SCache: Efficient and General Shuffle Management for DAG Computing Frameworks

Rui Ren, Zhongxuan Wu, Zhouwang Fu, Tao Song, Zhengwei Qi, *Member, IEEE,* Haibing Guan, *Member, IEEE,*

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*, an open source plug-in 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%, and the overall completion time of TPC-DS queries improves 40% on average. We have implemented SCache as the external shuffle service and co-scheduler on both Apache Spark and Apache Hadoop. 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%, and the overall completion time of TPC-DS queries improves 40% on average. On Apache Hadoop, Scache optimize end-to-end Terasort completion time by 15%.

Index Terms—Distributed DAG frameworks, Shuffle, Optimization

1 Introduction

Recent years have witnessed widespread use of sophisticated frameworks such as Dryad [1], Spark [2], and Apache Tez [3]. Despite the differences among data-intensive frameworks, their communication is always structured as a shuffle phase, which takes place between successive computation stages. Such shuffle phase places significant burden for both the disk and network I/O, thus heavily affecting the end-to-end application performance. For instance, a MapReduce trace analysis from Facebook shows that shuffle accounts for 33% of the job completion time on average, and up to 70% in shuffle-heavy jobs [4].

Although continued efforts of performance optimization have been made among a variety of computing frameworks [5], [6], [7], [8], [9], [10], the shuffle is often poorly optimized in practice. In particular, we observe that one major deficiency lies in a lack of fine-grained, coordinated management among different system resources. As Figure 1 shows, the *shuffle write* is responsible for writing intermediate results to disk, which is attached to the tasks in ancestor stage (i.e., map task). And the *shuffle read* fetches intermediate results from *remote* disks through network, which is commonly integrated as part of the tasks in descendant stage (i.e., reduce task). Once scheduled, a fixed bundle of resources (i.e., CPU, memory, disk and network) named *slot* is assigned to each of the computation task, and the resources are released only after the task finishes. Such task aggregation together with the coarse-grained scheduling effectively simplifies task management. However, since each phase receives a fixed bundle of resources, attaching the *I/O intensive* shuffle phase to the *CPU/memory intensive* computation phase results in a poor multiplexing between computational and I/O resources.

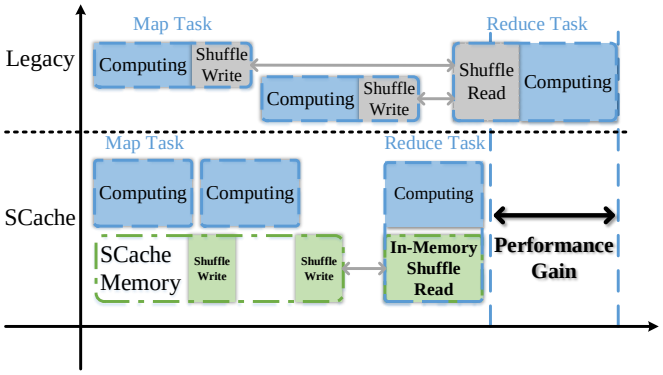


Fig. 1: Workflow Comparison between Legacy DAG Computing Frameworks and Frameworks with SCache

Moreover, the shuffle read phase introduces all-to-all communication pattern across the network, and such network I/O procedure is also poorly coordinated. Note that the shuffle read phase starts fetching data only after all the data from its ancestor stage by default. As a result, all the corresponding reduce tasks start fetching shuffle

- *R. Ren ...E-mail: renrui@sjtu.edu.cn*
- *W. Fu and Z. Wu ...E-mail: {wuchunghsuan, frankfzw}@sjtu.edu.cn*
- *T. Song ...E-mail: songt333@sjtu.edu.cn*
- *Z. Qi ...E-mail: qizhwei@sjtu.edu.cn*
- *H. Guan is with Shanghai Key Laboratory of Scalable Computing and Systems, and Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China.E-mail: hbguan@sjtu.edu.cn*

data almost simultaneously. Such synchronized network communication causes a burst demand for network I/O, which in turn greatly enlarges the shuffle read completion time. To desynchronize the network communication, an intuitive way is to launch some tasks in the descendent stage earlier such as *early start* from Apache Hadoop [11]. However, such early start is by no means a panacea. This is because although the reduce tasks can start fetching data earlier, the computation can only take place after all the data is ready. Since each phase receives a fixed bundle of resources, starting a reduce task early always introduces an unnecessary early allocation of slot.

To make things worse, we note that the above deficiencies generally exist in most of the DAG computing frameworks. As a result, even we can effectively resolve the above deficiencies by modifying one framework, updating one application at a time is impractical given the sheer number of computing frameworks available today.

In order to visually describe above-mentioned resources scheduling and analyze deficiencies, we propose *Framework Resources Quantification*(FRQ) model. FRQ model quantifies computing and I/O resources and displays resources scheduling strategy of DAG framework in time dimension. We use FRQ model to help us analyze the deficiencies of resources scheduling and optimize it.

Can we efficiently optimize the data shuffling without significantly changing every framework? In this paper, we answer this question in the affirmative with S(huffle)Cache, an opne source plug-in system which provides a shuffle-specific optimization for different DAG computing frameworks. Specifically, SCache takes over the whole shuffle phase from the underlying framework by providing a cross-framework API for both shuffle write and read. SCache's effectiveness lies in the following two key ideas. First, SCache decouples the shuffle read and write from both map and reduce tasks. Such decoupling effectively enables fine-grained resource management and better multiplexing between the computational and I/O resources. In addition, SCache pre-schedules the reduce tasks and pre-fetches the shuffle data to the location of the reduce tasks without launching them. Such pre-scheduling and pre-fetching effectively desynchronize the network I/O operation, while avoiding the waste of computational resources compared to the early start mechanism.

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

The main challenge to achieve this optimization is *pre-scheduling reduce tasks*. It is not critical for the simple DAG computing such as Hadoop MapReduce [12]. Unfortunately the complexity of DAG can amplify the defects of naïve pre-scheduling schemes. In particular, randomly assign reduce tasks might result in a collision of two heavy tasks on one node. This collision can aggravate data skew, thus hurting the performance. To address this challenge, we propose a heuristic scheme to predict the shuffle output distribution and pre-schedule reduce tasks (Section 3).

The second challenge is the *in-memory data management*. 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. To minimize the reserved memory while maximizing the performance gain of optimization and memory utilization, we propose two constraints: all-or-nothing and context-aware (Section 4.2).

We have implemented SCache and a customized Apache Spark [13]. The performance of SCache is evaluated with both simulations and testbed experiments on a 50-node Amazon EC2 cluster. We conduct basic test GroupByTest. We also evaluate Terasort [14] benchmark and standard workloads like TPC-DS [15] for multi-tenant modeling. In a nutshell, SCache can eliminate explicit shuffle process by at most 89% in varied application scenarios. More impressively, SCache reduces 40% of overall completion time of TPC-DS queries on average.

We have implemented SCache, a customized Apache Spark [13] and a customized Apache Hadoop. We have also designed a mathematic model called *Framework Resources Quantification*(FRQ) model. We use FRQ model to analyze the principle of SCache and calculate the time of each phase of computing job. The performance of SCache is evaluated with both simulations and testbed experiments on a 50-node Amazon EC2 cluster on both Apache Spark and Apache Hadoop. On Apache Spark, we conduct basic test GroupByTest. We also evaluate Terasort [14] benchmark and standard workloads like TPC-DS [15] for multi-tenant modeling. On Apache Hadoop, we focus on Terasort benchmark. In a nutshell, SCache can eliminate explicit shuffle process by at most 89% in varied application scenarios. More impressively, SCache reduces 40% of overall completion time of TPC-DS queries on average on Apache Spark. On Apache Hadoop, Scache optimize end-to-end Terasort completion time by 15%.

2 Background and Observations

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, shuffle is designed to achieve an all-to-all data blocks transfer among nodes. For a clear illustration, we use *map tasks* to define the tasks that generate shuffle data and use *reduce tasks* to define the tasks that consume shuffle data.

Overview of shuffle process. Each map task partitions the result data (key, value pair) into several buckets according to the partition function (e.g., hash). The total number of buckets equals the number of tasks in the next step. The shuffle process can be further split into two parts: *shuffle write* and *shuffle read*. The partitioned shuffle output data

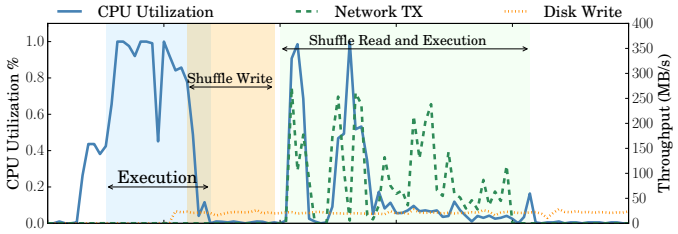


Fig. 2: CPU utilization and I/O throughput of a node during a Spark single shuffle application

will be spilled to local persistent storage during shuffle write. Shuffle read starts at the beginning of reduce tasks. It fetches data as reduce input from both remote and local storage.

Impact of shuffle process. Shuffle process is I/O intensive, which might introduces a significant latency to the application. Reports show that 60% of MapReduce jobs at Yahoo! and 20% at Facebook are shuffle intensive workloads [16]. 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 [4].

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 2 as an example. This job has 2 rounds of tasks for each node. We have marked 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.

2.2.1 Coarse Granularity Resource Allocation

In Figure 2, when a slot is scheduled to a task, it will not be released until the the task completes *shuffle write*. On the reduce side, the network transfer of shuffle data introduces an explicit I/O delay during *shuffle read*. Meanwhile, shuffle is I/O intensive. Both *shuffle write* and *shuffle read* occupy the slot without involving much CPU. The current coarse slot — task mapping results in an inconsistency between resource demands and slot allocation thus decreases the resource utilization. To break this inconsistency, a finer granularity resource allocation scheme must be provided.

2.2.2 Synchronized Shuffle Read

Almost all reduce tasks start *shuffle read* simultaneously. The synchronized *shuffle read* requests cause a burst of network traffic. As shown in Figure 2, the data transfer put a great deal of stress on network bandwidth, which may results in network congestion and further hurts the performance of reduce stage.

2.2.3 Inefficient Persistent Storage Operation

At first, both shuffle write and read are tightly coupled with task execution, which results in a blocking style I/O operation. This blocking I/O operation along with synchronized shuffle read may introduce significant latency, especially in a I/O performance bounded cluster. Besides, the legacy of spilling shuffle data to disk is too conservative in modern cluster with large memory. Compared to input dataset, the size of shuffle data is relatively small. For example, shuffle size of Spark Terasort [14] is less than 25% of input data. The data reported in [17] also shows that the amount of data shuffled is less than input data, by as much as 10%-20%. On the other hand, numbers of memory based distributed storage system have been proposed [7], [18], [19] to move data back to memory. We argue that the memory capacity is large enough to store the short-living shuffle data with cautious management.

2.2.4 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 [20] suggests that *10-100 maps* per-node and *0.95 or 1.75 $\times$ no. of nodes $\times$ no. of maximum container* per-node seem to be the right level of parallelism. Spark Configuration also recommends $2-3$ tasks per CPU core [21]. Since the shuffle data becomes available as soon as the end of map task execution. At the same time, the network is idle during the map stage (network utilization during map stage in Figure 2). If the destination host of the shuffle data can be predicted in priori, the property of multi-round can be leveraged to overlap the *shuffle read* operation.

Based on these observations, it is straightforward to come up with an optimization that starts shuffle read ahead of reduce stage to overlap the I/O operations in multi-round DAG computing tasks, and uses memory to store the shuffle data for further decreasing shuffle overhead. To achieve this optimization:

- Shuffle process should be decoupled from task execution to achieve a fine granularity scheduling scheme.
- Reduce tasks should be pre-scheduled without launching to achieve shuffle data pre-fetching.
- Shuffle process should be taken over and managed outside DAG frameworks to achieve a cross-framework optimization.

3 Shuffle Optimization

This section presents the detailed methodologies to achieve three design goals. The out-of-framework shuffle data management is used to decouple shuffle from execution and provide a cross-framework optimization. Two heuristic algorithms (Algorithm 1, 2) are used to achieve shuffle data pre-fetching without launching tasks.

3.1 Decouple Shuffle from Execution

During map tasks, the partitioner takes a set of key-value pairs as input and calculates the partition number for each of them by applying pre-defined the partition function. After that, it stores all key-value pairs in the corresponding

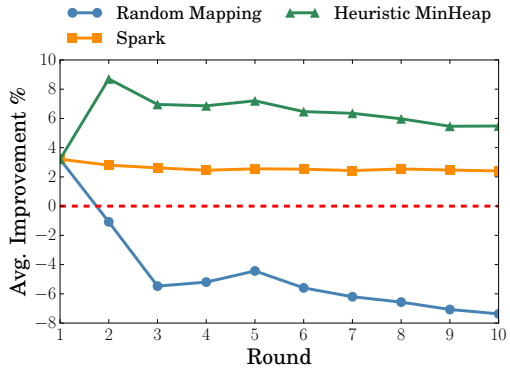


Fig. 3: Stage Completion Time Improvement of OpenCloud Trace

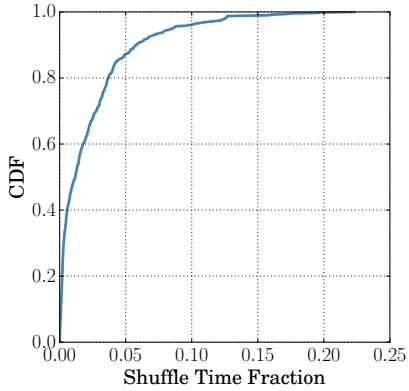


Fig. 4: Shuffle Time Fraction CDF of OpenCloud Trace

data blocks. Each of the block contains the key-value pairs for one reduce partition. At the same time, blocks will be spilled to disk. In order to avoid blocking the slot by disk operation, we use memory copy to hijack shuffle data from map tasks. By doing this, a slot can be released as soon as a task finishes CPU intensive computing. From the perspective of reduce task, shuffle read is decoupled by pre-fetching shuffle data to local SCache client before reduce tasks start.

3.2 Pre-schedule with Application Context

The pre-scheduling and pre-fetching start when the collected shuffle data exceeds the threshold. This is the most critic step toward the optimization. The task — node mapping is not determined until tasks are scheduled by the scheduler of DAG framework. Once the tasks are scheduled, the slots will be occupied to launch them. On the other hand, the shuffle data cannot be pre-fetched without the readiness of task — node mapping. To get rid of this dilemma, we propose a co-scheduling scheme with two heuristic algorithms (Algorithm 1, 2). That is, the task — node mapping is established ahead of DAG framework scheduler, and it is enforced to DAG scheduler before the real scheduling.

3.2.1 Problem of Random Mapping

The simplest way of pre-scheduling is mapping tasks to different nodes randomly. It only guarantees that each node run same number of tasks. As shown in Figure 3, we use

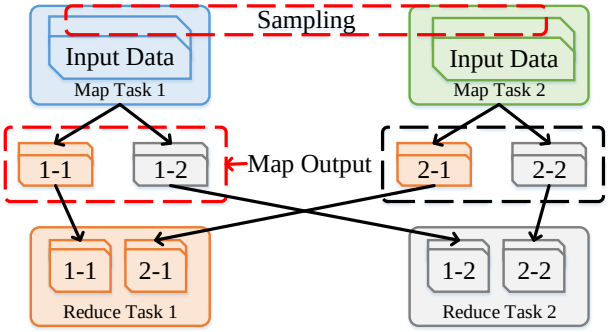


Fig. 5: Shuffle Data Prediction

traces from OpenCloud [1] for the simulation to evaluate the impact of different pre-scheduling algorithms. The baseline (red dot line in Figure 3) is the stage completion time with Spark default scheduling algorithm. And then we remove the shuffle read time of each task, and run the simulation under three different schemes: random mapping, Spark FIFO, and our heuristic MinHeap. Note that most of the traces from OpenCloud is shuffle-light workload as shown in Figure 4. The average shuffle read time is 3.2% of total reduce completion time.

Random mapping works well when there is only one round of tasks. But the performance collapses as the round number grows. It is because that data skew commonly exists in data-parallel computing [22], [23], [24]. Several heavy tasks might be assigned on the same node. This collision then slows down the whole stage and makes the performance even worse than the baseline. In addition, randomly assigned tasks also ignore the data locality between shuffle map output and shuffle reduce input, which might introduces extra network traffic in cluster.

3.2.2 Shuffle Output Prediction

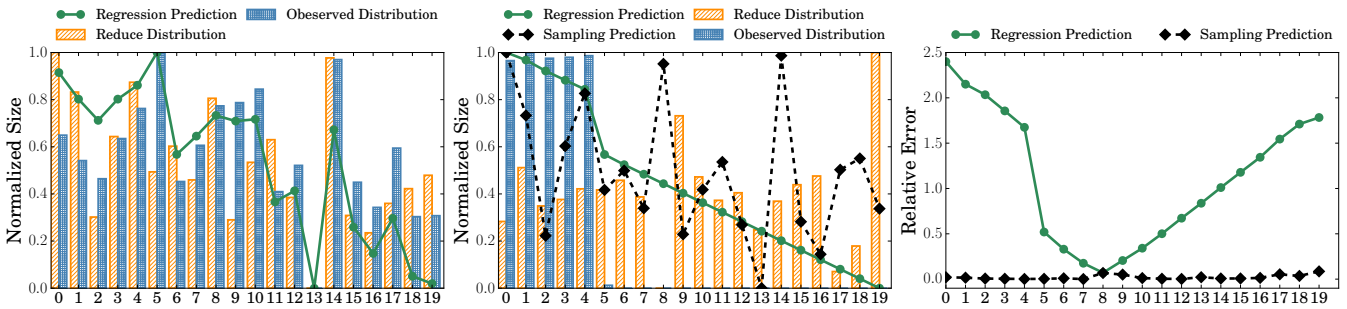
The problem of random mapping was obviously caused by application context (e.g., shuffle data size) unawareness. Note that the optimal schedule decision can be made under the awareness of shuffle dependencies number, partition number, and shuffle size for each partition. The first two of them can be easily extracted from DAG information. The scheduling can be made with the "prediction" of shuffle size.

According to the DAG computing process, the shuffle size of each reduce task is decided by input data, map task computation, and hash partitioner. Each map task produces a data block for each reduce task. The size of each reduce partition can be calculated $reduceSize_i = \sum_{j=0}^{m} BlockSize_{ji}$ (m is the number of map tasks). $BlockSize_{ji}$ represents the size of block which is produced by map $task_j$ for reduce $task_i$ (e.g., block '1-1' in Figure 5).

For the simple DAG applications such as Hadoop MapReduce [12], the $BlockSize_{ji}$ can be predicted with decent accuracy by liner regression model based on observation that the ratio of map output size and input size are invariant given the same job configuration [25], [26].

But the sophisticated DAG computing frameworks like Spark introduce more uncertainties. For instance, the cus-

1. http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html



(a) Linear Regression Prediction of Hash Partitioner (b) Linear Regression and Sampling Prediction of Range Partitioner (c) Prediction Relative Error of Range Partitioner

Fig. 6: Reduction Distribution Prediction

tomized partitioner might bring large inconsistency between observed map output blocks distribution and the final shuffle data distribution. In Figure 6, we use different datasets with different partitioners, and normalize the distribution to $0-1$ to fit in one figure. In Figure 6a, we use a random input dataset with the hash partitioner. In Figure 6b, we use a skew dataset with the range partitioner of Spark [13]. The observed map outputs are randomly picked. As we can see, in hash partitioner, the distribution of observed map output is close to the final reduce input distribution. The prediction results also turn out to be good. However, this inconsistency results in a deviation in linear regression model. To handle this inconsistency, we introduce another methodology named weighted reservoir sampling. The $BlockSize_{ji}$ can be calculated by

$$BlockSize_{ji} = InputSize_j \times \frac{sample_i}{s \times p} \\ sample_i = \text{number of samples for } reduce_i \tag{1}$$

For each partition of map task, we use classic reservoir sampling to randomly pick $s \times p$ of samples, where p is the number of reduce tasks and s is a tunable number. After that, the map function is called locally to process the sampled data (*sampling* in Figure 5). The final sampling outputs are collected with the $InputSize$ of each map partition which is used as the weight for each set of samples.

In Figure 6b, we set s equals 3, the result of sampling prediction is much better even in a very skew scenario. The variance of the normalization between sampling prediction and reduce distribution is because the standard deviation of the prediction result is relatively small compared to the average prediction size, which is 0.0015 in this example. Figure 6c further proves 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 large relative error.

To avoid extra overhead, the sampling prediction will be triggered only when the range partitioner or customized non-hash partitioner occurs. We will show the detail evaluation of sampling in the Section 6.

During the phase of shuffle output prediction, the composition of each reduce partition is calculated as well. We define $prob_i$ as

$$prob_i = \max_{0 \le j \le m} \frac{BlockSize_{ji}}{reduceSize_i} \\ m = \text{number of map tasks} \tag{2}$$

This parameter is used to achieve a better locality while performing shuffle scheduling.

3.2.3 Heuristic MinHeap Scheduling

In order to balance load on each node while reducing the network traffic, we present a heuristic MinHeap scheduling algorithm (Algorithm 1) for single shuffle. Heuristic MinHeap can be divided into two rounds. In the first round (i.e., the first $while$ in $SCHEDULE$), the reduce tasks are first sorted by size in a descending order. For hosts, we use a min-heap according to size of assigned tasks to maintain the priority. So that the tasks can be distributed evenly in the cluster. In the second round, the task — node mapping will be adjusted according to the locality. The $SWAP\_TASKS$ will only be trigged when the $host\_id$ of a reduce task is not equal the $assigned\_id$. For a task which contains at most $prob$ data from $host$, the normalized probability $norm$ is calculated as a bound of performance degradation. We set maximum $upper\_bound$ of performance degradation equals to 10% that can be traded for locality (in extreme skew scenarios). Inside the $SWAP\_TASKS$, tasks will be selected and swapped without exceeding the $upper\_bound$ of each host. Combining these information helps the scheduler make a more balanced task — node mapping than the naïve Spark FIFO scheduling algorithm. We use the OpenCloud 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%). This is the by-product optimization harvested from shuffle size prediction.

3.2.4 Cope with Multiple Shuffles

Multiple shuffles commonly exist in modern DAG computing. The techniques mentioned in Section 3.2.2 can only handle the ongoing shuffle. For those pending shuffles, it is impossible to predict the sizes. This dilemma can be relieved by having all map tasks of shuffle to be scheduled by DAG framework simultaneously. But doing this introduces large overhead such as extra task serialization. To avoid violating

the optimization from framework, we present Accumulate Heuristic Scheduling algorithm (Algorithm 2) to cope with multiple shuffles.

As illustrated in $M\_SCHEDULE$, the size of reduce on each node of previously scheduled $shuffles$ are counted. When a new shuffle starts, the $M\_SCHEDULE$ is called to schedule the new one accumulatively. The $size$ of each reduce and its corresponding $porb$ and $host$ in $p\_reduces$ are updated with data of $shuffles$ before $SCHEDULE$ is called. When the new task — node mapping is available, if the new $assigned\_host$ of a reduce does not equal to the original one, the re-shuffle will be triggered to transfer data to a new host for further computing. This re-shuffle is rare since the previously shuffled data in a reduce partition contributes a huge composition. It means in the schedule phase, the $SWAP\_TASKS$ can help revise the scheduling to match the previous mapping as much as possible while maintaining the good load balance.

Algorithm 1 Heuristic MinHeap Scheduling for Single Shuffle

procedure SCHEDULE($m, host\_ids, p\_reduces$)
  $m \leftarrow$ partition number of map tasks
  $R \leftarrow$ sort $p\_reduces$ by size in descending order
  $M \leftarrow$ min-heap $\{host\_id \rightarrow ([reduces], size)\}$
  $idx \leftarrow \text{len}(R) - 1$
  while $idx \geq 0$ **do** ▷ Schedule reduces by MinHeap
    $M[0].size \mathrel{+}= R[idx].size$
    $M[0].reduces.append(R[idx])$
    $R[idx].assigned\_host \leftarrow M[0].host\_id$
    Sift down $M[0]$ by $size$
    $idx \leftarrow idx - 1$
  $max \leftarrow$ maximum size in M
  Convert M to mapping $\{host\_id \rightarrow ([rid\_arr], size)\}$
  for all $reduce$ in R **do** ▷ Heuristic swap by locality
    if $reduce.assigned\_id \neq reduce.host\_id$ **then**
      $p \leftarrow reduce.prob$
      $norm \leftarrow (p - 1/m) / (1 - 1/m) / 10$
      $upper\_bound \leftarrow (1 + norm) \times max$
      SWAP_TASKS($M, reduce, upper\_bound$)
  return M
procedure SWAP_TASKS($M, reduce, upper\_bound$)
  $reduces \leftarrow M[reduce.host\_id].reduce$
  $candidates \leftarrow$ Select from $reduces$ that $assigned\_id \neq host\_id$ **and** total size closest to $reduce.size$
  $\Delta size \leftarrow sizeOf(candidates) - reduce.size$
  $size\_host \leftarrow M[reduce.host\_id].size - \Delta size$
  $size\_assigned \leftarrow M[reduce.assigned\_id].size + \Delta size$
  if $size\_host \leq upper\_bound$ **and**
    $size\_assigned \leq upper\_bound$ **then**
    Swap $candidates$ and $reduce$
    Update $size$ in M
    Update $assigned\_host$ in $candidates$ and $reduce$

Algorithm 2 Accumulate Heuristic Scheduling for Multi-Shuffles

procedure M_SCHEDULE($m, host\_id, p\_reduces, shuffles$)
  $m \leftarrow$ partition number of map tasks ▷ $shuffles$ is the previous schedule result
  for all r in $p\_reduces$ **do**
    $r.size \mathrel{+}= shuffles[r.rid].size$
    if $shuffles[r.rid].size \geq r.size \times r.prob$ **then**
      $r.prob \leftarrow shuffles[r.rid].size / r.size$
      $r.host\_id \leftarrow shuffles[r.rid].assigned\_host$
  $M \leftarrow SCHEDULE(m, host\_id, p\_reduces)$
  for all $host\_id$ in M **do** ▷ Re-shuffle
    for all r in $M[host\_id].reduces$ **do**
      if $host \neq shuffles[r.rid].assigned\_host$ **then**
        Re-shuffle data to $host$
        $shuffles[r.rid].assigned\_host \leftarrow host$
  return M

4 IMPLEMENTATION

This section presents an overview of the implementation of SCache – an open source cross-framework shuffle data management system with a DAG co-scheduler. Here we use Spark as an example of DAG framework to illustrate the work flow of shuffle optimization. We will first present the 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 of three components: a distributed shuffle data management system, a DAG co-scheduler, and a daemon inside Spark. As shown in Figure 7, SCache employs the legacy master-slaves architecture like GFS [27] for shuffle data management system. The master node of SCache coordinates the shuffle blocks globally with application context. The worker node reserves memory to store blocks. The coordination provides two guarantees: (a)data is stored in memory before tasks start and (b)data is scheduled on-off memory with all-or-nothing and context-aware constraints. The daemon bridges the communication between Spark and SCache. The co-scheduler is dedicated to pre-schedule reduce tasks with DAG information and enforce the scheduling results to Spark scheduler.

When a Spark job starts, the DAG will be first generated. During the DAG generation, the shuffle dependencies among Resilient Distributed Datasets (RDDs) will then be submitted through the daemon process in Spark driver. The SCache master recognizes all shuffle dependencies in a RPC call as a shuffle scheduling unit. For each shuffle dependency, the shuffle ID, the type of partitioner, the number of map tasks, and the number of reduce tasks are included. If there is a specialized partitioner, such as range partitioner, in the shuffle dependencies, the daemon will insert a sampling application before the dependent RDDs. We will elaborate the sampling procedure in the Section 4.1.1.

For the hash partitioner, when a map task finishes com-

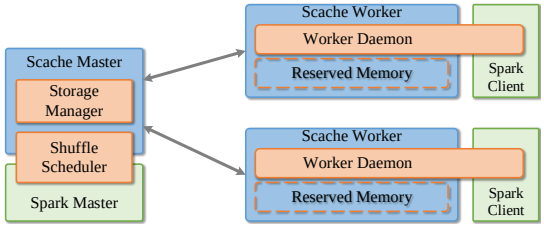


Fig. 7: SCache Architecture

puting, the SCache daemon process will transfer the map output data from Spark executor's JVM to the reserved memory through memory copy. After that the slot will be released without blocking on disk operations. When the shuffle map output is received, the SCache worker will notify the master of the block ID and reduce size distribution of this block (see map output in Figure 5). If the collected map output data reach the observation threshold, the DAG co-scheduler will run the scheduling Algorithm 1 and 2 to pre-schedule the reduce tasks and then broadcast the scheduling result to start pre-fetching on each worker. More specifically, when a map task is finished, each node will receive a broadcast message. SCache worker will filter the reduce tasks ID that will be launched on itself and start pre-fetching shuffle data from the remote. To enforce SCache pre-scheduled tasks – node mapping, we insert some lines of codes in Spark DAG Scheduler. For RDDs with shuffle dependencies, Spark DAG scheduler will consult SCache master to get the preferred location for each partition and set *NODE_LOCAL* locality level on corresponding reduce tasks.

4.1.1 Reservoir 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 the daemon in Spark driver. The daemon then inserts a sampling job before the corresponding RDD. The sampling job uses a reservoir sampling algorithm [28] on each partition of RDD. For the sample number, we set the size to $3 \times number\ of\ partitions$ for balancing overhead and accuracy (it can be tuned by configuration). The sampling job randomly selects some items and performs a local shuffle with partitioner (see Figure 8). At the same time, the items number is counted as the weight. These sampling data will be aggregated by reduce ID on SCache master to predict the reduce partition size. After the prediction, SCache master will call Algorithm 2 and 1 to do the scheduling.

4.2 Memory Management

As mentioned in Section 2.2, though the shuffle size is relatively small, memory management should still be cautious enough to limit the effect of performance of DAG framework. When the size of cached blocks reaches the limitation of reserved memory, SCache flushes some of them to the disk temporarily, and re-fetches them when some cached shuffle blocks are consumed or pre-fetched. To achieve the maximum overall improvement, SCache leverages two constraints to manage the in-memory data — all-or-nothing and context-aware-priority.

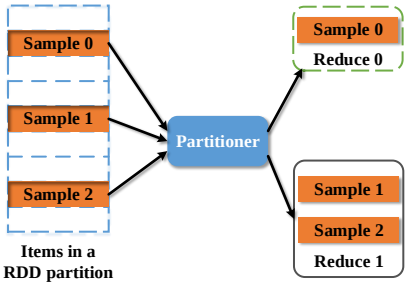


Fig. 8: Reservoir Sampling of One Partition

4.2.1 All-or-Nothing Constraint

Memory cached shuffle blocks can speed up the reduce task execution process. But this acceleration of a single task is necessary but insufficient for a shorter stage completion time. Based on the observation in Section 2.2.4, in most cases one single stage contains multi-rounds of tasks. If one task misses a memory cache and exceeds the original bottleneck of this round, that task might become the new bottleneck and further slow down the whole stage. PACMan [8] has also proved that for multi-round stage/job, the completion time improves in steps when $n \times number\ of\ tasks\ in\ one\ round$ of tasks have data cached simultaneously. Therefore, the cached shuffle blocks need to match the demand of all tasks in one running round at least. We refer to this as the all-or-nothing constraint.

According to all-or-nothing constraint, SCache master leverages the pre-scheduled results to determine the bound of each round, and sets corresponding blocks as the minimum unit of storage management. For those incomplete units, SCache will mark them as the lowest priority.

4.2.2 Context-Aware-Priority Constraint

Unlike the traditional cache replacement schemes such as MIN [29], the cached shuffle data will only be used once (without failure) in DAG computing. That is, the effort of improving hit rate in most legacy schemes can not benefit DAG application, while their approaches can easily violate the all-or-nothing constraint. SCache also leverages application context to select victim storage units when the reserved memory is full.

At first, SCache master searches for the incomplete units and flushes all belonging blocks to disk cluster-widely.

If all the units are completed, SCache selects victims based on two factors — *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 among task sets, which leads to a higher priority for those having more tasks unfinished. So SCache sets priorities from high to low in a descending order of remaining storage units of a shuffle unit. 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.
- Intra-shuffle unit: SCache also needs to decide the priority among storage units inside a shuffle unit. According to the task scheduling inside a task set of Spark, the

tasks with smaller ID will be scheduled firstly. Based on this, SCache can assign the lower priority to storage units with larger task ID.

4.3 Cost of adapting DAG frameworks

SCache provides API through RPC, such as *putBlock(blockId)*, *getBlock(blockId)*, and *getScheduleResult(shuffleId)*. The concise design makes it easy to adapt DAG frameworks to enable SCache optimization. For example, it only takes about 500 lines of code in Spark to integrate SCache. By a glance of Hadoop source code, we believe that the costs of enabling SCache on MapReduce [11] and YARN [30] based DAG computing framework, such as Tez [3], are also very low.

4.4 Fault tolerance

Since fault tolerance is not a crucial goal of SCache, we have not implemented fault tolerance mechanism. We plan to implement SCache master with Apache ZooKeeper [31] to provide constantly service. For SCache worker, a promising way to prevent failure is to select some backup nodes to store replications of shuffle data during pre-scheduling. In addition, there are also advanced fast recovery techniques such as FineFRC [32]. We leave this to the future work.

5 Framework Resources Quantification Model

In this chapter, we introduce *Framework Resources Quantification*(FRQ) model to describe the performance of DAG frameworks. FRQ model quantifies computing and I/O resources and displays them in time dimension. According to FRQ model, we can calculate the execution time required by the application under any circumstances, including different DAG framework, hardware environment, and so on. Therefore FRQ model is able to help us analyze the resources scheduling of DAG framework and evaluate their performance. We will first introduce FRQ model in Subsection 5.1. In the following Subsection 5.2, we will use FRQ model to describe three different computation job and analyze their performance. In the last Subsection 5.3, we will use the actual experimental results to verify the FRQ model.

5.1 The FRQ Model

The current distributed parallel computing frameworks mostly use DAG(Directed acyclic graph) to describe computation logic. A shuffle phase is required between each adjacent DAG computation phase. In order to better analyze the relationship between the computation phase and the shuffle phase, we propose FRQ model. After quantifying computing and I/O resources, FRQ model is able to describe different resource scheduling strategies. For convenience, we introduce FRQ model by taking a simple MapReduce as an example in this section.

Figure 9 shows how the FRQ model describes a MapReduce task. FRQ model has five input parameters:

- Input Data Size(D): The data size of the computation phase.
- Data Conversion Rate(R): The conversion rate of the input data to the shuffle data during a computation phase. This conversion rate depends on the algorithm used in the computation phase.
- Computation Round Number(N): The number of rounds needed to complete the computation phase using the current computation resources. The number of rounds depends on the current computation resources and the settings of the computation job. Take Hadoop MapReduce as an example, suppose we have 50 CPUs and enough memory, the Map phase consists of 200 map tasks. Then we need 4 rounds of computation to complete the Map phase.
- Computation Speed(V_i): The computation speed for each computation phase. The computation speed depends on the algorithm used in the computation phase.
- Shuffle Speed($V_{Shuffle}$): Transmission speed during shuffle. Shuffle speed depends on Network and storage device bandwidth.

We can calculate the execution time of each phase of the job with these five parameters. Obviously, the total execution time of job in this case is the sum of the Map phase time and Reduce phase time:

$$T_{Job} = T_{Map} + T_{Reduce} \tag{3}$$

Map phase time depends on input data size and Map computation speed:

$$T_{Map} = \frac{D}{V_{Map}} \tag{4}$$

The Reduce phase time formula is as follows:

$$T_{Reduce} = \frac{D \times R}{V_{Reduce}} + K \times T_{P\_Shuffle} \tag{5}$$

$\frac{D \times R}{V_{Reduce}}$ represents the ideal computation time, and $K \times T_{P\_Shuffle}$ represents the calculated overhead. K is empirical value.The overhead depends on the parallel time of shuffle phase and the Reduce phase.The parallel time is denoted by $T_{P\_Shuffle}$. And the total time of shuffle phase is represented by $T_{Shuffle}$. Because the computatinon of the Reduce phase relies on the data transfer results of the Shuffle phase, a portion of the computation in the Reduce phase need to wait for the transfer results. The overhead is caused by these waiting. The FRQ model uses K to indicate the extent of the waiting.

$$T_{Shuffle} = \frac{D}{V_{Shuffle}} \tag{6}$$

For shuffle-heavy computing jobs, we can optimize the job completion time by reducing $T_{P\_Shuffle}$. Improving IO speed is a effective way to reduce shuffle time. Another optimization method is to use the idle IO resources in the Map phase for pre-fetching(see Figure 9). Both of the above methods can effectively reduce $T_{P\_Shuffle}$. When using FRQ model to describe a computation job, we can easily analyze the resource scheduling strategy of the computation framework. Different computing frameworks may use different resource scheduling strategies. FRQ model can evaluate the scheduling strategies of these computing frameworks and help us optimize them.

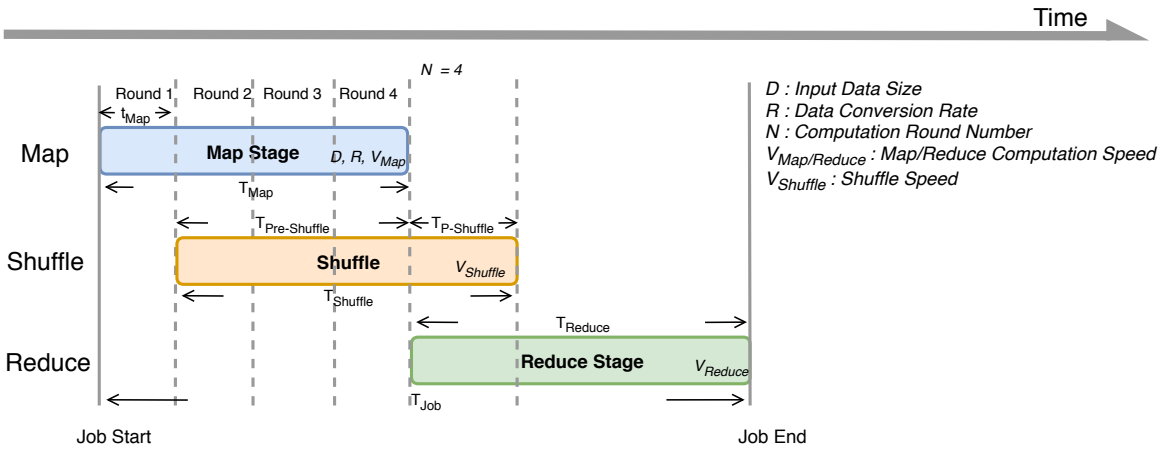
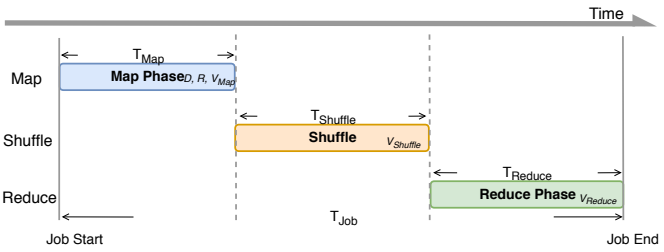
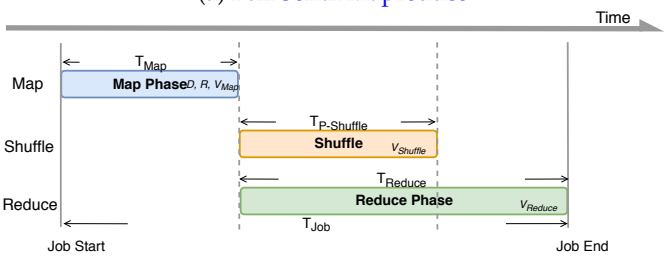


Fig. 9: FRQ Model

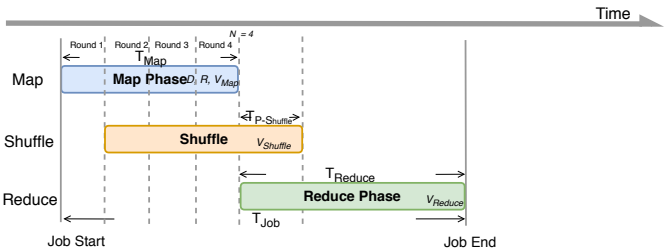


(a) Full Serial Mapreduce

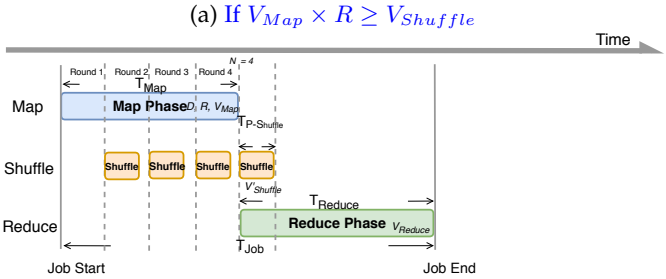


(b) Hadoop Mapreduce

Fig. 10: FRQ Model With Different Scheduling Strategies



(a) If $V_{Map} \times R \geq V_{Shuffle}$



(b) If $V_{Map} \times R < V_{Shuffle}$

Fig. 11: FRQ Model With SCache

5.2 Model Analysis

FRQ model can describe a variety of resource scheduling strategies. First, we analyze a naive scheduling strategy. As shown in Figure 10a, FRQ model describes a Mapreduce job that is fully serially executed. The parallel time of shuffle phase and the Reduce phase is 0, in which case $T_{P\_Shuffle}$ is 0. Therefore, the overhead of the Reduce phase is 0. But since shuffle and computation are serial execution, the total execution time of job becomes longer:

$$T_{Job} = T_{Map} + T_{Shuffle} + T_{Reduce} \quad (7)$$

Obviously, this is an inefficient scheduling strategy. No computing framework uses this scheduling method. Due to serialization, the IO resource is idle during the Reduce phase and Map phase. The scheduling strategy is naive and has a lot of room for optimization.

Figure 10b shows the scheduling strategy of Hadoop Mapreduce. In Hadoop Mapreduce, Shuffle phase and Reduce phase start at the same time. In this case, $T_{P\_Shuffle}$ is equal to $T_{Shuffle}$. Due to the increase in $T_{P\_Shuffle}$, the time of Reduce phase will increase(see equation 5). Because the Shuffle phase and the computation phase are executed in parallel, the total execution time of job is the sum of T_{Map} and T_{Reduce}(see equation 3). The execution time of Shuffle phase is hidden in the Reduce phase. This scheduling strategy is much more efficient than the one in Figure 1. However, after analyzing this model, we found that the IO resource in the Map phase is idle. This scheduling strategy can be optimized.

Figure 11 shows the scheduling strategy for Hadoop Mapreduce with SCache (Suppose N is 4). SCache starts pre-fetching and pre-scheduling in the Map phase. This scheduling strategy can make better use of resources and avoid IO resources being idle in the Map phase. According to the design of SCache pre-fetching, we found that using FRQ model to describe the scheduling strategy of SCache needs to distinguish two situations:

- $V_{Map} \times R \geq V_{Shuffle}$(Figure 11a): The meaning of $V_{Map} \times R$ is the speed at which shuffle data is generated. The meaning of the inequality is that the speed of

generating shuffle data($V_{Map} \times R$) is greater than or equal to the shuffle speed($V_{Shuffle}$). When the Round1 of the Map phase ends, the SCache starts shuffling data until the end of the shuffle phase. Due to shuffle speed is slower, the shuffle phase is uninterrupted. SCache transmit the shuffle data generated in the last round of Map phase during the Reduce phase. Therefore $T_{PShuffle}$ is equal to one-N of the total time of the shuffle phase:

$$T_{P\_Shuffle} = T_{Shuffle} - \frac{(N-1) \times T_{Map}}{N} \tag{8}$$

- $V_{Map} \times R < V_{Shuffle}$(Figure 11b): When the speed of generating shuffle data($V_{Map} \times R$) is less than the shuffle speed($V_{Shuffle}$), SCache needs to wait for shuffle data to be generated. As Figure 11b shown, the shuffle phase will be interrupted in each Round. Thus $T_{Pshuffle}$ is equal to the total time of shuffle ($T_{Shuffle}$) minus the time that shuffle is executed in the Map phase:

$$T_{P\_Shuffle} = T_{Shuffle} \times \frac{1}{N} \tag{9}$$

Compared to the original Hadoop Mapreduce resource scheduling strategy, Hadoop Mapreduce with SCache reduces $T_{P\_Shuffle}$ and thus lessens *Reduce Time*(T_{Reduce}). This is how pre-fetching optimizes the total execution time of job.

5.3 Model Verification

In order to verify FRQ model, we run experiment on two environments. The first environment is on Amazon EC2 and it has 50 m4.xlarge nodes as shown in Section 6.1. Another environment is in our lab. Out lab environment has 4 nodes and each nodes has 128GB and 32 CPUs. To simplify the calculation of the FRQ model, we use Hadoop Mapreduce as framwork(only three phases: *Map*, *Shuffle*, and *Reduce*) and Terasort as experimental application. We deploye Hadoop with SCache and without SCache on both environments.

Table 1 shows the calculational results of FRQ model in the lab environment. Workload is from 16 GB to 64 GB. D and N are set according to the application parameters. $R, V_{Map}, V_{Shuffle}$,and V_{Reduce} are calculated based on experimental results. K is the empirical value, we set K to 0.5 and 0.6, which reflects that $T_{P\_Shuffle}$ has less impact on the Reduce phase in the case of SCache. The formulas of $T_{Job}, T_{Map}, T_{Reduce}$ and $T_{Shuffle}$ are Equation 3, Equation 4, Equation 5 and Equation 6, respectively. In the case of SCache, Terasort on Hadoop Mapreduce satisfies the situation in Figure 11a($V_{Map} \times R \geq V_{Shuffle}$), thus the formula of $T_{P\_Shuffle}$ is Equation 8. In the case of Legacy, since pre-fetching is not used, $T_{P\_Shuffle}$ is equal to $T_{Shuffle}$(see Equation 6). $ExpT_{Job}$ represents the actual experiment data, we calculate $Error$ according to T_{Job} and $ExpT_{Job}$. The formular of $Error$ is:

$$Error = \frac{ExpT_{Job} - T_{Job}}{T_{Job}} \tag{10}$$

Table 2 shows the calculational results of FRQ model in Amazon EC2 environment. $V_{Map}, V_{Shuffle}$, and V_{Reduce} are modified because of the different hardware devices. We also set K to the same empirical value. The formulas in the table are all the same except $T_{P\_Shuffle}$. In this environment, Terasort on Hadoop Mapreduce satisfies the situation in Figure 11b($V_{Map} \times R < V_{Shuffle}$), thus the formula of $T_{P\_Shuffle}$ is Equation 9. In the case of Legacy, $T_{P\_Shuffle}$ is still euqal to $T_{Shuffle}$.

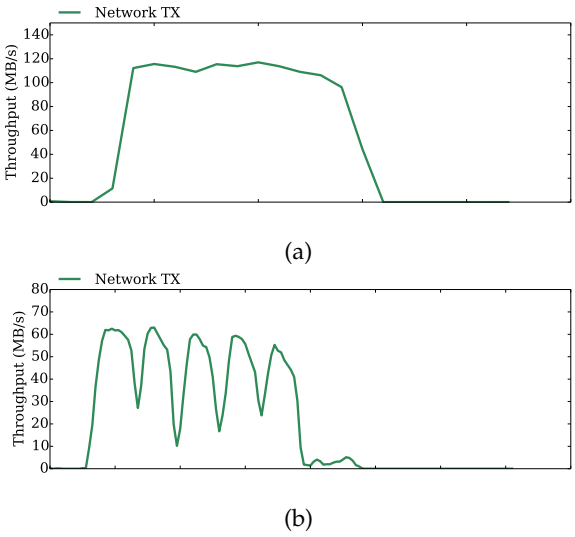


Fig. 12: FRQ Model With SCache

In order to verify the above-mentioned two cases when using SCache, we monitor network utilization and plot it in Figure 12. Figure 12a shows the utilization of Terasort in the lab environment. The network utilization remains high until shuffle phase is complete. This situation is consistent with Figure 11a. Figure 12b shows the utilization in Amazon EC2 environment. The network utilization is 5 regular peaks, this situation is also consistent with Figure 11b. Therefore, we believe that FRQ model is able to accurately describe framework with SCache.

In terms of accuracy, the experimental values are all greater than the calculated values. This is because the application has some extra overhead at runtime, such as network warm-up, the overhead of allocating slots, and so on. In the case where the input data is small and the total time is short, the error caused by the overhead is amplified. Overall, the error between T_{Job} and $ExpT_{Job}$ is basically below 10%, such errors are within tolerance . Therefore, We believe that FRQ model can accurately describe DAG framework.

6 Evaluation

This section reveals the evaluation of SCache with comprehensive workloads and benchmarks. In Spark with SCache, first we run a simple DAG job with single shuffle to analyze hardware utilization and impact of shuffle optimization from the scope of a task to a job. Then we use a recognized shuffle intensive benchmark — Terasort [14] to evaluate SCache with different data partition schemes. To verify the universality of Scache, we also implemented SCache on Hadoop MapReduce. We use Terasort to evaluate the performance of SCache on Hadoop MapReduce.

In order to prove the performance gain of SCache with a real production workload, we also evaluate Spark TPC-

TABLE 1: Hadoop Mapreduce on 4 nodes cluster in FRQ model

D: GB, V_i: GB/s, T_i: s

	D	R	N	V_{Map}	V_{Reduce}	$V_{Shuffle}$	K	T_{Map}	$T_{Shuffle}$	$T_{P\_Shuffle}$	T_{Reduce}	T_{Job}	$ExpT_{Job}$	$Error$
SCache	16	1	2	0.65	1	0.47	0.5	24.62	34.04	21.73	26.87	51.48	55	6.39%
	32	1	4	0.65	1	0.47	0.5	49.23	68.09	31.16	47.58	96.81	104	6.91%
	48	1	6	0.65	1	0.47	0.5	73.85	102.13	40.59	68.29	142.14	151	5.87%
	64	1	8	0.65	1	0.47	0.5	98.46	136.17	50.02	89.01	187.47	193	2.87%
Legacy	16	1	2	0.65	1	0.47	0.6	24.62	34.04	34.04	36.43	61.04	73	16.38%
	32	1	4	0.65	1	0.47	0.6	49.23	68.09	68.09	72.85	122.08	135	9.57%
	48	1	6	0.65	1	0.47	0.6	73.85	102.13	102.13	109.28	183.12	188	2.59%
	64	1	8	0.65	1	0.47	0.6	98.46	136.17	136.17	145.70	244.16	249	1.94%

TABLE 2: Hadoop Mapreduce on 50 AWS m4.xlarge nodes cluster in FRQ model

D: GB, V_i: GB/s, T_i: s

	D	R	N	V_{Map}	V_{Reduce}	$V_{Shuffle}$	K	T_{Map}	$T_{Shuffle}$	$T_{P\_Shuffle}$	T_{Reduce}	T_{Job}	$ExpT_{Job}$	$Error$
SCache	128	1	5	1.15	1.46	1.4	0.5	111.30	91.43	18.29	96.81	208.12	232	10.29%
	256	1	5	1.15	1.46	1.4	0.5	222.61	182.86	36.57	193.63	416.24	432	3.65%
	384	1	5	1.15	1.46	1.4	0.5	333.91	274.29	54.86	290.44	624.36	685	8.85%
Legacy	128	1	5	1.15	1.46	1.4	0.6	111.30	91.43	91.43	142.53	253.83	266	4.57%
	256	1	5	1.15	1.46	1.4	0.6	222.61	182.86	182.86	285.06	507.67	524	3.12%
	384	1	5	1.15	1.46	1.4	0.6	333.91	274.29	274.29	427.59	761.50	776	1.87%

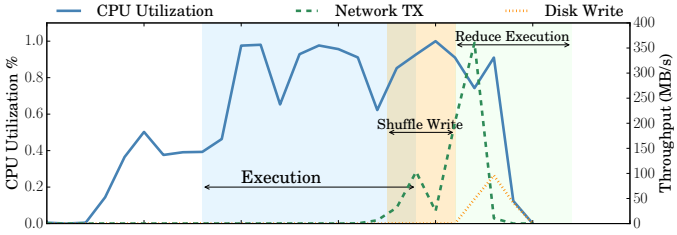


Fig. 13: CPU utilization and I/O throughput of a node during a Spark single shuffle application With SCache

DS [1] and present the overall performance improvement. At last, we measure the overhead of weighted reservoir sampling. In terms of universality, we implemented SCache on Hadoop MapReduce and Spark respectively. It can be proved that the optimization scheme provided by SCache can adapt to the current mainstream distributed DAG framework without a lot of change cost. In short, SCache can decrease 89% time of Spark shuffle. Furthermore, SCache achieves a 75% and 50% improvement of reduce stage completion time respectively in simple DAG application and Terasort without introducing extra network traffic. More impressively, the overall completion time of TPC-DS can be improved 40% on average by applying optimization from SCache. Meanwhile, Hadoop Mapreduce with Scache optimize job completion time by up to 25% and an average of 15%

6.1 Setup

We implement a Spark demon to enable shuffle optimization as a representative. We run our experiments on a 50 m4.xlarge nodes cluster on Amazon EC2 [2]. 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 2).

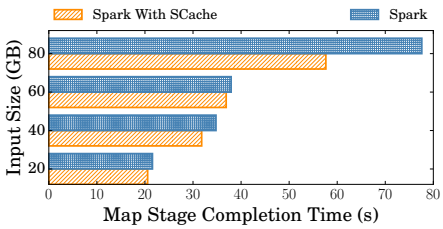
1. https://github.com/databricks/spark-sql-perf
2. http://aws.amazon.com/ec2/

6.2 Simple DAG Analysis

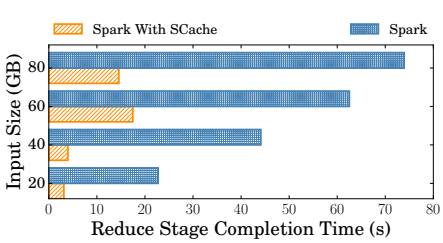
We first run the same single shuffle test as mentioned in Figure 2. As shown in Figure 13, the hardware utilization is captured from one node during the job. Note that since the completion time of whole job is about 50% less than Spark without SCache, the duration of Figure 13 is cut in half as well. An overlap among CPU, disk, and network can be easily observed in Figure 13. That is, SCache prevents the computing process from being cutting off by the I/O operations with a fine-grained resource allocation. It ensures the overall CPU utilization of the cluster stays in a high level. In addition, the decoupling of shuffle write helps free the CPU earlier and leads to a faster map task computation. At the same time, the shuffle pre-fetching in the early stage of map phase shifts the network transfer completion time, which decreases reduce stage completion time. The combination is the main performance gain we achieved on the scope of hardware utilization by SCache.

The performance evaluation in Figure 15 shows the consistent results with our analysis on hardware utilization. For each stage, we run 5 rounds of tasks with different input size. By running spark with SCache, the completion time of map stage can be reduced 10% on average. For reduce stage, instead, SCache achieves a 75% performance gain in the completion time of the reduce stage.

A detail analysis into the nutshell of varied overall performance gain on different stages is presented with Figure 15. For each stage, we pick the median task. About 40% of shuffle write time can be eliminated by SCache (Figure 15a) in a map task. Because the serialization of data is CPU intensive [17] and it is inevitable while moving data out of JVM, SCache can not eliminate the whole phase of shuffle write. It results in a less performance gain in the map stage. On the other hand, the network transfer introduces

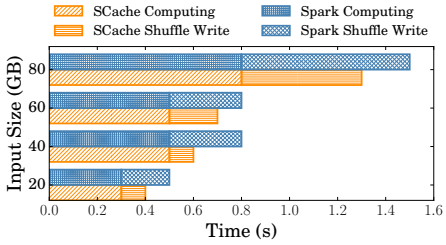


(a) Map Stage Completion Time

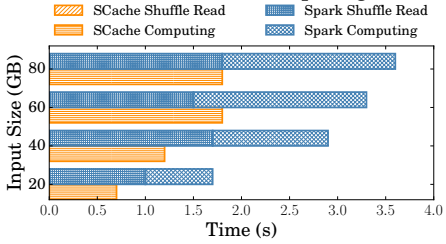


(b) Reduce Stage Completion Time

Fig. 14: Stage Completion Time of Single Shuffle Test

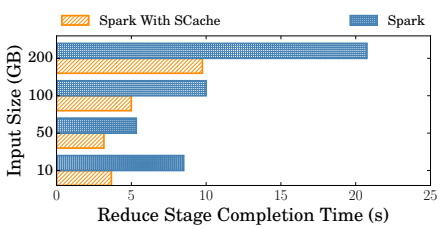


(a) Median Task in Map Stages

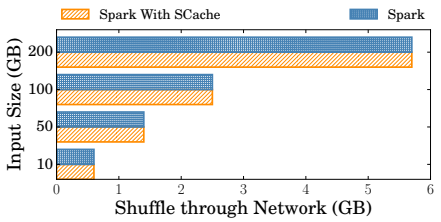


(b) Median Task in Reduce Stages

Fig. 15: Median Task Completion Time of Single Shuffle Test



(a) Reduce Stage of First Shuffle



(b) Network Traffic of Second Shuffle

Fig. 16: Terasort Evaluation

a significantly latency in shuffle read of reduce task (Figure 15b). By doing shuffle data pre-fetching for the reduce tasks, the shuffle read time decreases 100%. That is, the explicit network transfer is perfectly overlapped by SCache shuffle pre-fetching. In overall, SCache can help Spark decrease by 89% time in the whole shuffle process. 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 the reduce stage.

6.3 Terasort

We also evaluate Terasort [14] — a recognized shuffle intensive benchmark for distributed system analysis. Terasort consists of two consecutive shuffles. The first shuffle reads the input data and uses a customized hash partition function for re-partitioning. The second shuffle partitions the data through a range partitioner. As the range bounds set by range partitioner almost match the same pattern of the first shuffle, almost 93% of input data is from one particular map task for each reduce task. So we take the second shuffle as an extreme case to evaluate the scheduling locality for SCache.

As shown in Figure 16a, 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 16b). 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 16b reveals that SCache pre-scheduling produces exactly same network traffic of the 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 optimum.

6.4 Hadoop

To prove SCache compatibility as a plug-in, we also implemented Scache on Hadoop MapReduce as the external shuffle service and co-scheduler. Although *pre-scheduling reduce tasks* is not critical for the simple DAG computing such as Hadoop MapReduce, some shuffle-heavy job on Hadoop can still be optimized.

Figure 20 shows the hardware resource utilization of Hadoop Mapreduce running Terasort. Both figures have the same proportion of time. Hadoop MapReduce with SCache brings 15% of total time optimization with input data size 384GB. Without SCache, Hadoop writes intermediate data locally in the Map phase. Hadoop began to shuffle in the reduce stage. Because the large amount of shuffle data reaches the network bottleneck, Hadoop needs to wait for network transfer in the front of reduce stage. This causes the CPU resources to be idle. As shown in the Figure 20a, Hadoop Madpreduce with SCache start pre-fetching and pre-scheduling in the Map phase. This avoids the Reduce phase waiting for the shuffle data. Furthermore, pre-fetching utilize the idle IO throughput in the Map phase. As shown in Figure 19, after better fine-grained utilization of hardware resources, Hadoop Mapreduce with Scache optimize Terasort completion time by up to 15% and an average of 13% with input data sizes from 128GB to 512GB.

6.5 Production Workload

We also evaluate some shuffle heavy queries from TPC-DS [15]. 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 the performance of Spark with SCache by picking some of the TPC-DS queries with shuffle intensive attribute. As shown in Figure 17, the horizontal axis is query number, and the vertical axis is query completion time. Spark with SCache outperforms

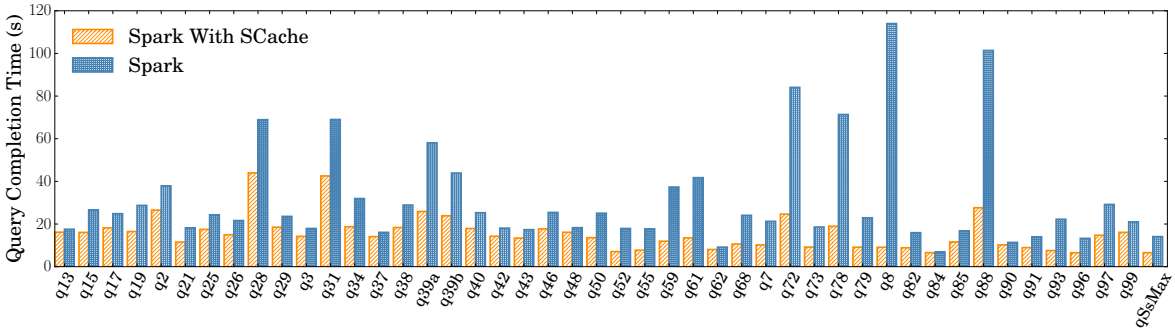


Fig. 17: TPC-DS Benchmark Evaluation

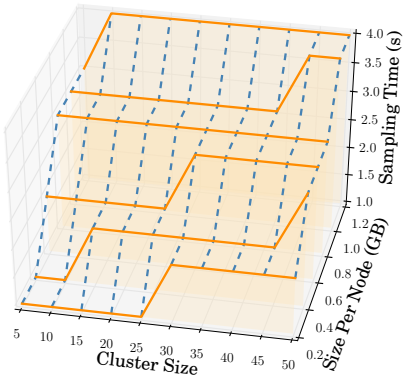


Fig. 18: Sampling Overhead

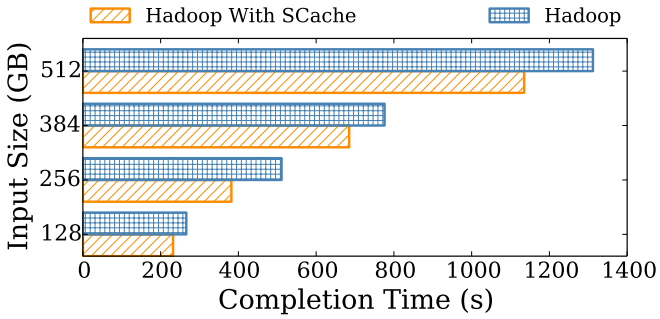
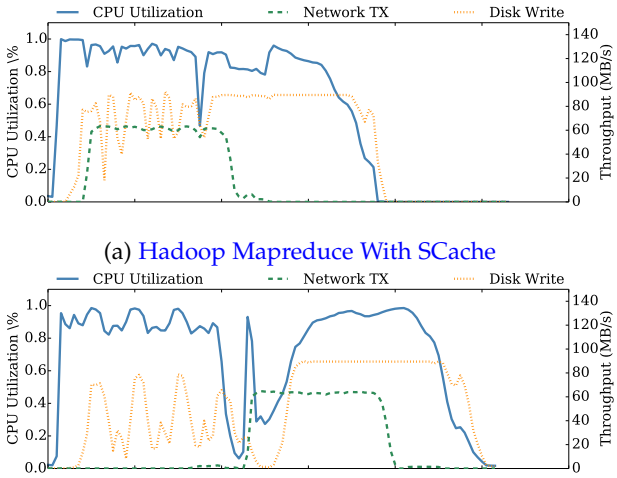


Fig. 19: Hadoop Mapreduce Terasort completion time



(a) Hadoop Mapreduce With SCache

(b) Hadoop Mapreduce Without SCache

Fig. 20: CPU utilization and I/O throughput of a node during a Hadoop Mapreduce Terasort job

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 completion time of queries, we believe that our shuffle optimization is promising.

6.6 Overhead of Sampling

In this part, we evaluate the overhead of sampling with different input data sizes on one node and cluster scales. In Figure 18, the overhead of sampling only grows with the increase of input size on each node. But it remains relatively stable when the cluster size scales up. It makes SCache a scalable system in cluster.

7 Related Work

This section describes related work of shuffle optimization.

Pre-scheduling: Slow-start from Apache Hadoop MapReduce [11] is the most classic approach to handle shuffle overhead. Starfish [33] gets sampled data statics for self-tuning system parameters (e.g. slow-start, map and reduce slot ratio, etc). DynMR [34] dynamically starts reduce tasks in late map stage to decrease the time of waiting for outputs of map tasks. All of them left the explicit I/O time in an occupied computation slot. Instead, SCache can start shuffle pre-fetching without consuming slots. iShuffle [25] decouples shuffle from reducers and designs a centralized shuffle controller. But it can neither handle multiple shuffles nor schedule multiple rounds of reduce tasks. iHadoop [35] 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 multiple shuffles without breaking load balancing by combining DAG information and heuristic algorithms.

Delay-scheduling: Delay Scheduling [10] delays tasks assignment to get better data locality, which can reduce the network traffic. ShuffleWatcher [16] delays shuffle fetching when network is saturated. At the same time, it achieves better data locality. Both Quincy [9] and Fair Scheduling [36] can reduce shuffle data by optimizing data locality of

map tasks. But all of them can not mitigate explicit I/O in both map and reduce tasks. In addition their optimizations fluctuate under different network performances and data distributions, whereas SCache can provide a stable performance gain by shuffle in-memory caching and pre-fetching.

Network layer optimization: Varys [37] and Aalo [38] provide the network layer optimization for shuffle transfer. Though the efforts are limited throughout whole shuffle process, they can be easily applied on SCache to further improve the performance.

8 Conclusion

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

References

[1] 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*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[2] 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*. USENIX Association, 2012, pp. 2–2.

[3] 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*. ACM, 2015, pp. 1357–1369.

[4] 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*, vol. 41, no. 4. ACM, 2011, pp. 98–109.

[5] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 194–204. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688508

[6] S. Babu, "Towards automatic optimization of mapreduce programs," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 137–142. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807150

[7] 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*. ACM, 2014, pp. 1–15.

[8] 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*. USENIX Association, 2012, pp. 20–20.

[9] 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*. ACM, 2009, pp. 261–276.

[10] 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*. ACM, 2010, pp. 265–278.

[11] Apache. (2017) Apache hadoop. [Online]. Available: http://hadoop.apache.com/

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[14] E. Higgs, A. Trivedi, and J. Zhang. (2017) Spark terasort. [Online]. Available: https://github.com/ehiggs/spark-terasort

[15] TPC. (2017) Tpc benchmark ds (tpc-ds): The benchmark standard for decision support solutions including big data. [Online]. Available: http://www.tpc.org/tpcds/

[16] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shuffleswatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters." in *USENIX Annual Technical Conference*, 2014, pp. 1–12.

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

[18] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1012889.1012894

[19] 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*. ACM, 2011, pp. 29–41.

[20] Apache. (2017) Apache hadoop tutorial. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html/

[21] ——. (2017) Apache spark 1.6.2 configuration. [Online]. Available: http://spark.apache.org/docs/1.6.2/configuration.html

[22] 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*. ACM, 2012, pp. 25–36.

[23] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in mapreduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[24] 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*. IEEE, 2012, pp. 522–533.

[25] 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*. ACM, 2014, pp. 97–108.

[26] 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*. Springer, 2011, pp. 165–186.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[28] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[29] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[30] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, 2010, p. 9.

[32] R. Christodoulopoulou, K. Manassiev, A. Bilas, and C. Amza, "Fast and transparent recovery for continuous availability of cluster-based servers," in *Proceedings of the Eleventh ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 221–229. [Online]. Available: http://doi.acm.org/10.1145/1122971.1123005

[33] 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*, vol. 11, no. 2011, 2011, pp. 261–272.

[34] 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*. ACM, 2014, p. 2.

[35] 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*. IEEE, 2011, pp. 81–90.

[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*, 2013, pp. 279–289.

[37] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 443–454.

[38] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 393–406. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787480

PLACE PHOTO HERE

Michael Shell Biography text here.