Efficient Shuffle Management for DAG Computing Frameworks Based on the FRQ Model

***Abstract*—In industrial large-scale data-parallel analytics, shuffle, or the cross-network read and the aggregation of partitioned data between tasks with data dependencies, usually bring in large overhead. Due to the dependency constraints, execution of those descendant tasks could be delayed by long shuffles. To reduce shuffle overhead, we present *SCache*, an open source plug-in system that particularly focuses on shuffle optimization. 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 also propose a new performance model called *Framework Resources Quantification* (FRQ) model to analyze DAG frameworks and evaluate the SCache shuffle optimization. The FRQ model quantifies the utilization of resources and predicts the execution time of each phase of computing jobs. We have implemented SCache on both Apache Spark and Apache Hadoop MapReduce. The performance of SCache has been 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 MapReduce, SCache optimizes end-to-end Terasort completion time by 15%.**

***Index Terms*—Distributed DAG frameworks, Shuffle, Optimization, Performance model**

I. Introduction

WE are in an era of data explosion — 2.5 quintillion bytes of data are created every day according to IBM's report[1]. In *industry 4.0*, an increasing number of IoT sensors are embedded in the industrial production line [1]. During the manufacturing process, the information about the assembly lines, stations, and machines is continuously generated and collected. Using distributed computing frameworks to analyze industrial big-data is an inevitable trend [2]. Several sophisticated frameworks are used in industrial big-data analysis such as Hadoop MapReduce[2], Spark[3], and Storm[4]. Industrial big-data is more structured, correlated, and ready for analytics than traditional big-data, because industrial big-data is generated by automated equipment [3], [4].

According to a cross-industry study [5], both industrial and traditional big-data share a characteristic during analytical processing — a small fraction of the daily workload uses well over 90% of the cluster's resources, and these workloads often contain a huge shuffle size. According to another MapReduce trace analysis from Facebook, the shuffle phase accounts for 33% of the job completion time on average, and up to 70% in shuffle-heavy jobs [6]. The shuffle phase is crucial and heavily affecting the end-to-end application performance. Most of the popular frameworks 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 [7], and Tez[5], etc. Despite the differences among data-intensive frameworks, the shuffle phase is always essential as communication between successive computation stages.

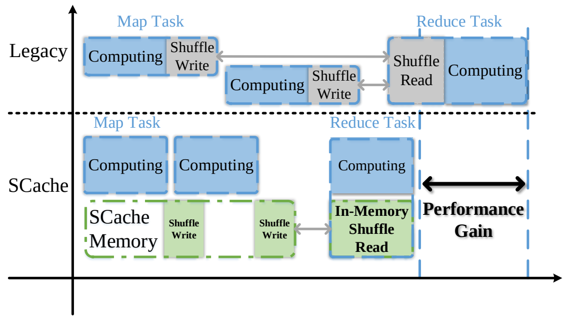


Fig. 1: Workflow Comparison between Legacy DAG Computing Frameworks and Frameworks with SCache

Although continuous efforts of performance optimization have been made among a variety of computing frameworks [8]–[13], the shuffle phase is often poorly optimized in practice. In particular, we observe that one major deficiency lies in the coupled scheduling 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 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 coupled 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 the same

[1]http://www-01.ibm.com/software/data/bigdata/
[2]http://hadoop.apache.com/
[3]https://spark.apache.org/
[4]http://storm.apache.com/
[5]https://tez.apache.org/

execution phase are scheduled at 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. However, such early-start is by no means a panacea. This is mainly because the early-start always introduces an extra early allocation of the slot leading to a slow execution of the current stage.

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[6] 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 more flexible resource management and better multiplexing between the computational and I/O resources. Second, 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 III).

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 IV).

[6]https://github.com/frankfzw/SCache

We also propose a new performance model called *Framework Resources Quantification* (FRQ) model. The FRQ model quantifies computing and I/O resources and visualizes the resources scheduling strategies of DAG frameworks in the time dimension. We use the FRQ model to assist in analyzing the deficiencies of resources scheduling and optimize it. In the industrial production, the FRQ model can discover the irrationality of resource scheduling in big-data analysis by revealing the relationships between the various phases (Section V).

We have implemented SCache on both Spark and Hadoop MapReduce. 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 a basic test - *GroupByTest*. We also evaluate the system with Terasort[7] benchmark and standard workloads like TPC-DS[8] for multi-tenant modeling. On Apache Hadoop, we focus on Terasort benchmark. In a nutshell, SCache can eliminate explicit shuffle time 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 optimizes end-to-end Terasort completion time by 15%.

II. Background and Observations

In this section, we first study the typical shuffle characteristics (II-A), and then spot the opportunities to achieve shuffle optimization (II-B).

A. *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) 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.

2) 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 [14]. For those shuffle-heavy jobs, the shuffle latency may even dominate Job Completion Time.

[7]https://github.com/ehiggs/spark-terasort
[8]http://www.tpc.org/tpcds/

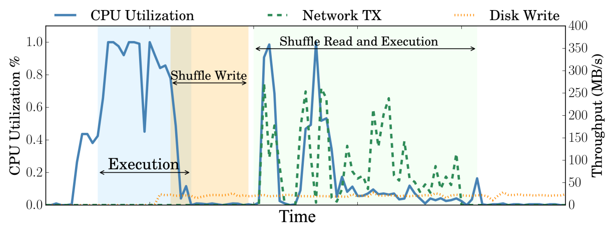


Fig. 2: CPU Utilization and I/O Throughput of a Node During a Spark Single Shuffle Application

B. 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.

1) Coarse Granularity Resource Allocation: As shown in Figure 2, the network transfer of shuffle data introduces an explicit I/O delay during *shuffle read*. Both *shuffle write* and *shuffle read* occupy the slot without significantly involving CPU. The current coarse slot-task mapping results in an imbalance between task's resource demand and slot allocation thus decreasing the resource utilization.

2) Synchronized Shuffle Read: The synchronized shuffle read requests cause several bursts of network traffic, which may result in network congestion and further slow down the network transfer.

3) Inefficient Persistent Storage Operation: Both shuffle write and shuffle read are tightly coupled with task execution, which results in a blocking I/O operation. This blocking I/O operation may introduce significant latency, especially in an I/O performance bounded cluster. Besides, most of the DAG frameworks store shuffle data on disks (e.g., Spark, Hadoop MapReduce, Dryad, etc). We argue that the memory capacity is large enough to store the short-living shuffle data during shuffle phases since several memory-based distributed storage systems have been proposed [10], [15].

4) Multi-round Tasks Execution: Both experience and DAG framework manuals (e.g., Hadoop and Spark) recommend that multi-round execution of each stage will benefit the performance of applications. Since the shuffle data becomes available when map tasks complete, and the network is idle during the map stage (*Network TX* during map stage in Figure 2), we propose an optimization that starts *shuffle read* ahead of reduce stage to overlap the I/O operations in multi-round map tasks.

III. Shuffle Optimization

This section presents the detailed methodologies to achieve the three design goals. The out-of-framework shuffle data

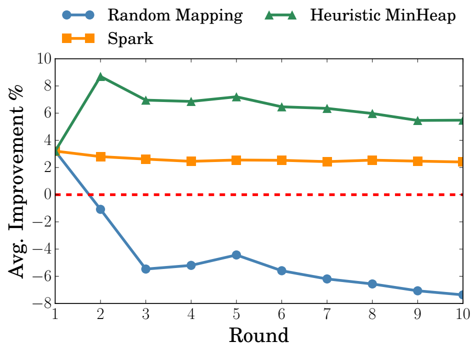


Fig. 3: Stage Completion Time Improvement of OpenCloud Trace

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.

A. 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.

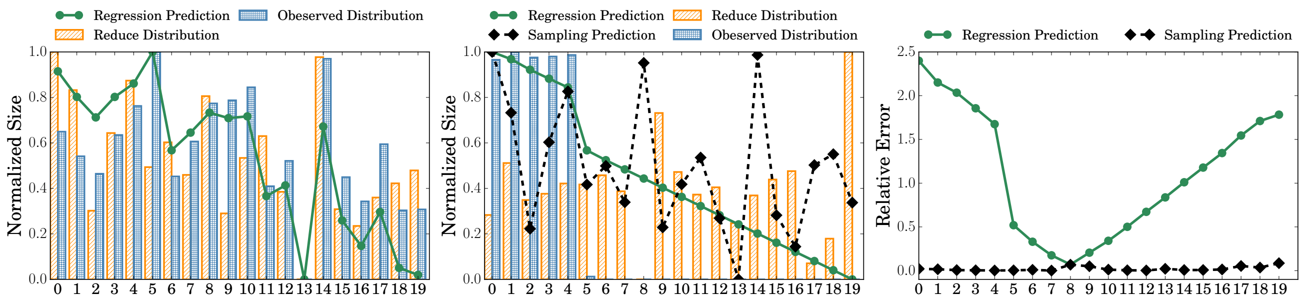
From the perspective of reduce task, SCache provides an API named $getBlock$ to replace the original implementation of shuffle read. With the precondition of shuffle data pre-fetching, the $getBlock$ leverages the memory copy to fetch the shuffle data from the local memory of SCache.

B. Pre-scheduling 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. And 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.

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[9] for the simulation. The average shuffle read time is 3.2% of total reduce completion time. As shown in Figure 3, 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

[9]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. 4: Reduction Distribution Prediction

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]. 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.

2) Shuffle Output Prediction: The problem of random mapping is obviously caused by application context (e.g., shuffle data size) ignorance. For the most DAG applications with random large scale input, the shuffle data size can be predicted accurately by a liner regression model based on the observation that the ratio of map output size and input size are invariant given the same job configuration [17].

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.
```

However, the linear regression model can fail in some uncertainties introduced by sophisticated frameworks like Spark. 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 4a and Figure 4b. With a random input and a hash partitioner in Figure 4a, 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 in Figure 4b results in a deviation from the linear regression model, because the RangePartitioner might introduce an extreme high data locality skew.

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. Finally, the partitioned outputs are collected with the $InputSize_j$ as the weight of the samples. The $inputSize_j$ is the input size of jth map task. Note that sampling does not consume the input data of map tasks.

Figure 4c proves that the sampling prediction can provide much more accurate result than the linear regression. We will show the overhead evaluation of sampling in Section VI.

3) Heuristic MinHeap Scheduling: 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)[10] to achieve $4/3$-*approximation* optimum. Since the sizes of tasks are considered while scheduling, *Heuristic MinHeap Scheduling* can achieve a shorter makespan than Spark FIFO which is a 2-*approximation* optimum. Simulation of OpenCloud trace in Figure 3 also shows that *Heuristic MinHeap Scheduling* 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

[10]http://www.designofapproxalgs.com/

of performance degradation. Inside the $SWAP\_TASKS$, tasks will be selected and swapped without exceeding the $upper\_bound$.

4) Cope with Multiple Shuffle Dependencies: A reduce stage can have more than one shuffle dependency in the current DAG computing frameworks. To cope with multiple shuffle dependencies, we present the *Accumulated Heuristic Scheduling* algorithm. As illustrated in Algorithm 2, the sizes of previous *shuffles* scheduled by *Heuristic MinHeap Scheduling* 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 $SWAP\_TASKS$.

Algorithm 2 Accumulated Heuristic Scheduling for Multi-Shuffles

```
1:  procedure M_SCHEDULE($m, host\_id, p\_reduces, shuffles$)
2:      $m \leftarrow$ partition number of map tasks    ▷ shuffles are the previous schedule result
3:      for all $r$ in $p\_reduces$ do
4:          $r.size$ += $shuffles[r.rid].size$
5:          $new\_prob \leftarrow shuffles[r.rid].size / r.size$
6:          if $new\_prob \geq r.prob$ then
7:              $r.prob \leftarrow new\_prob$
8:              $r.host\_id \leftarrow shuffles[r.rid].assigned\_host$
9:      $M \leftarrow SCHEDULE(m, host\_id, p\_reduces)$
10:     for all $host\_id$ in $M$ do                ▷ Re-shuffle
11:         for all $r$ in $M[host\_id].reduces$ do
12:             if $host \neq shuffles[r.rid].assigned\_host$ then
13:                 Re-shuffle data to $host$
14:                 $shuffles[r.rid].assigned\_host \leftarrow host$
        return $M$
```

IV. Implementation

This section presents an overview of the implementation of SCache. We first present the system overview and the detail of sampling in Subsection IV-A. The following IV-B subsection focuses on two constraints on memory management.

A. 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 5, SCache employs the legacy master-slaves architecture for the 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 the original scheduler in framework. When a DAG job is submitted, the DAG information is generated in framework

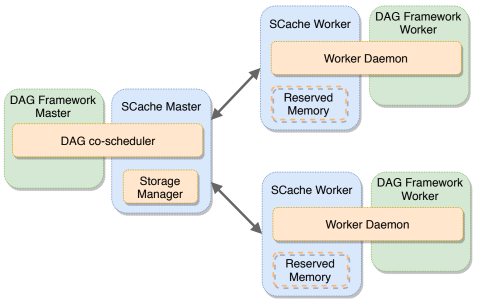


Fig. 5: SCache Architecture

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 IV-A1.

When a map task finishes computing, the shuffle write implementation of the 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 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 to start pre-fetching on each worker. To enforce the DAG framework to run according to the SCache pre-scheduled results, we also insert some lines of codes in framework scheduler.

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 [18] on each partition. The result will be aggregated 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.

B. Memory Management

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*.

1) All-or-Nothing Constraint: Based on the observation in Section II-B4, 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. Therefore, SCache master sets blocks of one round as the minimum unit of storage management. We refer to this as the all-or-nothing constraint.

2) Context-Aware-Priority Constraint: 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: (a) For the tasks in the different stages, SCache sets the higher priority to storage units with a earlier submission time; (b) For the tasks in the same stage, SCache sets the higher priority to storage units with a smaller task ID.

V. Framework Resources Quantification Model

In this section, we introduce *Framework Resources Quantification* (FRQ) model to describe the performance of DAG frameworks. The FRQ model quantifies computing and I/O resources and visualizes them in the time dimension. In the industrial production, the FRQ model is able to predict the execution time required by the application under any circumstances, including different DAG frameworks, hardware environments, etc. The FRQ model assist us in analyzing the resource scheduling of DAG framework and evaluate their performance. We first introduce the FRQ model in Subsection V-A. In the following Subsection V-B, we use the FRQ model to describe three different computation jobs and analyze their performance. In the last Subsection V-C, we use the actual experimental results to evaluate the FRQ model.

A. The FRQ Model

To better analyze the relationship between the computation phase and the shuffle phase, we propose the FRQ model. After quantifying computing and I/O resources, the FRQ model can describe different resource scheduling strategies. For convenience, we introduce the FRQ model by taking a simple MapReduce job as an example in this section.

Figure 6 shows how the FRQ model describes a MapReduce task. The FRQ model has five input parameters:

- Input Data Size (D): The Input data size of the job.
- Data Conversion Rate (R): The conversion rate of the input data to the shuffle data during a computation phase. This rate depends on the algorithm used in the computation phase.
- Computation Round Number (N): The number of rounds needed to complete the computation phase. These rounds depend on the current computation resources and the configuration of the framework. Take Hadoop MapReduce as an example. Suppose we have a cluster with 50 CPUs and enough memory, the map phase consists of 200 map tasks, and each map requires 1 CPU. Then we need 4 rounds of computation to complete the map phase.
- Computation Speed (V_i): The computation speed for each computation phase. This speed depends on the algorithm used in the computation phase.
- Shuffle Speed $(V_{Shuffle})$: Transmission speed in the shuffle phase. This speed depends on network and storage device bandwidth.

The FRQ model calculates the execution time of each phase of the job with these five parameters. As shown in Figure 6, the total execution time of a job is the sum of the map phase time and reduce phase time: $T_{Job} = T_{Map} + T_{Reduce}$.

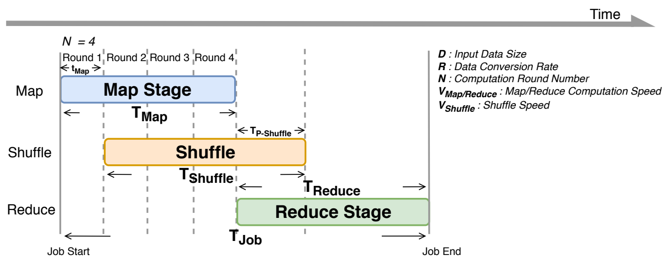


Fig. 6: Framework Resources Quantification (FRQ) Model with Full Parallel MapReduce

$$T_{Job} = T_{Map} + T_{Reduce} \tag{1}$$

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

$$T_{Map} = \frac{D}{V_{Map}} \tag{2}$$

The reduce phase time formula is as follows:

$$T_{Reduce} = \frac{D \times R}{V_{Reduce}} + K \times T_{P\_Shuffle} \tag{3}$$

$\frac{D\times R}{V_{Reduce}}$ represents the ideal computation time of a reduce phase, and $(K \times T_{P\_Shuffle})$ represents the computing overhead. We use $T_{P\_Shuffle}$ to represent the overlap time between the shuffle phase and the reduce phase. $T_{Shuffle}$ represents the total time of the 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 V-B. 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 needs to wait for the transfer results. This waiting causes the overhead. The FRQ model uses K to indicate the extent of the waiting.

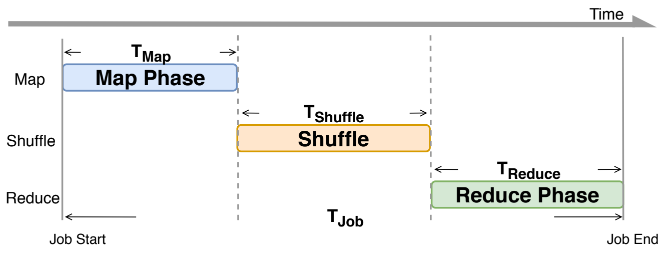
The shuffle phase time formula is as follows:

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

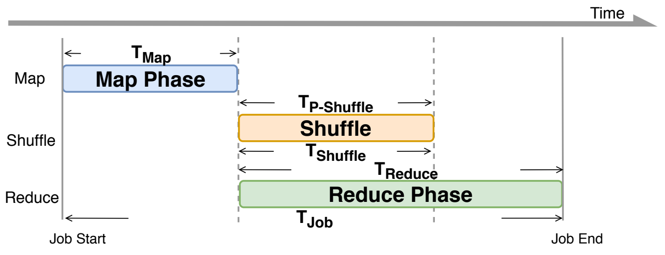
According to Equation 1&3, we can optimize the job completion time by reducing $T_{P\_Shuffle}$. Improving I/O speed is an effective way to reduce shuffle time. Another optimization method is to use the idle I/O resources in the map phase for pre-fetching (see Figure 6). Both of the above methods can effectively reduce $T_{P\_Shuffle}$.

B. Model Analysis

The FRQ model can describe a variety of resource scheduling strategies. First, we analyze a simple scheduling strategy which used by Apache Spark by default. As shown in Figure 7a, the FRQ model describes a MapReduce job that is entirely serially executed. The overlap time between the shuffle phase and the reduce phase is 0, in which case $T_{P\_Shuffle}$ is 0.

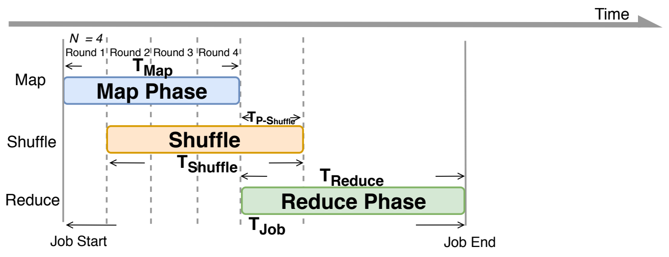


(a) Full Serial MapReduce

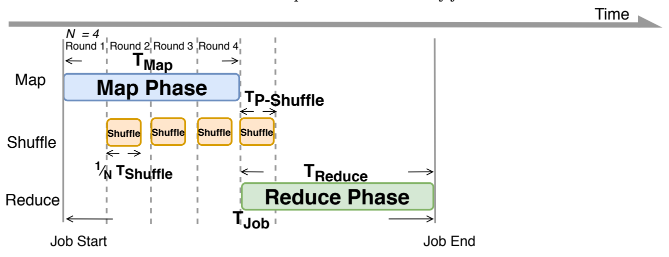


(b) Half Parallel MapReduce

Fig. 7: 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. 8: Framework Resources Quantification (FRQ) Model with Full Parallel MapReduce in Different Environments

Therefore, the overhead of the reduce phase is 0. The total execution time of a job is also different from the above:

$$T_{Job} = T_{Map} + T_{Shuffle} + T_{Reduce} \tag{5}$$

Due to serialization, the I/O resource is idle during the reduce phase and map phase. This scheduling strategy is simple and has much room for optimization.

Figure 7b shows a more efficient scheduling strategy which is used by Hadoop MapReduce. 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 increases (according to Equation 3). Because the shuffle phase and the computation phase are executed in parallel, the total execution time of a job is the sum of T_{Map} and T_{Reduce} (see Equation 1). The execution time of the shuffle phase is hidden in the reduce phase. However, we also found that the I/O resource in the map phase is still idle. This scheduling strategy can still be optimized.

Figure 8 shows the scheduling strategy for Hadoop MapReduce with SCache (Suppose N is 4). SCache also overlap the shuffle phase and the map phases by starting pre-fetching and pre-scheduling in the map phase. This scheduling strategy avoid the I/O resource being idle in the map phase. According to the design of SCache pre-fetching, we found that using the FRQ model to describe the scheduling strategy of SCache needs to distinguish two situations:

1) $V_{Map} \times R \geq V_{Shuffle}$ (Figure 8a)
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 ($V_{Shuffle}$). In this situation, the shuffle phase is uninterrupted. The I/O resource will be fully utilized during the whole shuffle phase. As a result, the formula of $T_{P\_Shuffle}$ is as followed:

$$T_{P\_Shuffle} = T_{Shuffle} - \frac{(N-1) \times T_{Map}}{N} \tag{6}$$

2) $V_{Map} \times R < V_{Shuffle}$ (Figure 8b)
When the shuffle speed ($V_{Shuffle}$) is faster, SCache needs to wait for shuffle data to be generated. As Figure 8b shown, the shuffle phase will be interrupted in each round. In this case, the formula of $T_{P\_Shuffle}$ is as followed:

$$T_{P\_Shuffle} = T_{Shuffle} \times \frac{1}{N} \tag{7}$$

Compared to the original Hadoop MapReduce resource scheduling strategy, Hadoop MapReduce with SCache shortens $T_{P\_Shuffle}$ and thus shortens T_{Reduce}. This is how pre-fetching optimizes the total execution time of a job.

C. Model evaluation

To evaluate the FRQ model, we run experiments on two environments: (a) 50 Amazon EC2 m4.xlarge nodes cluster as shown in Subsection VI-A; (b) 4 in-house nodes cluster with 128GB memory and 32 cores per node. To simplify the calculation of the FRQ model, we run the Terasort as an experimental application on a Hadoop MapReduce framework. We deployed Hadoop with SCache and without SCache in both environments.

Table I shows the calculational results of the FRQ model in the in-house environment. The 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. In the Hadoop with SCache, Terasort satisfies the situation in Figure 8a ($V_{Map} \times R \geq V_{Shuffle}$). In the original Hadoop, since pre-fetching is not used, $T_{P\_Shuffle}$ is equal to $T_{Shuffle}$(see

	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%

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

TABLE I: Hadoop MapReduce on 4 nodes cluster in the FRQ model

	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%

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

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

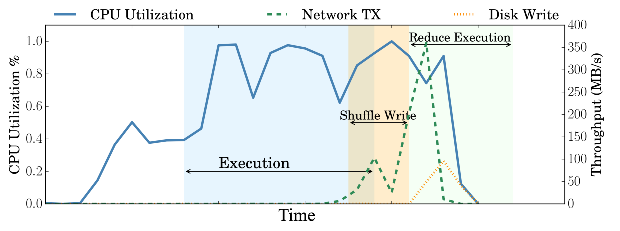


Fig. 9: CPU utilization and I/O throughput of a node during a Spark single shuffle application with SCache

Equation 4). $ExpT_{Job}$ represents the actual experiment data, we calculate $Error$ according to T_{Job} and $ExpT_{Job}$.

Table II shows the calculational results of the 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 8b ($V_{Map} \times R < V_{Shuffle}$), thus the formula of $T_{P\_Shuffle}$ is Equation 7.

Regarding accuracy, the experimental values are all larger 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, etc. This overhead will be amplified when the input data is small or the total execution time is short. Overall, the error between T_{Job} and $ExpT_{Job}$ is mainly below 10%, such errors are acceptable. Therefore, we believe that the FRQ model can accurately describe DAG frameworks.

VI. Evaluation

This section reveals the evaluation of SCache with comprehensive workloads and benchmarks which include common operations in the industrial big-data analysis. We implement and evaluate SCache on Spark and Hadoop MapReduce, since they are the two most distributed computing frameworks using in industrial big-data analysis. First, we evaluate the Spark with SCache in 2 different benchmarks. Second, 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.

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 optimizes job completion time by up to 15% and an average of 13%.

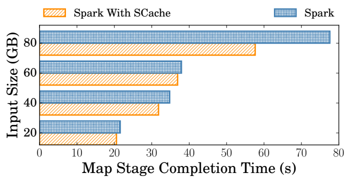
A. Setup

We modified Spark to enable shuffle optimization of SCache as a representative. The shuffle configuration of Spark is set to the default[11]. We run the experiments on a 50-node m4.xlarge cluster on Amazon EC2[12]. 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).

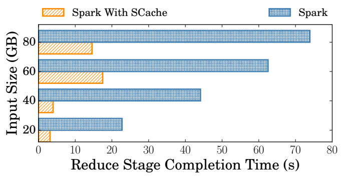
B. Spark with SCache

1) Simple DAG Analysis: As shown in Figure 9, we first run the same single shuffle test shown in Figure 2. Note that since the completion time of whole job is about 50% less than Spark without SCache, the duration of Figure 9 is cut in half as well. An overlap among CPU, disk, and network can be easily observed in Figure 9. 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

[11]http://spark.apache.org/docs/1.6.2/configuration.html
[12]http://aws.amazon.com/ec2/

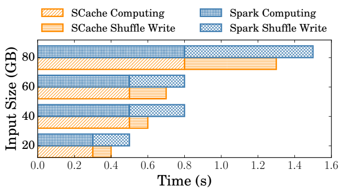


(a) Map Stage Completion Time

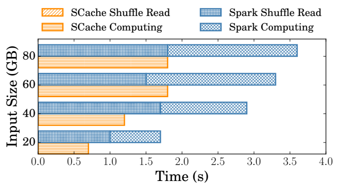


(b) Reduce Stage Completion Time

Fig. 10: Stage Completion Time of Single Shuffle Test



(a) Median Task in Map Stages



(b) Median Task in Reduce Stages

Fig. 11: Median Task Completion Time of Single Shuffle Test

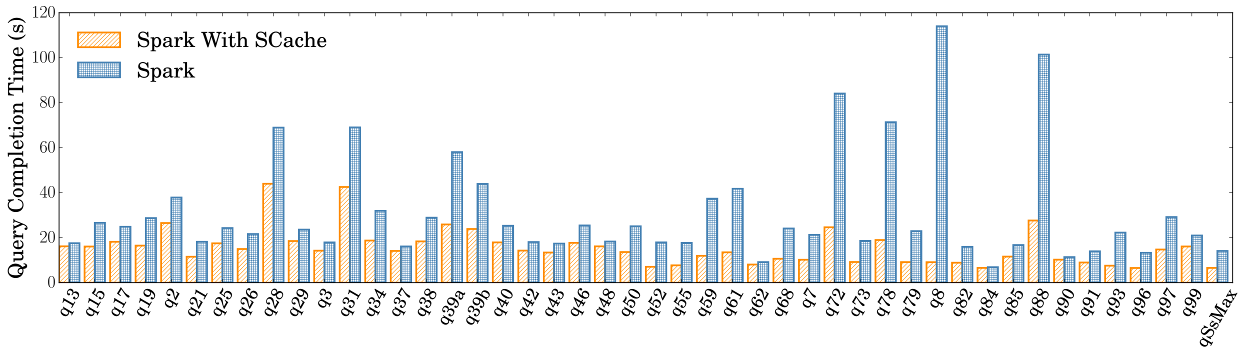


Fig. 12: TPC-DS Benchmark Evaluation

and multiplexing of hardware resource are increased, thus improving the performance of Spark.

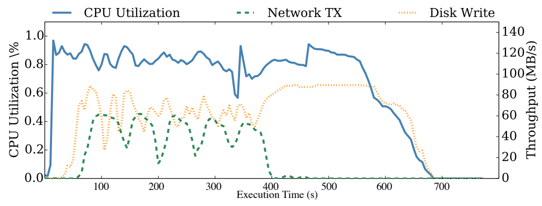
In the map task, the disk operations are replaced by the memory copies to decouple the shuffle write. It helps eliminate 40% of shuffle write time (Figure 11a), which leads to a 10% improvement of map stage completion time in Figure 10a. Note that the shuffle write time can be observed even with the optimization of SCache. The reason is that before moving data out of Spark's JVM, the serialization is inevitable and CPU intensive [19].

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. As a result, the combination of these optimizations decreases 100% overhead of the shuffle read in a reduce task (Figure 11b). In addition, the heuristic algorithm can achieve a balanced pre-scheduling result, thus providing 80% improvement in reduce stage completion time (Figure 10b).

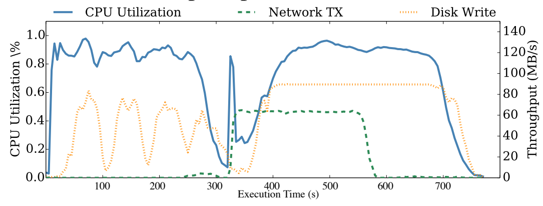
Overall, SCache can help Spark decrease by 89% overhead of the whole shuffle process.

2) Industrial Production Workload: We also evaluate some queries from TPC-DS[13]. 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 12, 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.

[13]http://www.tpc.org/tpcds/



(a) Hadoop MapReduce with SCache



(b) Hadoop MapReduce without SCache

Fig. 13: CPU utilization and I/O throughput of a node during a Hadoop MapReduce Terasort job

C. Hadoop MapReduce with SCache

To prove SCache compatibility as a cross-framework plug-in, we also implemented SCache on Hadoop MapReduce. Although the simple DAG computing alleviates the effect of *pre-scheduling*, the shuffle-heavy jobs can still be optimized by the SCache shuffle data management.

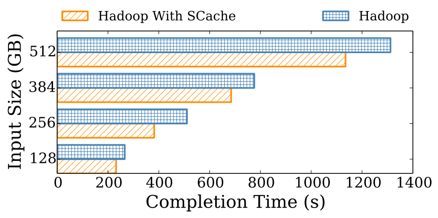


Fig. 14: Hadoop MapReduce Terasort Completion Time

As Figure 13 shows, Hadoop MapReduce with SCache brings 15% of total time optimization with 384GB input data size. As shown in Figure 13b, Hadoop MapReduce without SCache writes intermediate data locally in the map phase. The shuffle phase and the reduce phase start simultaneously. Because a large amount of shuffle data reaches the network bottleneck, the beginning part of reduce phase needs to wait for network transfer. This causes the CPU resources to be idle. On the other hand, in Figure 13a, Hadoop MapReduce with SCache starts pre-fetching in the map phase. This avoids the reduce phase waiting for the shuffle data. Furthermore, pre-fetching utilizes the idle I/O throughput in the map phase. As shown in Figure 14, after better fine-grained utilization of hardware resources, Hadoop MapReduce with SCache optimizes Terasort overall completion time by up to 15% and an average of 13% with input data sizes from 128GB to 512GB.

VII. Related Work

Modeling [20]–[22] have proposed several methods to model the DAG computing process. However, the above models are not able to accurately describe the overhead caused by the shuffle process under different scheduling strategies. The FRQ model focuses on describing the overhead caused by the shuffle process in different scheduling strategies, which satisfies our demand.

Industrial big-data In the *Industry 4.0* which leading by the German Government, industrial big-data refers to a huge amount of time-series data which generated by industrial equipment in smart factories. More than 1000 Exabytes industrial data is annually generated by smart factories and the data is expected to increase 20-fold in the next ten years [23]. Nowadays, the real challenge of big-data is not how to collect it, but how to manage it logically and efficiently [4]. Several distributed computing frameworks become efficient tools in industrial big-data analysis [1], [24], [25].

Pre-scheduling Besides *slow-start* from Hadoop MapReduce, [17], [26]–[28] have also proposed several methods to improve the DAG frameworks. However, we have proved that randomly assign tasks may hurt the overall performance in Section III-B1. Different from these works, SCache pre-schedules multiple shuffles without breaking load balancing.

Network layer optimization According to [29], [30], we can also combine SDN with SCache to further improve the performance on the network later.

VIII. 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 task pre-scheduling and shuffle data pre-fetching with application context, SCache significantly mitigates the shuffle overhead. The benchmark using in evaluations covers the most situations of industrial big-data analysis. The evaluations show that SCache can provide a promising speedup in both traditional and industrial big-data workloads. Furthermore, we propose *Framework Resources Quantification* (FRQ) model to assist in analyzing shuffle process of DAG computing frameworks. At last, we evaluate the SCache shuffle optimization by the FRQ model.

References

[1] P. Lade, R. Ghosh, and S. Srinivasan, "Manufacturing analytics and industrial internet of things," IEEE Intelligent Systems, vol. 32, no. 3, pp. 74–79, 2017.

[2] J. Lee, H.-A. Kao, and S. Yang, "Service innovation and smart analytics for industry 4.0 and big data environment," Procedia Cirp, vol. 16, pp. 3–8, 2014.

[3] P. Basanta-Val, "An efficient industrial big-data engine," IEEE Transactions on Industrial Informatics, vol. 14, no. 4, pp. 1361–1369, 2018.

[4] Z. Lv, H. Song, P. Basanta-Val, A. Steed, and M. Jo, "Next-generation big data analytics: State of the art, challenges, and future research topics," IEEE Transactions on Industrial Informatics, vol. 13, no. 4, pp. 1891–1899, 2017.

[5] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," Proceedings of the VLDB Endowment, vol. 5, no. 12, pp. 1802–1813, 2012.

[6] 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.

[7] 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.

[8] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li, "A parallel random forest algorithm for big data in a spark cloud computing environment," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 4, pp. 919–933, 2017.

[9] 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

[10] 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.

[11] B. Heintz, A. Chandra, R. K. Sitaraman, and J. Weissman, "End-to-end optimization for geo-distributed mapreduce," IEEE Transactions on Cloud Computing, vol. 4, no. 3, pp. 293–306, 2016.

[12] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou, "Improving performance of heterogeneous mapreduce clusters with adaptive task tuning," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 3, pp. 774–786, 2017.

[13] M. Wasi-ur Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, "A comprehensive study of mapreduce over lustre for intermediate data placement and shuffle strategies on hpc clusters," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 3, pp. 633–646, 2017.

[14] 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.

[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] Y. Guo, J. Rao, D. Cheng, and X. Zhou, “ishuffle: Improving hadoop performance with shuffle-on-write,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 6, pp. 1649–1662, 2017.

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

[19] 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.

[20] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, “Hadoop performance modeling for job estimation and resource provisioning,” IEEE Transactions on Parallel & Distributed Systems, no. 2, pp. 441–454, 2016.

[21] F. Farhat, D. Tootaghaj, Y. He, A. Sivasubramaniam, M. Kandemir, and C. Das, “Stochastic modeling and optimization of stragglers,” IEEE Transactions on Cloud Computing, 2016.

[22] K. Chen, J. Powers, S. Guo, and F. Tian, “Cresp: Towards optimal resource provisioning for mapreduce computing in public clouds,” IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1403–1412, 2014.

[23] S. Yin and O. Kaynak, “Big data for modern industry: challenges and trends [point of view],” Proceedings of the IEEE, vol. 103, no. 2, pp. 143–146, 2015.

[24] X. Li, J. Song, and B. Huang, “A scientific workflow management system architecture and its scheduling based on cloud service platform for manufacturing big data analytics,” The International Journal of Advanced Manufacturing Technology, vol. 84, no. 1-4, pp. 119–131, 2016.

[25] M. H. ur Rehman, E. Ahmed, I. Yaqoob, I. A. T. Hashem, M. Imran, and S. Ahmad, “Big data analytics in industrial iot using a concentric computing model,” IEEE Communications Magazine, vol. 56, no. 2, pp. 37–43, 2018.

[26] 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.

[27] 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.

[28] 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.

[29] P. Qin, B. Dai, B. Huang, and G. Xu, “Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data,” IEEE Systems Journal, vol. 11, no. 4, pp. 2337–2344, 2017.

[30] X. Huang, S. Cheng, K. Cao, P. Cong, T. Wei, and S. Hu, “A survey of deployment solutions and optimization strategies for hybrid sdn networks,” IEEE Communications Surveys & Tutorials, 2018.