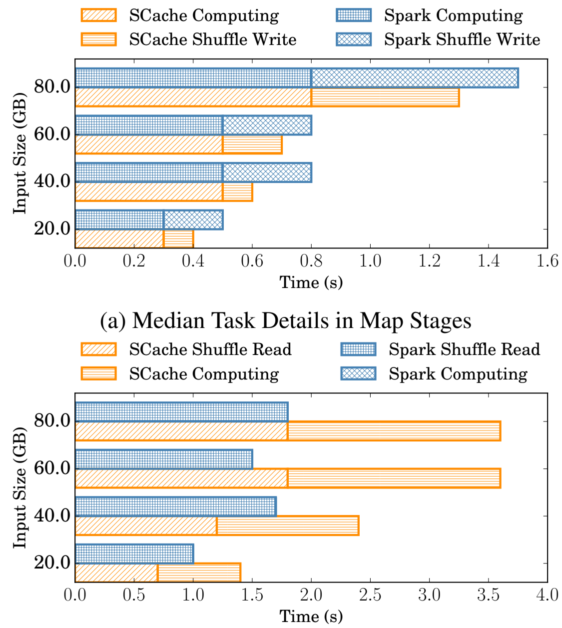
As shown in Figure 11, we run the single shuffle test with different input sizes in the cluster. For each stage, we run 5 rounds of tasks. The stage completion time is presented seperately in Figure 11a and Figure 11b. By running spark with SCache, the completion tiem of map stage can be reduce 10% on average. For reduce stage, instead, SCache achieves a 75% performance gain in the completion time of whole reduce stage.

Combining with Figure 13, 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 13. For a single map task, about 40% of shuffle write time can be eliminated by SCache (Figure 13a). Because the serialization of data is CPU intensive [31] 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 perfomance gain in the map stage completion time. On the reduce side, instead, the network transfer contribute a significantly latency in shuffle read for a single task (Figure 13b). By doing shuffle data pre-fetch for the reduce tasks in Figure 13b, the shuffle read time decreases 100%, which means shuffle data pre-fetch almost hide all the explict network transfer in the reduce stage. In overall, SCache can help Spark decrease about 89% time in the whole shuffle process. In addition, heuristic reduce tasks scheduling acheives better load balance in cluster than the Spark default FIFO scheuling 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.

5.3 Terasort

In this part, we evaluate the both Teraort [9]. Terasort [9] is a shuffle intensive benchmark for distributed system analystics. It consists of two consecutive shuffles. The first shuffle read the input data and use a customed hash partition function for re-partitioning. The second shuffle partitions the data through a range partitoner. As the range bounds set by range partitioner almost match the same pattern of the first shuffle, for each reduce task, almost 93% of input data is from one particular map task. It makes the shuffle data transferred through network extreamly small under Spark locality preferred task scheduling. So we take the second shuffle as a extream case to evaluate the scheduling locality for SCache.

As shown in Figure 12a, we present the first shuffle as the evalution of shuffle optimization. At the same time, we use the second the shuffle to evaluate in the dimension of scheduling locality (Figure 12b). For the first shuffle, Spark with SCache runs 2 × faster during the reduce stage with the input data in a range from 10GB to 200GB. At the same time, Figure 12b reveals that SCache pre-scheduling produces exactly same network traffic of second shuffle as Spark, which implies that SCache pre-scheduling can obtain the best locality while balancing the load. In contrast, Spark delays scheduling reduce tasks with the shuffle map output to achieve this optimal.



(a) Median Task Details in Map Stages

(b) Median Task Details in Reduce Stages

Figure 13: Median Task Completion Time Comparison of Single Shuffle Test

5.4 Production Workload

We also evalute some shuffle heavy queries from TPC-DS [10]. 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 the considered as the strandardized 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 14, 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. 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 compeletion time of queries, we believe that our shuffle optimization is promising.

5.4.1 Single Shuffle Test

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 15, the overhead of sampling only grows with the increasing of input size on each node. But it keep 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 [23] is a self-tuning system in Hadoop. The basic idea is to get sampled data statics for tuning system parameters (e.g. slowstart, map and reduce slot ratio, etc). However, these parameters cannot be changed once the jobs begin. DynMR [33] 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 in both map and reduce phases. iShuffle [16] 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 nor schedule multiple rounds of reduce tasks. iHadoop [20] 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 [37] delays tasks assignment to get better data locality, which can reduce the network traffic. ShuffleWatcher [12] delays shuffle fetching when network is saturated. At the same time,

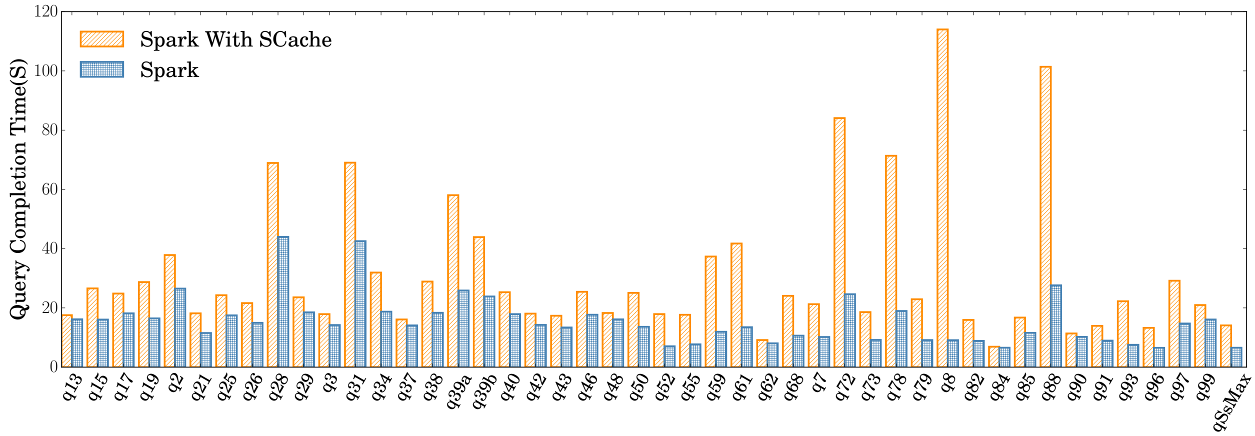


Figure 14: TPC-DS Benchmark Evaluation

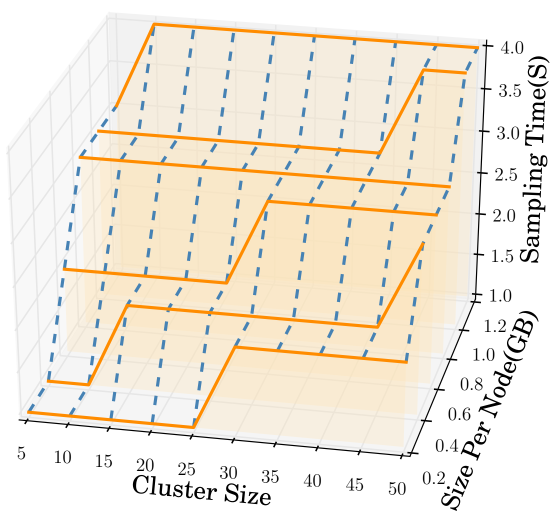


Figure 15: Sampling Overhead

it achieves better data locality. Both Quincy [27] and Fair Scheduling [36] 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 mitigate shuffle overhead between DAG computing stages. And the evaluation results show a promising improvement. But we realize some limitations of SCache.

Fault tolerence 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 pre-scheduling results 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 [25] to provide constantly service. If a failure happens on a worker, the optimization of tasks on that node will fail. It also violates of all-or-nothing constraint. A promising way to solve the failure on worker is to select some backup nodes to store repcliations of shuffle data during pre-scheduling to prevent the worker failure. We believe combing the high speed of network and memory is a better choice for fault tolerance. 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 [24] 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/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] memcached: a distributed memory object caching system. http://www.memcached.org/.

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

[9] Spark terasort. https://github.com/ehiggs/spark-terasort.

[10] Tpc benchmark ds (tpc-ds): The benchmark standard for decision support solutions including big data. http://www.tpc.org/tpcds/.

[11] Tpc-ds benchmark on spark. https://github.com/databricks/spark-sql-perf.

[12] 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.

[13] 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.

[14] 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.

[15] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[16] 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.

[17] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, and X. Li. Preemptive reducetask scheduling for fair and fast job completion. In *ICAC*, pages 279–289, 2013.

[37] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, 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.

[38] 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.