SCache: Efficient and General Shuffle Management for DAG Computing Frameworks

Rui Ren, Chunghsuan 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 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. During shuffle optimization process, we propose a performance model called *Framework Resources Quantification* (FRQ) model. FRQ model displays utilization of resources in time dimension and calculate execution time of computing jobs on different DAG frameworks. We use FRQ model to assist in analyzing shuffle process and verify SCache shuffle optimization by mathematics. By extracting and analyzing shuffle dependencies prior to the actual task execution, SCache can adopt heuristic pre-scheduling combining with shuffle size prediction to pre-fetch shuffle data and balance load on each node. Meanwhile, SCache takes full advantage of the system memory to accelerate the shuffle process. 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 the widespread use of sophisticated frameworks, such as Hadoop MapReduce[1], Dryad [1], Spark [2], and Apache Tez [3]. Most of them define jobs as directed acyclic graphs (DAGs), such as map-reduce pipeline in Hadoop MapReduce, lineage of resilient distributed datasets (RDDs) in Spark, vertices and edges in Dryad and Tez, etc. 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 a 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 continuous 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 the 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 stages (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 stages (i.e., reduce task). Once scheduled, a fixed bundle

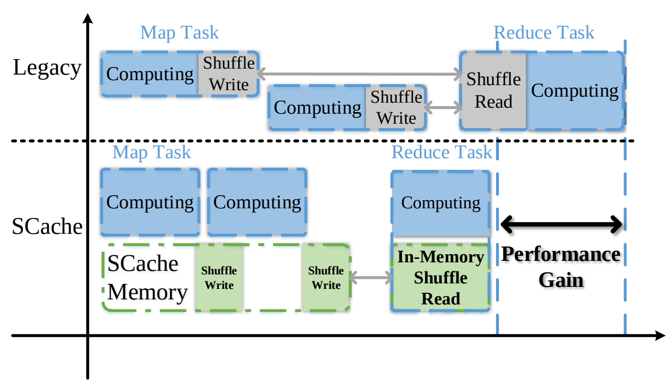


Fig. 1: Workflow Comparison between Legacy DAG Computing Frameworks and Frameworks with SCache

of resources (i.e., CPU, memory, disk and network) named *slot* is assigned to a 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 a cluster has a limited number of slots, 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.

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 the corresponding reduce task starts. Meanwhile, the reduce tasks belonging to a same execution phase are scheduled at

- *The authors are with the School of Software, Shanghai Jiao Tong University, Shanghai 200000, China.*
  Corresponding author: Zhengwei Qi, E-mail: qizhwei@sjtu.edu.cn

1. http://hadoop.apache.com/

the same time by default. As a result, all the corresponding reduce tasks start fetching shuffle 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 ”slow-start” from Hadoop MapReduce[1]. However, such early-start is by no means a panacea. This is because the early-start always introduces an extra early allocation of the slot leading to a slow execution of the current stage.

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 analyze above-mentioned resources scheduling, we propose a performance model called *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 assist in analyze the deficiencies of resources scheduling and optimize it. FRQ model has five parameters: *Input Data Size, Data Conversion Rate, Computation Round Number, Computation Speed,* and *Shuffle Speed*. According to the above five parameters and the scheduling strategy of the DAG framework, FRQ model is able to calculate the execution time of each phase of a computing job. Take Apache Hadoop MapReduce as a simple DAG computing example, we use FRQ to model each phases of the computing, including Map, Shuffle, and Reduce. By revealing the relationships between the various phases, FRQ model assist us in discovering the irrationality of resource scheduling during computing. I/O resource in map phase is not fully utilized. Furthermore, due to coupling with reduce phase, shuffle phase delays the execution time of reduce phase. This situation will be amplified in shuffle-heavy tasks.

Can we efficiently optimize the data shuffling without significantly changing DAG frameworks? In this paper, we answer this question in the affirmative with S(huffle)Cache, an open source[2] 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 write and read 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 *without launching* them and pre-fetches the shuffle data. Such pre-scheduling and pre-fetching effectively overlap the network transfer time, desynchronize the network communication, and avoid the extra early allocation of slots.

The workflow of a DAG framework with SCache is presented in Figure 1. SCache replaces the disk operations of shuffle write by the memory copy in map tasks. The slot is released after the memory copy. The shuffle data is stored in the reserved memory of SCache until all reduce tasks are pre-scheduled. Then the shuffle data is pre-fetched according to the pre-scheduling results. The application-context-aware memory management caches the shuffle data in memory before launching the reduce task. By applying these optimizations, SCache can help the DAG framework achieve a significant performance gain.

The main challenge to achieve this optimization is *pre-scheduling reduce tasks without launching*. First, the complexity of DAG can amplify the defects of naïve scheduling schemes. In particular, randomly assigning reduce tasks might result in a collision of two heavy tasks on one node. This collision can aggravate data skew, thus hurting the performance. Second, pre-scheduling without launching violates the design of most frameworks that launch a task after scheduling. To address the challenges, we propose a heuristic task pre-scheduling scheme with shuffle data prediction and a task co-scheduler (Section 3).

Another challenge is the *in-memory data management*. To prevent shuffle data touching the disk, SCache leverages extra memory to store the shuffle data. To minimize the reserved memory while maximizing the performance gain, we propose two constraints: all-or-nothing and context-aware (Section 4.2).

We have implemented SCache, a customized Apache Spark [11] and a customized Apache Hadoop. We have also designed a performance model called *Framework Resources Quantification* (FRQ) model to analyze the shuffle optimization of SCache and calculate the execution 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[3] benchmark and standard workloads like TPC-DS[4] 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 applications. 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 transfer among nodes. For a clear illustration, we use *map tasks* to define the tasks that produce shuffle data and use *reduce tasks* to define the tasks that consume shuffle data.

1. http://hadoop.apache.com/
2. https://github.com/frankfzw/SCache
3. https://github.com/ehiggs/spark-terasort
4. http://www.tpc.org/tpcds/

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 reduce tasks in the successive step. The shuffle process can be further split into two parts: *shuffle write* and *shuffle read*. Shuffle write starts at the end of a map task and writes the partitioned map output data to local persistent storage. Shuffle read starts at the beginning of a reduce task and fetches the partitioned data from remote as its input.

Impact of shuffle process. Shuffle is I/O intensive, which might introduce a significant latency to the application. Reports show that 60% of MapReduce jobs at Yahoo! and 20% at Facebook are shuffle-heavy workloads [12]. For those shuffle-heavy 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-heavy jobs [4].

2.2 Observations

Can we mitigate or even remove the overhead of shuffle? To find the answers, we ran some typical Spark applications on a 5-node `m4.xlarge` EC2 cluster and analyzed the design and implementation of shuffle in some DAG frameworks. Here we present the hardware utilization trace of one node running Spark's *GroupByTest* in Figure 2 as an example. This job has 2 rounds of tasks for each node. The *Map Execution* is marked from the launch time of the first map task to the execution end time of the last one. The *Shuffle Write* is marked from the beginning of the first shuffle write in the map stage. The *Shuffle Read and Reduce Execution* is marked from the launch time of the first reduce task.

2.2.1 Coarse Granularity Resource Allocation

When a slot is assigned to a task, it will not be released until the task completes (i.e., the end of shuffle write in Figure 2). On the reduce side, the network transfer of shuffle data introduces an explicit I/O delay during shuffle read (i.e., the beginning of shuffle read and execution in Figure 2). Meanwhile, both shuffle write and shuffle read occupy the slot without significantly involving CPU as presented in Figure 2. The current coarse slot-task mapping results in an imbalance between task's resource demand and slot allocation thus decreasing the resource utilization. Unfortunately this defect exists not only in Spark [11] but also Hadoop MapReduce and Apache Tez [3]. A finer granularity resource allocation scheme should be provided to reduce these delays.

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 causes a high demand of network bandwidth, which may result in network congestion and further slow down the network transfer. It also happens in other frameworks that follow Bulk Synchronous Parallel (BSP) paradigm, such as Hadoop MapReduce, Dryad [1], etc.

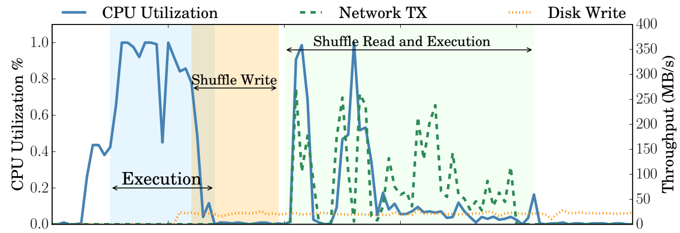


Fig. 2: CPU utilization and I/O throughput of a node during a Spark single shuffle application

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 I/O operation. This blocking I/O operation along with synchronized shuffle read may introduce significant latency, especially in an I/O performance bounded cluster. Besides, the legacy of storing shuffle data on disk is inefficient in modern clusters with large memory. Compared to input dataset, the size of shuffle data is relatively small. For example, the shuffle size of Spark Terasort[1] is less than 25% of input data. The data reported in [13] also show that the amount of data shuffled is less than the input data by as much as 10%-20%. On the other hand, memory based distributed storage systems have been proposed [7], [14], [15] to move data back to memory, but most of the DAG frameworks still store shuffle data on disks (e.g., Spark [11], Hadoop MapReduce, Dryad [1], etc). 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 a cluster with n slots, the number of tasks should be $n \times k (k \geq 1)$. For example, Hadoop MapReduce Tutorial[2] suggests that *10-100 maps* per-node and *0.95 or 1.75* $\times$ *number of nodes* $\times$ *number of maximum container reduces* per-node seem to be the right level of parallelism. Spark configuration also recommends 2-3 tasks per CPU core[3] (i.e., $k =$ 2-3).

Since the shuffle data becomes available as soon as the end of a task's execution, and the network is idle during the map stage ("Network TX" during map stage in Figure 2), the property of multi-round tasks can be leveraged to hide the cost by starting shuffle data transfer at the end rounds of map tasks.

Based on these observations, most of the DAG frameworks share the execution paradigm as well as the expense of shuffle process. To mitigate the shuffle overhead, we propose an optimization that starts shuffle read ahead of reduce stage to overlap the I/O operations in multi-round map tasks, and uses memory to store the shuffle data. To achieve this optimization:

- Shuffle process should be decoupled from task execution to achieve a fine granularity scheduling scheme.

1. https://github.com/ehiggs/spark-terasort
2. http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html
3. http://spark.apache.org/docs/1.6.2/configuration.html

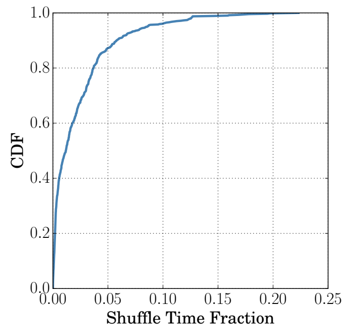


Fig. 3: Shuffle Time Fraction CDF of OpenCloud Trace

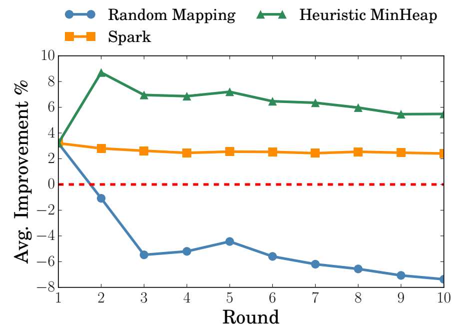


Fig. 4: Stage Completion Time Improvement of OpenCloud Trace

- 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.

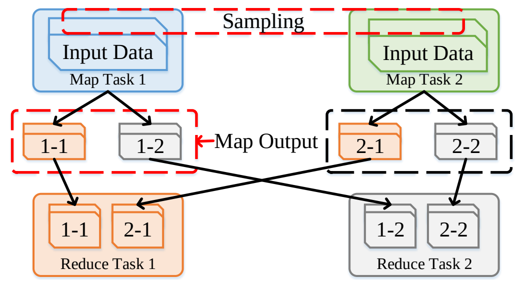


Fig. 5: Shuffle Data Prediction

3 Shuffle Optimization

This section presents the detailed methodologies to achieve the 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) and a co-scheduler is used to achieve shuffle data pre-fetching without launching tasks.

3.1 Decouple Shuffle from Execution

To achieve the decoupling of map tasks and reduce tasks, the original shuffle write and read implementation in the current frameworks should be modified to apply the API of SCache. To prevent the release of a slot being blocked by shuffle write, SCache provides a disk-write-like API named $putBlock$ to handle the storage of partitioned shuffle data blocks produced by a map task. Inside the $putBlock$, SCache uses memory copy to move the shuffle data blocks out of map tasks and store them in the reserved memory. After the memory copy, the slot will be released immediately.

From the perspective of reduce task, SCache provides an API named $getBlock$ to replace the original implementation of shuffle read. With the precondition of shuffle data pre-fetching, the $getBlock$ leverages the memory copy to fetch the shuffle data from the local memory of SCache.

3.2 Pre-schedule with Application Context

The pre-scheduling and pre-fetching are the most critical aspects of 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 awareness of task-node mapping. We propose a co-scheduling scheme with two heuristic algorithms (Algorithm 1, 2). That is, the task-node mapping is established a priori, and then it is enforced by the co-scheduler when the DAG framework starts task scheduling.

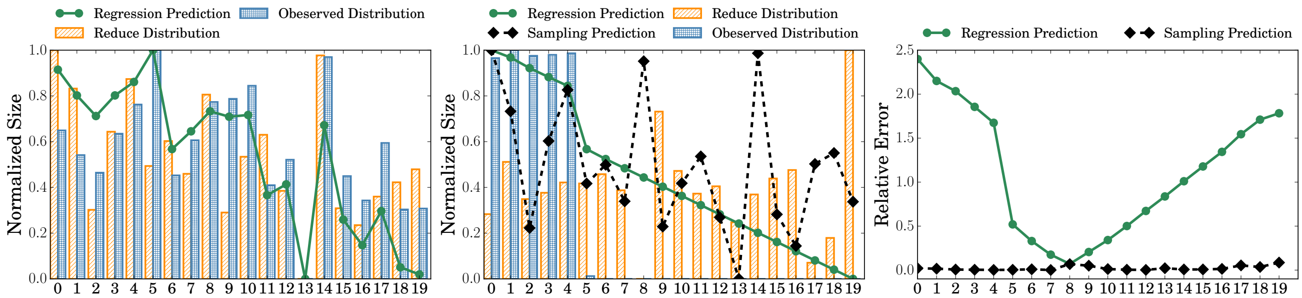
3.2.1 Problem of Random Mapping

The simplest way of pre-scheduling is mapping tasks to nodes randomly and evenly. In order to evaluate the effectiveness of random mapping, we use traces from OpenCloud[1] for the simulation. Note that most of the traces from OpenCloud are shuffle-light workload as shown in Figure 3. The average shuffle read time is 3.2% of total reduce completion time. As shown in Figure 4, the baseline (i.e., red dotted line) is the stage completion time with Spark FIFO scheduling algorithm. We then remove the shuffle read time of each task and run the simulation under three scheduling schemes: random mapping, Spark FIFO, and our heuristic MinHeap. Random mapping works well when there is only one round of tasks, but the performance drops as the round number grows. This is because that data skew commonly exists in data-parallel computing [16], [17], [18]. Several heavy tasks may be assigned to the same node, thus slowing down the whole stage. In addition, randomly assigned tasks also ignore the data locality between shuffle map output and reduce input, which may introduce extra network traffic in cluster.

3.2.2 Shuffle Output Prediction

The problem of random mapping is obviously caused by application context (e.g., shuffle data size) ignorance. Note that a balanced schedule decision can be made under the consideration of the size of each reduce task, and the size of a reduce task produced by one shuffle $reduceSize_i = \sum_{j=0}^{m} BlockSize_{ji}$, where the m is the number of map tasks that can be easily extracted from DAG information; $BlockSize_{ji}$ represents the size of block which is produced by map $task_j$ for reduce $task_i$ (e.g., block '1-1' in Figure 5). The final sizes of reduce tasks can be calculated by

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

aggregating $reduceSize_i$ by reduce ID among all shuffle dependencies. So the pre-scheduling can be made if the "prediction" of size of shuffle block is practical.

For the most DAG applications with random large scale input, the $BlockSize_{ji}$ in a particular shuffle can be predicted with decent accuracy by liner regression model (i.e., equation 1) based on observation that the ratio of map output size and input size are invariant given the same job configuration [19], [20]:

$$BlockSize_{ji} = a \times inputSize_j + b \tag{1}$$

The $inputSize_j$ is the input size of jth map task. The a and b can be determined using the observed $inputSize_j$ and $BlockSize_{ji}$.

Though the linear regression is stable in most scenarios, it can fail in some uncertainties introduced by sophisticated frameworks like Spark [11]. For instance, the customized partitioner may result in large inconsistency between observed map output blocks distribution and the final reduce input distribution. We present two particular examples with 20 tasks respectively in Figure 6a and Figure 6b. The data in are normalized to $0 - 1$ because the prediction of SCache only produces the data distribution instead of the real size. The observed map outputs are randomly picked. With a random input and a hash partitioner in Figure 6a, the distribution of observed map output is close to the final reduce input distribution. The prediction results also fit them well. However, the data partitioned by Spark RangePartitioner [11] in Figure 6b results in a deviation from the linear regression model, because the RangePartitioner might introduce an extreme high data locality skew. That is, for one reduce task, almost all of the input data are produced by a particular map task (e.g., the observed map tasks only produce data for reduce task 0-5 in Figure 6b). The data locality skew results in a missing of other reduce tasks' data in the observed map outputs.

To handle this corner case, we introduce another methodology, named *weighted reservoir sampling*, as a substitution of linear regression. Note that linear regression will be replaced only when a RangePartitioner or a customized non-hash partitioner occurs. For each 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). Finally, the partitioned outputs are collected with the $InputSize_j$ as the weight of the samples. Note that sampling does not consume the input data of map tasks. The $BlockSize_{ji}$ can be calculated by:

$$\begin{gathered} BlockSize_{ji} = InputSize_j \times \frac{sample_i}{s \times p} \\ sample_i = \text{number of samples for } reduce_i \end{gathered} \tag{2}$$

In Figure 6b, when s is set to 3, the result of sampling prediction is much better than linear regression. The variance of the normalization between sampling prediction and reduce distribution is because the standard deviation of the prediction results 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 input size of reduce task. On the other hand, the result of linear regression comes out with a large relative error. Though the weighted reservoir sampling is precise, it also introduced extra overhead. We will show the overhead evaluation of sampling in Section 6.

During both of the predictions, the composition of each reduce partition is calculated as well. We define $prob_i$ as

$$\begin{gathered} prob_i = \max_{0 \le j \le m} \frac{BlockSize_{kji}}{reduceSize_i} \\ m = \text{number of map tasks} \end{gathered} \tag{3}$$

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

3.2.3 Heuristic MinHeap Scheduling

As long as the input sizes of reduce tasks are available, the pre-scheduling is a classic scheduling problem without considering the data locality. But ignoring the data locality can introduce extra network transfer. In order to balance load while minimizing the network traffic, we present the Heuristic MinHeap scheduling algorithm (Algorithm 1).

For the pre-scheduling itself (i.e., the first *while* in Algorithm 1), the algorithm maintains a min-heap to simulate the load of each node and applies the longest processing time rule (LPT) [21] to achieve $4/3$-*approximation* optimum. Since the sizes of tasks are considered while scheduling, Heuristic MinHeap can achieve a shorter makespan than Spark FIFO which is a 2-*approximation* optimum. Simulation of OpenCloud trace in Figure 4 also shows that Heuristic MinHeap has a better improvement (average 5.7%) than the Spark FIFO (average 2.7%). After pre-scheduling, the task-node mapping will be adjusted according to the locality. The $SWAP\_TASKS$ will be triggered when the $host\_id$ of a task does not equal the $assigned\_id$. Based on the $prob$, the normalized probability $norm$ is calculated as a bound of performance degradation. Inside the $SWAP\_TASKS$, tasks will be selected and swapped without exceeding the $upper\_bound$.

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 non-increasing order
  $M \leftarrow$ min-heap $\{host\_id \rightarrow ([reduces], size)\}$
  $idx \leftarrow 0$
  while $idx <$ len$R$ **do**
    $M[0].size \mathrel{+}= R[idx].size$
    $M[0].reduces.append(R[idx])$
    $R[idx].assigned\_id \leftarrow M[0].host\_id$
    Sift down $M[0]$ by $size$
    $idx \leftarrow idx - 1$
  $max \leftarrow$ maximum size in M
  for all $reduce$ in R **do** ▷ Heuristic locality swap
    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$(M, reduce, upper_bound)$
  Swap tasks between node $host\_id$ and node $assigned\_id$
  of $reduce$ without exceeding the $upper\_bound$
  of both nodes.
  Return if it is impossible.

Algorithm 2 Accumulated Heuristic Scheduling for Multi-Shuffles

procedure M_SCHEDULE$(m, host_id, p_reduces, shuffles)$
  $m \leftarrow$ partition number of map tasks ▷ *shuffles* are the previous schedule result
  for all r in $p\_reduces$ **do**
    $r.size \mathrel{+}= shuffles[r.rid].size$
    $new\_prob \leftarrow shuffles[r.rid].size/r.size$
    if $new\_prob \geq r.prob$ **then**
      $r.prob \leftarrow new\_prob$
      $r.host\_id \leftarrow shuffles[r.rid].assigned\_host$
  $M \leftarrow SCHEDULE(m, host\_id, p\_reduces)$
  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

3.2.4 Cope with Multiple Shuffle Dependencies

A reduce stage can have more than one shuffle dependencies in the current DAG computing frameworks. The technique mentioned in Section 3.2.2 can only handle an ongoing shuffle. For those pending shuffles, it is impossible to predict their sizes. This problem can be solved by having all map tasks of pending shuffles launched simultaneously. But doing this introduces large overhead such as extra task serialization. To avoid violating the optimization from framework, we present the Accumulated Heuristic Scheduling algorithm to cope with multiple shuffle dependencies.

As illustrated in Algorithm 2, the sizes of previous *shuffles* scheduled by Heuristic MinHeap are counted. When a new shuffle starts, the predicted $size$, $prob$, and $host\_id$ in $p\_reduces$ are accumulated with previous *shuffles*. After scheduling, if the new $assigned\_id$ of a reduce task did not equal the original one, a re-shuffle will be triggered to transfer data to the new host. This re-shuffle is rare since the previous shuffle data contributes a huge composition (i.e., high $prob$) after the accumulation, which leads to a higher probability of tasks swap in $SWAP\_TASKS$.

4 Implementation

This section presents an overview of the implementation of SCache. We first present the system overview and the detail of sampling in Subsection 4.1. The following 4.2 subsection focuses on the two constraints on memory management. In Subsection 4.3, we will evaluate cross-framework capability of SCache based on Hadoop MapReduce and Spark. At last, we discuss the cost of adapting SCache and the fault tolerance.

4.1 System Overview

SCache consists of three components: a distributed shuffle data management system, a DAG co-scheduler, and a worker daemon. As a plug-in system, SCache needs to rely on a DAG framework. As shown in Figure 7, SCache employs the legacy master-slaves architecture like GFS [22] 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 DAG framework and SCache. The co-scheduler is dedicated to pre-schedule reduce tasks with DAG information and enforce the scheduling results to original scheduler in framework.

When a DAG job is submitted, the DAG information will be generated in framework task scheduler. Before the computing tasks begin, the shuffle dependencies are determined based on DAG. 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 computing job. We will elaborate the sampling procedure in the Section 4.1.1.

When a map task finishes computing, the shuffle write implementation of DAG framework is modified to call the SCache API and move all the blocks out of framework worker through memory copy. After that the slot will be released (without being blocked on disk operations). When a block of the map output (i.e., "map output" in Figure 5) is received, the SCache worker will send the block ID and the size to the master. If the collected map output data reach the observation threshold, the DAG co-scheduler will run the scheduling Algorithm 1 or 2 to pre-schedule the reduce tasks and then broadcast the scheduling result

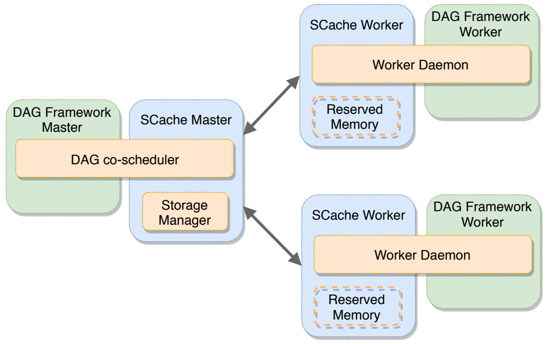


Fig. 7: SCache Architecture

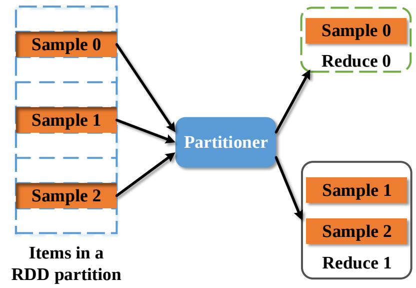


Fig. 8: Reservoir Sampling of One Partition

to start pre-fetching on each worker. SCache worker will filter the reduce tasks' IDs that are scheduled on itself and start pre-fetching shuffle data from the remote. In order to force DAG framework to run according to the SCache pre-scheduled results, we insert some lines of codes in framework scheduler. After modification, DAG scheduler will consult SCache co-scheduler to get the preferred location for each task.

4.1.1 Reservoir Sampling

If the submitted shuffle dependencies contained a RangePartitioner or a customized non-hash partitioner, the SCache master will send a sampling request to the framework master. The sampling job uses a reservoir sampling algorithm [23] on each partition. For the sample number, it can be tuned to balance the overhead and accuracy. 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 task ID on SCache master to predict the reduce partition size. After the prediction, SCache master will call Algorithm 1 or 2 to do the pre-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 limit 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.

4.2.1 All-or-Nothing Constraint

This acceleration of in-memory cache 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 missed a memory cache and exceeded the original bottleneck of this round, that task might become the new bottleneck and then 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 blocks of one round as the minimum unit of storage management. For those incomplete units, SCache marks them as the lowest priority.

4.2.2 Context-Aware-Priority Constraint

Unlike the traditional cache replacement schemes such as MIN [24], the cached shuffle data will only be used once (without failure), but the legacy cache managements are designed to improve the hit rate. SCache leverages application context to select victim storage units when the reserved memory is full.

At first, SCache flushes blocks of the incomplete units to disk cluster-widely. If all the units are completed, SCache selects victims based on two factors-*inter-shuffle* and *intra-shuffle*.

- Inter-shuffle: SCache master follows the scheduling scheme of Spark to determine the inter-shuffle priority. For example, Spark FIFO scheduler schedules the tasks of different stages according to the submission order. So SCache sets the priorities according to the submission time of each shuffle.
- Intra-shuffle: The intra-shuffle priorities are determined according to the task scheduling inside a stage. For example Spark schedules tasks with smaller ID at first. Based on this, SCache can assign the lower priority to storage units with a larger task ID.

4.3 Analysis of cross-framework capability

Shuffle optimization of SCache inevitably requires modification on dependent DAG framework. SCache provides API through RPC, such as *putBlock (blockId)*, *getBlock (blockId)*, and *getScheduleResult (shuffleId)*. In order to use SCache, we mainly need to modify the two parts of the framework: (a)We need to insert codes in the DAG scheduler to let framework provide DAG information and follow pre-scheduled result of SCache. (b)The shuffle data between the Map task and the Reduce task should be transfered to SCahce Storge Management.

To prove the cross-framework capability of SCache, we adapt SCache on Hadoop MapReduce and Spark respectively. Taking Hadoop MapReduce as an example, we insert

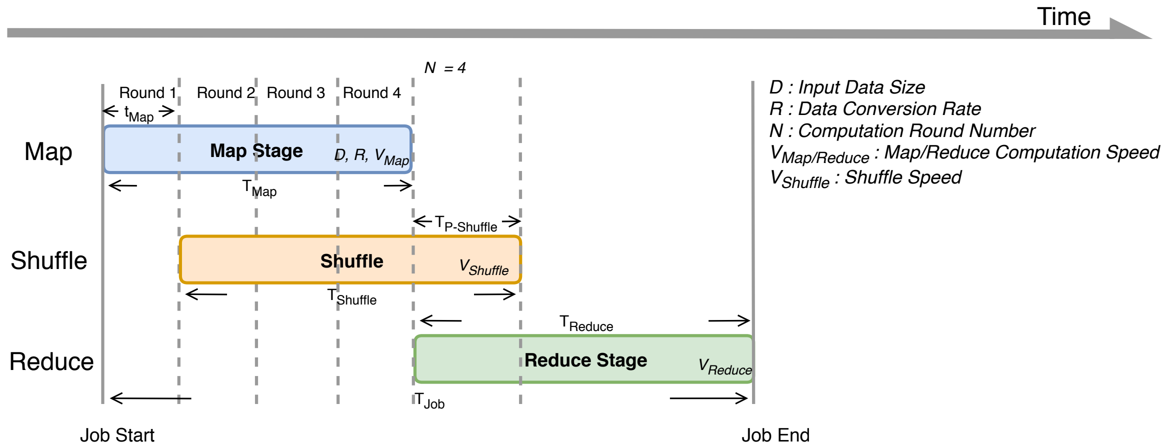


Fig. 9: Framework Resources Quantification (FRQ) Model with Full Parallel MapReduce

codes in ResourceManager and modify codes in MapTask and ReduceTask to call SCache API through RPC. It takes about 400 lines of code. In Spark, we mainly modified the DAGScheduler and the corresponding data fetcher. It only takes about 500 lines of code. Such hundreds of lines of code modification are very small compared to the hundreds of thousands of lines of code in DAG framework. we believe that the costs of enabling SCache on other DAG computing frameworks, such as Tez [3], are also very low.

4.4 Fault tolerance

Due to the characteristic of shuffle data (e.g., short-lived, write-once, read-once), we believe fault tolerance is not a crucial goal of SCache at present. We plan to implement SCache master with Apache ZooKeeper [25] to provide constantly service. If a failure happened inside the SCache worker, SCache daemon could block the shuffle write/read operations until the worker process restarted without violating the correctness of DAG computing. A possible way to handle this failure is selecting some backup nodes to store replications. But the replications can introduce a significant network overhead [26]. Currently, we left the sever faults (e.g., the failure of a node) to the DAG frameworks. We believe it is a more promising way because most DAG frameworks have more advanced fast recovery schemes on the application layer, such as paralleled recovery of Spark. Meanwhile, SCache can still provide shuffle optimization during the recovery.

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 data parallel computing frameworks mostly use *Directed acyclic graphs* (DAGs) 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 job 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. As shown in Figure 9, in full parallel mapReduce, the total execution time of a job is the sum of the map phase time and reduce phase time:

$$T_{Job} = T_{Map} + T_{Reduce} \tag{4}$$

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

$$T_{Map} = \frac{D}{V_{Map}} \tag{5}$$

The reduce phase time formula is as follows:

$$T_{Reduce} = \frac{D \times R}{V_{Reduce}} + K \times T_{P\_Shuffle} \tag{6}$$

We use $T_{P\_Shuffle}$ to represent the overlap time between shuffle phase and reduce phase. $T_{Shuffle}$ represents the total time of shuffle phase. The relationship between $T_{P\_Shuffle}$ and $T_{Shuffle}$ is determined by the resources scheduling strategy of DAG frameworks. This relationship will be shown in the following Subsection 5.2. $\frac{D \times R}{V_{Reduce}}$ represents the ideal computation time of a reduce phase, and $K \times T_{P\_Shuffle}$ represents the computing overhead. K is an empirical value. Because the computation 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.

$$T_{Shuffle} = \frac{D}{V_{Shuffle}} \tag{7}$$

For shuffle-heavy computing jobs, we can optimize the job completion time by reducing $T_{P\_Shuffle}$. Improving I/O speed is a effective way to reduce shuffle time. Another optimization method is to use the idle I/O 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.

5.2 Model Analysis

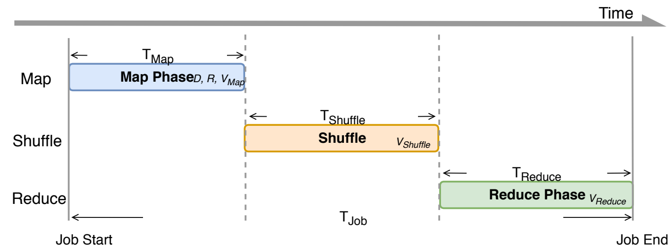
The FRQ model can describe a variety of resource scheduling strategies. First, we analyze a naive scheduling strategy. As shown in Figure 10a, FRQ model describes a MapReduce job that is fully serially executed. The overlap time between shuffle phase and the reduce phase is 0, in which case $T_{P\_Shuffle}$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

$$T_{Job} = T_{Map} + T_{Shuffle} + T_{Reduce} \tag{8}$$

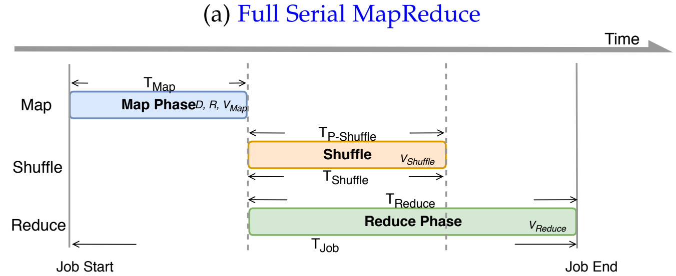
Apache Spark uses this scheduling strategy by default. Due to serialization, the I/O resource is idle during the reduce phase and map phase. The scheduling strategy is simple and has a lot of room for optimization.

Figure 10b shows one of the optimization methods. In this scheduling strategy, 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 (according to equation 6). 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 4). The execution time of shuffle phase is hidden in the reduce phase. This scheduling strategy is used by Hadoop MapReduce. After analyzing this model, we believe that this strategy will be more efficient than the above serial strategy in the same situation. Meanwhile, we found that the I/O resource in the map phase is still idle. This scheduling strategy can be optimized.

Hadoop MapReduce overlaps shuffle phase with reduce

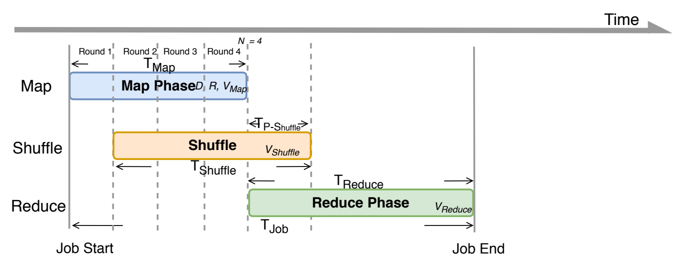


(a) Full Serial MapReduce

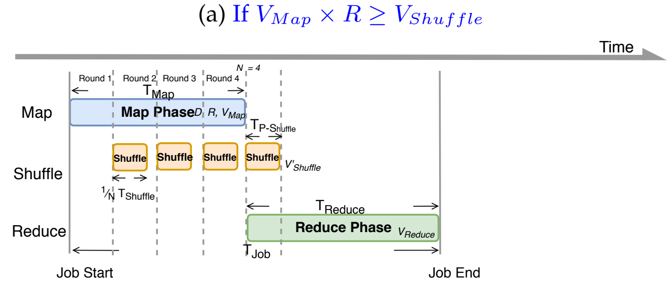


(b) Half Parallel MapReduce

Fig. 10: Framework Resources Quantification (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: Framework Resources Quantification (FRQ) Model with Full Parallel MapReduce in Different Environments

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%

phase and utilizes idle resources. We intuitively think that we can also overlap shuffle phase one and map phases. We implemented this idea with SCache. 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 the I/O resource 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:

1) $V_{Map} \times R \geq V_{Shuffle}$ (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:

$$T_{P\_Shuffle} = T_{Shuffle} - \frac{(N-1) \times T_{Map}}{N} \tag{9}$$

2) $V_{Map} \times R < V_{Shuffle}$ (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 $T_{P_{Shuffle}}$ is equal to the total time of shuffle ($T_{Shuffle}$) minus the time that shuffle is executed in the map phase:

$$T_{P\_Shuffle} = T_{Shuffle} \times \frac{1}{N} \tag{10}$$

Compared to the original Hadoop MapReduce resource scheduling strategy, Hadoop MapReduce with SCache reduces $T_{P\_Shuffle}$ and thus lessens T_{Reduce}. This is how pre-fetching optimizes the total execution time of a 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 framework and Terasort as experimental application. We deployed 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 4, Equation 5, Equation 6 and Equation 7, 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 9. In the case of Legacy, since pre-fetching is not used, $T_{P\_Shuffle}$ is equal to $T_{Shuffle}$(see Equation 7). $ExpT_{Job}$ represents the actual experiment data, we calculate $Error$ according to T_{Job} and $ExpT_{Job}$. The formula of $Error$ is:

$$Error = \frac{ExpT_{Job} - T_{Job}}{T_{Job}} \tag{11}$$

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 10. In the previous case, $T_{P\_Shuffle}$ is still euqal to $T_{Shuffle}$.

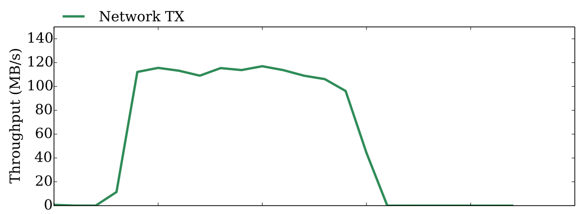
In order to verify the above-mentioned two cases when using SCache, we monitor network utilization and plot it in Figure 12. Figure 12a shows the utilization of Terasort in the lab environment. The network utilization remains high until shuffle phase is complete. This situation is consistent with Figure 11a. Figure 12b shows the utilization in Amazon EC2 environment. The network utilization has 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

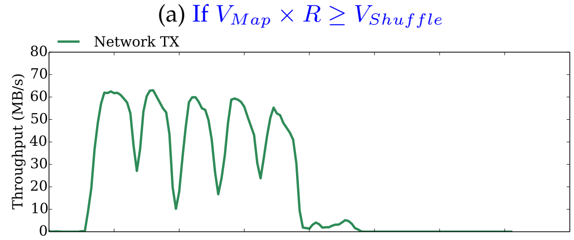
TABLE 2: Hadoop MapReduce on 50 AWS m4.xlarge nodes cluster in FRQ model

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%



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



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

Fig. 12: Network utilization on Hadoop MapReduce with SCache

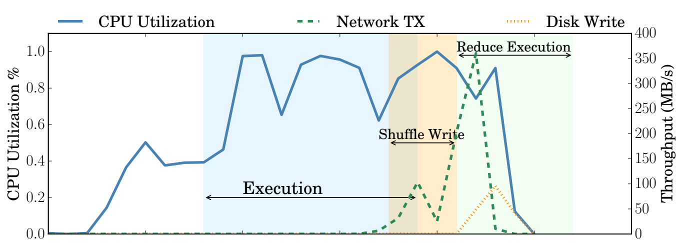


Fig. 13: CPU utilization and I/O throughput of a node during a Spark single shuffle application With SCache

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. First we run a job with single shuffle to analyze hardware utilization and see the impacts of different components from the scope of a task to a job. Then we use a recognized shuffle intensive benchmark — Terasort to evaluate SCache with different data partition schemes.

In order to prove the performance gain of SCache with a real production workload, we also evaluate Spark TPC-DS[1] and present the overall performance improvement. To prove SCache compatibility as a cross-framework plug-in, we implemented SCache on both Hadoop MapReduce and Spark. Due to the simple DAG computing in Hadoop MapReduce, we only use Terasort as a shuffle-heavy benchmark to evaluate the performance of Hadoop MapReduce with SCache. Finally, we measure the overhead of weighted reservoir sampling. In summary, SCache can decrease 89% time of Spark shuffle without introducing extra network transfer. More impressively, the overall completion time of TPC-DS can be improved 40% on average by applying the optimization from SCache. Meanwhile, Hadoop MapReduce with SCache optimize job completion time by up to 15% and an average of 13%

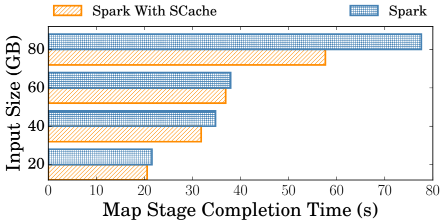
6.1 Setup

We modified Spark to enable shuffle optimization of SCache as a representative. The shuffle configuration of Spark is set to the default[2]. We run the experiments on a 50-node m4.xlarge cluster on Amazon EC2[3]. Each node has 16GB memory and 4 CPUs. The network bandwidth provided by Amazon is insufficient. Our evaluations reveal the bandwidth is only about 300 Mbps (see Figure 2).

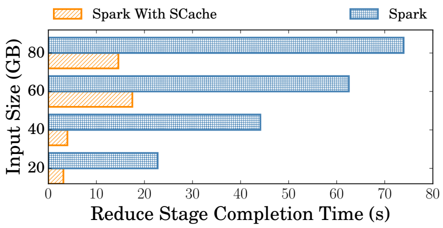
6.2 Simple DAG Analysis

We first run the same single shuffle test shown in Figure 2. For each stage, we run 5 rounds of tasks with different input size. 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. It is because the decoupling of shuffle prevents the computing resource from being blocked by I/O operations. On the one hand, the decoupling of shuffle write helps free the slot earlier, so that it can be re-scheduled to a new map task. On the other hand, with the help of shuffle pre-fetching, the decoupling of shuffle read significantly decreases the CPU idle time at the beginning of a reduce task. At the same time, SCache manages the hardware resources to store and transfer shuffle data without interrupting the computing process. As a result, the utilization and multiplexing of hardware resource are increased, thus improving the performance of Spark.

1. https://github.com/databricks/spark-sql-perf
2. http://spark.apache.org/docs/1.6.2/configuration.html
3. http://aws.amazon.com/ec2/

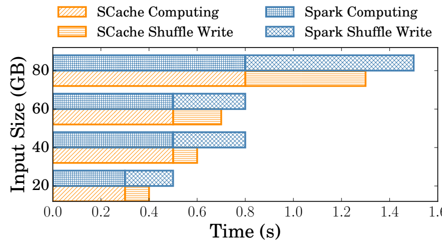


(a) Map Stage Completion Time

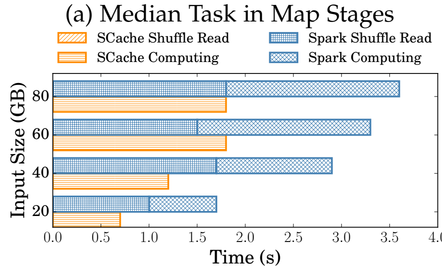


(b) Reduce Stage Completion Time

Fig. 14: Stage Completion Time of Single Shuffle Test

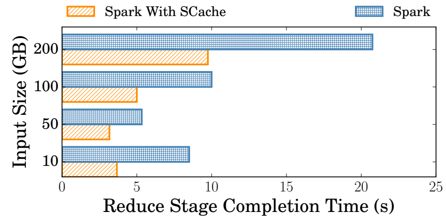


(a) Median Task in Map Stages

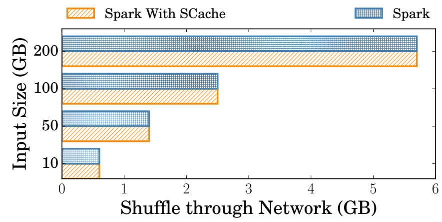


(b) Median Task in Reduce Stages

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

The performance evaluation in Figure 15 shows the consistent results with our observation of hardware utilization. For each stage, we pick the task that has median completion time. In the map task, the disk operations are replaced by the memory copies to decouple the shuffle write. It helps eliminate 40% of shuffle write time (Figure 15a), which leads to a 10% improvement of map stage completion time in Figure 14a. Note that the shuffle write time can be observed even with the optimization of SCache. The reason is that before moving data out of Spark's JVM, the serialization is inevitable and CPU intensive [13].

In the reduce task, most of the shuffle overhead is introduced by network transfer delay. By doing shuffle data pre-fetching based on the pre-scheduling results, the explicit network transfer is perfectly overlapped in the map stage. With the help of the co-scheduling scheme, SCache guarantees that each reduce task has the benefit of shuffle pre-fetching. The in-memory cache of shuffle data further reduce the shuffle read time. As a result, the combination of these optimizations decreases 100% overhead of the shuffle read in a reduce task (Figure 15b). In addition, the heuristic algorithm can achieve a balanced pre-scheduling result, thus providing 80% improvement in reduce stage completion time (Figure 14b).

In overall, SCache can help Spark decrease by 89% overhead of the whole shuffle process.

6.3 Terasort

We also evaluate Terasort[1]-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 hash partition function for re-partitioning. As shown in Figure 16a, Spark with SCache runs 2 × faster during the reduce stage of the first shuffle, which is consistent with the results in Section 6.2. It further proves the effectiveness of SCache's optimization.

1. https://github.com/ehiggs/spark-terasort

The second shuffle of Terasort partitions the data through a Spark RangePartitioner. In the second shuffle, almost 93% of input data of a reduce task is produced by one particular map task. So we take the second shuffle as an extreme case to evaluate the heuristic locality swap of SCache. In this shuffle, Spark schedules a reduce task to the node that produces most input data. By doing this, Spark minimizes the shuffle data through network. At the same time, Figure 16b reveals that SCache produces exactly same network traffic as Spark. It implies that the heuristic locality swap of SCache can obtain the best locality while balancing the load.

6.4 Hadoop MapReduce with SCache

To prove SCache compatibility as a cross-framework 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 in Hadoop MapReduce, some shuffle-heavy jobs on Hadoop Madpreduce can still be optimized by SCache shuffle data management.

Figure 19 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 384GB input data size. As shown in the Figure 19b, Hadoop MapReduce without SCache writes intermediate data locally in the map phase. The shuffle phase and reduce phase start simultaneously. Because the large amount of shuffle data reaches the network bottleneck, the beginning part of reduce needs to wait needs to wait for network transfer. This causes the CPU resources to be idle. As shown in the Figure 19a, Hadoop Madpreduce with SCache start pre-fetching in the map phase. This avoids the reduce phase waiting for the shuffle data. Furthermore, pre-fetching utilizes the idle IO throughput in the map phase. As shown in Figure 20, after better fine-grained utilization of hardware resources,

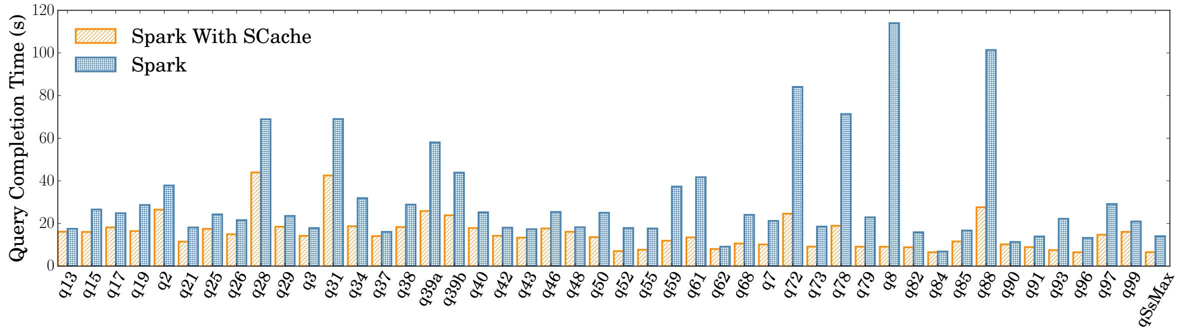


Fig. 17: TPC-DS Benchmark Evaluation

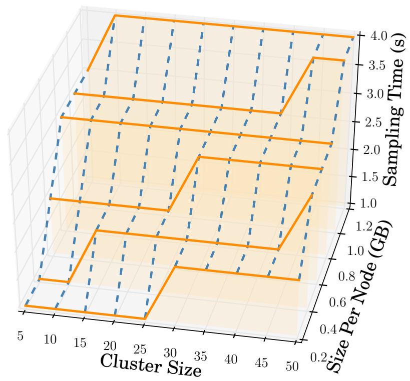


Fig. 18: Sampling Overhead

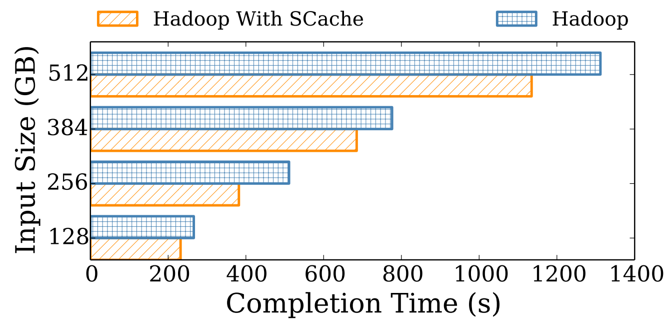
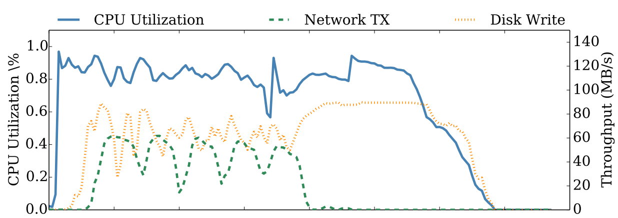
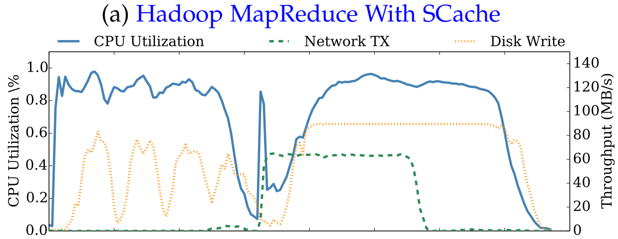


Fig. 20: Hadoop MapReduce Terasort completion time



(a) Hadoop MapReduce With SCache



(b) Hadoop MapReduce Without SCache

Fig. 19: CPU utilization and I/O throughput of a node during a Hadoop MapReduce Terasort job

Hadoop MapReduce with SCache optimize Terasort overall 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 queries from TPC-DS[1]. 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. As shown in Figure 17, the horizontal axis is query name and the vertical axis is query completion time. Note that we skip some queries due to the compatible issues. Spark with SCache outperforms the original Spark in almost all tested queries. Furthermore, in many queries, Spark with SCache outperforms original Spark by an order of magnitude. It is because that those queries contain shuffle-heavy operations such as ”groupby”, ”union”, etc. 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.

1. http://www.tpc.org/tpcds/

6.6 Overhead of Sampling

We evaluate the overhead of sampling with different input sizes and numbers of nodes. In Figure 18, the overhead of sampling only grows with the increase of input size on each node, but remains relatively stable when the cluster size scales up. Since the shuffle data is short-lived, write-once, and read-once, the central controller of SCache does not have to collect and manage complex metadata. Meanwhile, most of the optimizations such as fetching and storing shuffle data are finished by workers independently. So the cost of pre-scheduling algorithm and memory management are unlikely to make the master become the bottleneck of the scalability. Combined with the sampling overhead evaluation, we believe that SCache is scalable.

7 Related Work

Modeling: Most well-known DAG computing frameworks use similar *Bulk Synchronize Parallel* (BSP) [27] model to control data synchronization in each computing phase, i.e. *stage* in Spark, *superstep* in Pregel [28] and so on (Apache Hadoop Madpreduce[1] can also be considered as a special case of only one superstep in BSP model). In the process of optimizing the shuffle phases between adjacent computing phases, we design a performance model to assist in analyzing computing process. Inspired by [29], [30], and [31], we present *Framework Resources Quantification* (FRQ) model to describe the performance of DAG frameworks. In [29], the author designs a model and divides execution into three parts: Map, Shuffle and Reduce. And then the author uses a greedy algorithm to roughly calculate the max, min and mean execution time of map and reduce phases. In [30], the author presents a detailed set of mathematical performance models for describing the execution of a MapReduce job. The execution is separated into the phases: Read, Map, Collect, Spill, Merge, Shuffle, Merge, and Reduce. The performance models describe each above-mentioned phases and combine into a overall MapReduce job model. However, none of the above models meet our requirements for analyzing shuffle. The model in [29] is not able to accurately describe the overhead caused by shuffle under different scheduling strategies. The model in [30] calculates overhead of various phases including shuffle, but most of them are redundant. Different from these models, our FRQ model quantifies computing and I/O resources and displays them in time dimension. FRQ model focuses on describing the overhead caused by shuffle in different scheduling strategies, which satisfy our demand.

Pre-scheduling: Slow-start from Apache Hadoop MapReduce is a classic approach to handle shuffle overhead. Starfish [32] gets sampled data statics for self-tuning system parameters (e.g. slow-start, etc). DynMR [33] dynamically starts reduce tasks in late map stage. All of them have the explicit I/O time in occupied slots. SCache instead starts shuffle pre-fetching without consuming slots. iShuffle [19] 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 [34] aggressively pre-schedules tasks in multiple successive stages to start fetching shuffle. 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.

Delay-scheduling: Delay Scheduling [10] 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, it achieves better data locality. Both Quincy [9] and Fair Scheduling [35] can reduce shuffle data by optimizing data locality of map tasks. But all of them cannot 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 data pre-fetching and in-memory caching.

Network layer optimization: Varys [36] and Aalo [37] 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.

1. http://hadoop.apache.com/

8 Conclusion

In this paper, we present SCache, a cross-framework shuffle optimization for DAG computing frameworks. SCache decouples the shuffle from computing tasks and leverages memory to store shuffle data. By tasks pre-scheduling and shuffle data pre-fetching with application context, SCache significantly mitigates the shuffle overhead. Our evaluations on Spark and Hadoop MapReduce with SCache show that SCache can provide a promising speedup. Furthermore, we propose *Framework Resources Quantification* (FRQ) model to assist in analyzing shuffle process of DAG computing frameworks. At last we use FRQ model to verify SCache shuffle optimization by mathematics. Therefore, with the shared defects of shuffle among different frameworks, we believe that the optimization of SCache is general and easy to adapt.

9 Acknowledge

This work was supported in part by National Key Research & Development Program of China (No. 2016YFB1000502), National NSF of China (NO. 61672344, 61525204, 61732010), and Shanghai Key Laboratory of Scalable Computing and Systems.

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] 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

[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*, 2014, pp. 1–12.

[13] 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.

[14] 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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] D. P. Williamson and D. B. Shmoys, "The design of approximation algorithms. 2010," *preprint http://www. designofapproxalgs. com*, 2010.

[22] 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.

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

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

[25] 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.

[26] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "On availability of intermediate data in cloud computations." in *HotOS*, 2009.

[27] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[29] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 235–244.

[30] H. Herodotou, "Hadoop performance models," *arXiv preprint arXiv:1106.0940*, 2011.

[31] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE, 2010, pp. 373–380.

[32] 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.

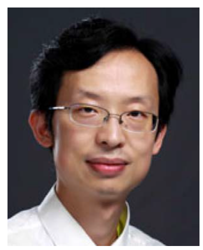
[33] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang, "Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 2.

[34] 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.

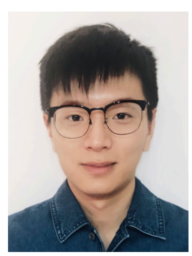
[35] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, and X. Li, "Preemptive reducetask scheduling for fair and fast job completion." in *ICAC*, 2013, pp. 279–289.

[36] 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.

[37] 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



Rui Ren was born in Shaanxi, China, in 1978. He received the B.S. and M.S. degrees from Shanghai Jiao Tong University, China, in 2000 and 2004, respectively. He is currently a lecturer in the School of Software, Shanghai Jiao Tong University, China.



Chunghsuan Wu received the BE degree from Shanghai Jiao Tong University, China, in 2013. He is currently pursuing a ME degree in Shanghai Jiao Tong University, China. His research interests mainly focus on distributed computing.

Zhouwang Fu is a graduated Master student of Shanghai Jiao Tong University in China. He received his Bachelor and Master degree in software engineering Shanghai Jiao Tong University.

Tao Song Tao Song is currently working as a postdoc at Shanghai Jiao Tong University in China. He received his Ph.D. degree in computer science and M.Eng. degree in software engineering from Shanghai Jiao Tong University. His research interests include data center networking, cloud computing, artificial intelligence and swarm intelligence.

Zhengwei Qi Zhengwei Qi received the BEng and MEng degrees from Northwestern Polytechnical University, in 1999 and 2002, and the PhD degree from Shanghai Jiao Tong University, in 2005. Currently, he is a professor in the School of Software, Shanghai Jiao Tong University (China). His research interests include distributed computing, virtualized security, model checking, program analysis and embedded systems.

Haibing Guan Haibing Guan received the PhD degree from Tongji University, in 1999. He is a professor of School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University, and the director of the Shanghai Key Laboratory of Scalable Computing and Systems. His research interests include distributed computing, network security, network storage, green IT, and cloud computing.