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

Rui Ren, Chunghsuan Wu, Zhouwang Fu, Tao Song, Yanqiang Liu*, Zhengwei Qi and Haibing Guan

ARTICLE INFO

Keywords:
Distributed DAG frameworks
Shuffle
Optimization
Performance model

ABSTRACT

In large-scale data-parallel analytics, shuffle, namely the cross-network read and the aggregation of partitioned data between tasks with data dependencies, usually bring in large overhead. To reduce shuffle overhead, we present *SCache*, an open-source plug-in system that particularly focuses on shuffle optimization. SCache adopts 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 DAG jobs. We have implemented SCache on both Spark and 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%.

1. 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].

According to a cross-industry study [3], 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 [4]. 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[2], *RDDs* in Spark[3], vertices in Dryad [5], etc. Shuffle phase is always essential as communication between successive computation stages.

Although continuous efforts of performance optimization have been made among a variety of computing frameworks [6, 7, 8, 9, 10, 11, 12, 13], the shuffle phase is often poorly optimized in practice. One of the major deficiencies of the shuffle phase is lacking coordinated management among different system resources [14]. As Figure 1 shows, the *shuffle write* is responsible for writing intermediate results to disk. And the *shuffle read* fetches intermediate results from remote disks through the network. Once scheduled, a fixed bundle of resources (i.e., CPU, memory, disk, and network) named *slot* is assigned to a task, and *slot*s are released only after the task finishes. Such task aggregation together with the coupled scheduling effectively simplifies task management. However, 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, since the shuffle read phase starts fetching data only after the corresponding reduce task starts, 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.

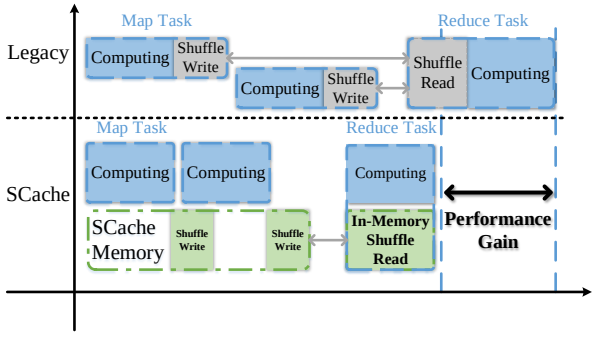


Figure 1: Workflow Comparison between Legacy DAG Computing Frameworks and Frameworks with SCache

To optimize the data shuffling without significantly chang-

*Corresponding author
oai@sjtu.edu.cn (Y. Liu)
ORCID(s): 0000-0002-8343-7923 (Y. Liu)

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

ing DAG frameworks, we propose *S(huffle)Cache*, an open source[4] plug-in system for different DAG computing frameworks. Specifically, SCache takes over the whole shuffle phase from the underlying framework. 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 and pre-fetches the shuffle data. 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.

We evaluate SCache on a 50-node Amazon EC2 cluster on both Spark and Hadoop MapReduce. 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[5], a standardized industry benchmark, on average on Apache Spark. On Apache Hadoop MapReduce, SCache also optimizes end-to-end Terasort completion time by 15%.

The rest of the paper is organized as follows: We present the methodologies of optimization which using by SCache in Section 3 and detail the implementation of SCache in Section 4. We introduce the FRQ performance model in Section 5. In Section 6, we present comprehensive evaluations about SCache and the FRQ model. We also discuss the related work in Section 7 and conclude in Section 8.

2. Background and Observations

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.

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 [15]. 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].

2.1. Observations

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`. 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.

2.1.1. Coarse Granularity Resource Allocation

When a slot is assigned to a task, it will not be released until the task completes. On the reduce side, the network transfer of shuffle data introduces an explicit I/O delay during shuffle read. Meanwhile, 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. Unfortunately this defect exists not only in Spark [16] but also Hadoop MapReduce and Apache Tez [17]. A finer granularity resource allocation scheme should be provided to reduce these delays.

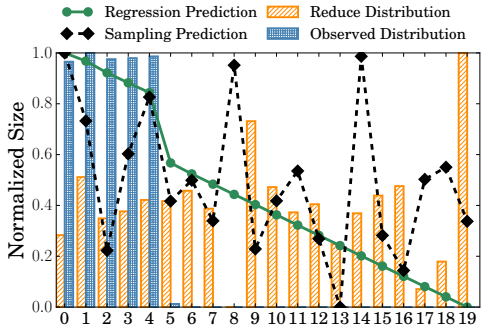
2.1.2. Synchronized Shuffle Read

Almost all reduce tasks start shuffle read simultaneously. The synchronized shuffle read requests cause a burst of network traffic. 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 [5], etc.

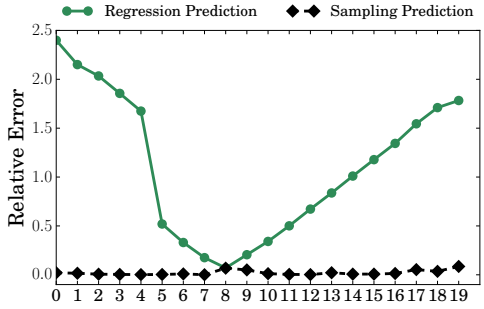
2.1.3. Inefficient Persistent Storage Operation

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 is less than 25% of input data. The data reported in [18] 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 [19, 9, 20] to move data back to memory, but most of the DAG frameworks still store shuffle data on disks (e.g., Spark [16], Hadoop MapReduce, Dryad [5], etc). We argue that the memory capacity is large enough to store the short-living shuffle data with cautious management.

[4] https://github.com/frankfzw/SCache
[5] http://www.tpc.org/tpcds/



(a) Linear Regression and Sampling Prediction of Range Partitioner



(b) Prediction Relative Error of Range Partitioner

Figure 2: Reduction Distribution Prediction

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.
- Reduce tasks should be pre-scheduled without launching to achieve shuffle data pre-fetching.

3. Shuffle Optimization

This section presents detailed methodologies to achieve shuffle optimization. Firstly, we discuss the reason causes shuffle overhead in the DAG frameworks. In the following subsection, we propose a shuffle data management system to decouple shuffle from execution. And we propose the pre-scheduling and pre-fetching to hide shuffle overhead in multi-round map tasks. Furthermore, two heuristic algorithms (Algorithm 1, 2) are used to improve the accuracy of prediction in the pre-scheduling.

3.1. Decouple Shuffle from Execution

To decouple the shuffle from reduce tasks, we propose a shuffle data management system called SCache to take over all shuffle data from the DAG frameworks. SCache provides two APIs named *putBlock* and *getBlock* to manage

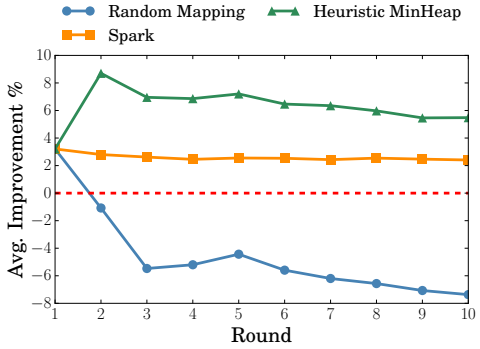


Figure 3: Stage Completion Time Improvement of OpenCloud Trace

the shuffle data. On the map side, after shuffle data blocks are produced, the map task uses *putBlock* to transfer the data blocks to SCache. Inside the *putBlock*, SCache uses memory copy to move the shuffle data blocks to SCache's reserved memory and release the slot immediately. After getting the shuffle data, SCache starts to shuffling data immediately. On the reduce side, due to the shuffle data pre-fetching of SCache, the reduce task use *getBlock* to get the shuffle data from SCache.

SCache caches the shuffle data on the memory instead of storing on disk like Spark or Hadoop MapReduce. In this aspect, SCache optimizes the time of shuffle read and shuffle write since memory gets better I/O performance.

3.2. Pre-scheduling and Pre-fetching Shuffle Data

The pre-scheduling and pre-fetching are the most critical aspects of the optimization. SCache starts pre-fetching and hides shuffle overhead into multi-round map tasks. However, the shuffle data cannot be pre-fetched without the awareness of task-node mapping.

To optimize shuffle phase, we propose a co-scheduling scheme with two heuristic algorithms (Algorithm 1, 2). The DAG frameworks follow the co-scheduler to start reduce tasks.

3.2.1. Shuffle Output Prediction

The simplest way of pre-scheduling is mapping tasks to nodes randomly and evenly. But without considering job context, the random mapping scheduling shows poor performance due to data skew and ignoring data locality. For the most DAG applications with random large scale input, the shuffle output can be predicted accurately by a linear regression model based on the observation that the ratio of map output size and input size are invariant given the same job configuration [21]. However, the linear regression model can fail in some scenarios. For example, some customized partitions may cause large inconsistency between observed map output distribution and the final reduce input distribution. To illustrate the case, we present a particular spark job which uses the Spark RangePartitioner in Figure 2a. The observed map outputs are picked randomly. The job uses the Spark RangePartitioner and introduces an extremely high

data skew. Due to the extremely high data skew introduced by the partition, the linear regression model cannot fit the result well. 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 2a). The data locality skew results in a missing of other reduce tasks' data in the observed map outputs.

To solve this problem, we propose a new methodology named *weighted reservoir sampling*. SCache uses this method instead of the linear regression to predict output when using a RangePartitioner or a customized non-hash partitioner. 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. $BlockSize_{ji}$ represents the size of block which is produced by map $task_j$ for reduce $task_i$:

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

In Figure 2a, when s is set to 3, the result of sampling prediction is much better than linear regression. Figure 2b further proves that the sampling prediction can provide a much more accurate result than the linear regression.

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

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

The $reduceSize_i$ represents the size of a reduce task, which is represented by $reduceSize_i = \sum_{j=0}^{m} BlockSize_{ji}$. We use $prob_i$ to achieve a better data locality while performing shuffle pre-scheduling.

3.2.2. Heuristic MinHeap Scheduling

To balance load while minimizing the network traffic, we present the *Heuristic MinHeap Scheduling* algorithm (Algorithm 1). To keep the pre-scheduling load balance, we maintain a min-heap in the first *while* loop (i.e., line 6-11). We use the min-heap to simulate the load of each node and apply the longest processing time rule (LPT)[6] to achieve $4/3$-*approximation* optimum. Spark FIFO only can achieve 2-*approximation* optimum. 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*. Since the algorithm only maintains a min-heap and traverses *reduce* for swapping, the algorithm needs $O(n)$ operations.

To evaluate *Heuristic MinHeap Scheduling* algorithm, we use traces from OpenCloud[7] for the simulation. As shown in Figure 3, we run the simulation under three scheduling schemes: Random Mapping, Spark FIFO, and heuristic MinHeap. After balancing load based on the job context (e.g. shuffle size, data locality), *Heuristic MinHeap Scheduling* has a better improvement (average 5.7%) than Spark (average 2.7%) and Random Mapping.

Algorithm 1 Heuristic MinHeap Scheduling for Single Shuffle

```
1:  procedure SCHEDULE(m, host_ids, p_reduces)
2:      m ← partition number of map tasks
3:      R ← sort p_reduces by size in decreasing order
4:      M ← min-heap {host_id → ([reduces], size)}
5:      idx ← 0
6:      while idx < len R do
7:          M[0].size += R[idx].size
8:          M[0].reduces.append(R[idx])
9:          R[idx].assigned_id ← M[0].host_id
10:         Sift down M[0] by size
11:         idx ← idx + 1
12:     end while
13:     max ← maximum size in M
14:     for all reduce in R do          ▷ Heuristic locality swap
15:         if reduce.assigned_id ≠ reduce.host_id then
16:             p ← reduce.prob
17:             norm ← (p − 1/m) / (1 − 1/m) /10
18:             upper_bound ← (1 + norm) × max
19:             SWAP_TASKS(M, reduce, upper_bound)
20:         end if
21:     end for
    return M
22: end procedure
23: procedure SWAP_TASKS(M, reduce, upper_bound)
24:     Swap tasks between node host_id and node
    assigned_id
25:     of reduce without exceeding the upper_bound
26:     of both nodes.
    return
27: end procedure
```

3.2.3. 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 is 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.,

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

[7] http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html

high *prob*) after the accumulation, which leads to a higher probability of tasks swap in $SWAP\_TASKS$.

To traverse *reduce* for accumulating previous $shuffle$ and re-shuffling data, the algorithm needs $O(n)$ operations.

Algorithm 2 Accumulated Heuristic Scheduling for Multi-Shuffles

```
1:  procedure M_SCHEDULE(m, host_id, p_reduces, shuffles)
2:      m ← 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 ← shuffles[r.rid].size/r.size
6:          if new_prob ≥ r.prob then
7:              r.prob ← new_prob
8:              r.host_id ← shuffles[r.rid].assigned_host
9:          end if
10:     end for
11:     M ← SCHEDULE(m, host_id, p_reduces)
12:     for all host_id in M do                    ▷ Re-shuffle
13:         for all r in M[host_id].reduces do
14:             if host ≠ shuffles[r.rid].assigned_host then
15:                 Re-shuffle data to host
16:                 shuffles[r.rid].assigned_host ← host
17:             end if
18:         end for
19:     end forreturn M
20: end procedure
```

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 two constraints on memory management. In Subsection 4.3, we discuss the fault tolerance of the system.

4.1. System Overview

As shown in Figure 4, 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. The master node of SCache includes a DAG co-scheduler and a storage manager. The DAG co-scheduler is responsible for pre-scheduling the reduce tasks assignment and then forcing the DAG framework to assign reduce tasks according to this assignment. The storage manager is used for managing the shuffle blocks saved in worker nodes.

When a DAG job is submitted, the DAG information is generated in the framework task scheduler. If the RangePartitioner or a customized non-hash partitioner is used, SCache will insert an extra sampling job before the job starts. The sampling job uses a reservoir sampling algorithm [22] on each partition to gather the sample data. SCache master predicts the reduce partition size based on the data and then calls Algorithm 1 or 2 to do the pre-scheduling.

After a map task finishes computing, the shuffle writes in the map task are modified to use SCache API to memory copy all shuffle blocks to the SCache worker reserved memory. The SCache records the block ID and the size and sends them to the master. If the collected shuffle blocks reach a threshold, which is defined by the memory usage, the SCache co-scheduler will run the scheduling Algorithm 1 or 2 to pre-schedule the reduce tasks. After pre-scheduling, SCache workers start pre-fetching shuffle blocks according to the assignment from the master. To enforce the DAG framework to assign tasks according to the SCache pre-scheduled results, we also insert some lines of codes in the framework scheduler.

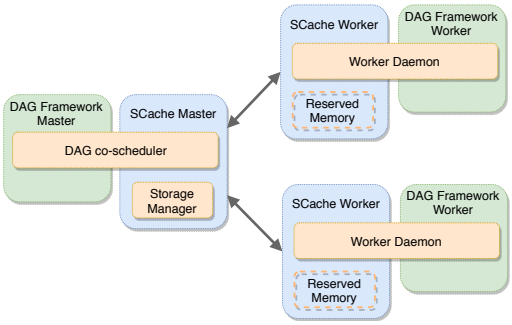


Figure 4: SCache Architecture

4.2. Memory Management

SCache uses memory management described below to ensure performance. If the reserved memory is running out, SCache will flush some of the blocks into the disk temporarily and then re-fetch them if needed. SCache leverages two constraints to manage the shuffle blocks: *all-or-nothing* and *context-aware-priority*.

4.2.1. All-or-Nothing Constraint

As discussed in the observation in Subsection 2.1, it is common to have multi-round execution of each stage. If one of the tasks in a round missed a memory cache, the re-fetch overhead will become a bottleneck and then slow down the whole stage. PACMan [10] 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, SCache master sets blocks of one round as the minimum unit of storage management. We refer to this as the all-or-nothing constraint.

4.2.2. 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. 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. If all the units are completed, SCache selects victims (i.e., tasks with lower priority) based on two factors: (a) For the tasks in the different stages, SCache sets the lower priority to storage units with an earlier submission time; (b) For the tasks in the same stage, SCache sets the

lower priority to storage units with a smaller task ID.

4.3. Discussion of Fault Tolerance

Due to the characteristic of shuffle data (e.g., short-lived, write-once, read-once), SCache only restarts failed workers without recovering their data and leaves the fault handling to the DAG frameworks. If a failure happens in a SCache worker during shuffle phases, the `getBlock` API will return a data-not-found error which causes the DAG framework restarts the current stage. During the re-computing, the DAG frameworks still use SCache to gain shuffle optimization.

Although the entire computing task has to be re-computed, we consider that there is a trade-off between the efficiency and the overhead of fault recovery with SCache. A possible way to avoid this re-computing is to use replications to recover the data. However, such replications can introduce a significant network overhead. We believe that the current way is more promising because most DAG frameworks have more advanced fast recovery schemes on the application layer, such as the paralleled recovery of Spark.

5. Framework Resources Quantification Model

In this section, we introduce *Framework Resources Quantification* (FRQ) model to evaluate the performance of DAG frameworks. The FRQ model is able to predict the execution time required by the application under most circumstances, including different DAG frameworks, hardware environments, etc. We first introduce the FRQ model in Subsection 5.1. In Subsection 5.2, we use the FRQ model to describe different computation jobs and discuss their performance.

5.1. The FRQ Model

We propose the FRQ model to better analyze the relationship between the computation phase and the shuffle phase in the DAG computing. The FRQ model focuses on describing the shuffle overhead which is significantly affected by the scheduling strategy. After quantifying computing and I/O resources, we use the FRQ model to evaluate different resource scheduling strategies.

As Figure 5 shows, the FRQ model divides the job into three phases: map, shuffle, and reduce. The horizontal axis represents time and the vertical axis represents different phases. Firstly, we introduce five input parameters of the FRQ model:

- 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.
- 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 (i.e., sufficient for the input data and the run-time memory of the frameworks), 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.
- Shuffle Speed ($V_{Shuffle}$): Transmission speed in the shuffle phase.

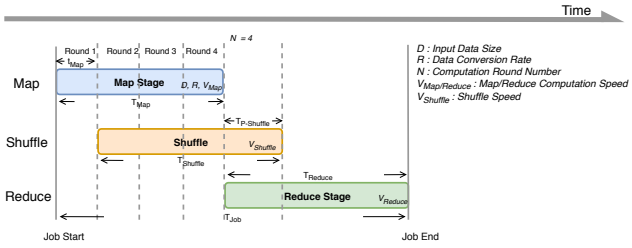


Figure 5: Framework Resources Quantification Model with Full Parallel MapReduce ($V_{Map} \times R > V_{Shuffle}$)

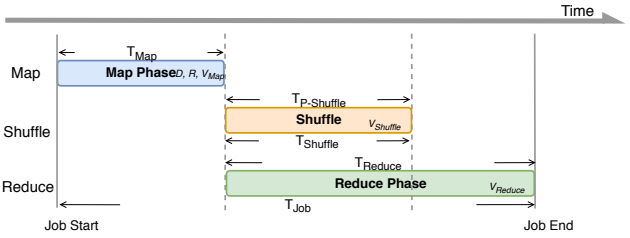


Figure 6: Framework Resources Quantification Model with Half Parallel MapReduce

Besides, the FRQ model also needs to input the scheduling strategies the framework takes. The FRQ model needs to know the start time of each phase. For example, as shown in Figure 6, this strategy starts the shuffle phase and the reduce phase at the same time. And Figure 5 shows the pre-fetching strategy used by SCache. In the pre-fetching strategy, the shuffle phase starts as soon as it can in the map phase.

The FRQ model calculates the execution time of each phase of the job with these five parameters. The total execution time of a job is the sum of the map phase time and reduce phase time:

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

The formulas of map phase time and shuffle phase time are as follows:

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

$T_{Penalty}$ is defined as the penalty time of fault recovery, which depends on the fault tolerance mechanism of the frameworks. With SCache, since the intermediate data is stored in memory, when fault occurs the entire computing task need to be re-computed from the beginning.

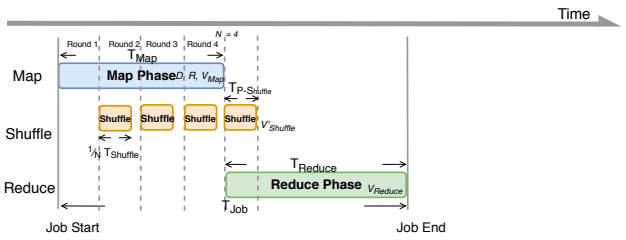


Figure 7: Framework Resources Quantification Model with Full Parallel MapReduce ($V_{Map} \times R < V_{Shuffle}$)

The reduce phase time formula is as follows:

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

$\frac{D \times R}{V_{Reduce}}$ represents the ideal computation time of a reduce phase, and $(K \times T_{P\_Shuffle})$ represents the computing overhead. $T_{P\_Shuffle}$ represents the overlap time between the shuffle phase and the reduce phase as shown in Figure 5. $T_{Shuffle}$ represents the total time of the shuffle phase. K is an empirical value which represents the overhead caused by shuffle waiting. Because the computation of the reduce phase relies on the results of data transferring of the shuffle phase, a portion of the computation needs to wait for the results of transferring. This waiting causes the overhead.

According to Equation 3 and 5, we can optimize the job completion time by reducing $T_{P\_Shuffle}$. By using different resource scheduling strategies, the formulas of $T_{P\_Shuffle}$ are different. In the next subsection, we show the formulas and discuss the performance of different strategies.

5.2. Different Scheduling Strategies on the FRQ Model

Figure 6 shows a 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 5). 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 3). The execution time of the shuffle phase is hidden in the reduce phase.

Figure 5 and Figure 7 show the scheduling strategy for Hadoop MapReduce with SCache (Suppose N is 4). SCache also overlaps the shuffle phase and the map phases by starting pre-fetching and pre-scheduling in the map phase. This scheduling strategy avoids 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 5)

   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:

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

2. $V_{Map} \times R < V_{Shuffle}$ (Figure 7)

   When the shuffle speed ($V_{Shuffle}$) is faster, SCache needs to wait for shuffle data to be generated. As Figure 7 shows, the shuffle phase will be interrupted in each round. In this case, the formula of $T_{P\_Shuffle}$ is as followed:

   $$T_{P\_Shuffle} = T_{Shuffle} \times \frac{1}{N} \quad (7)$$

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.

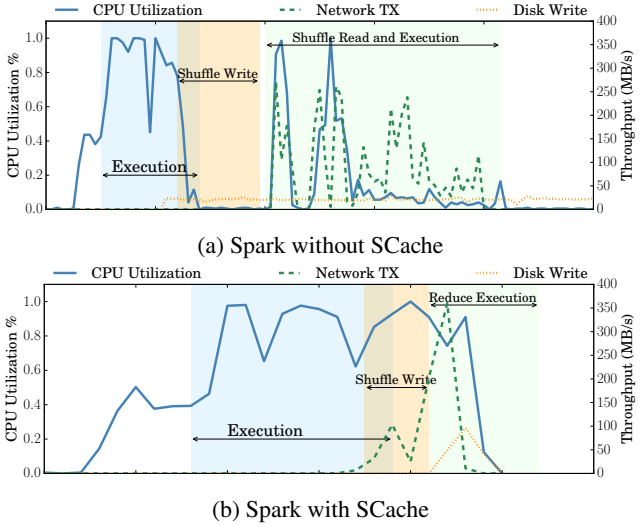


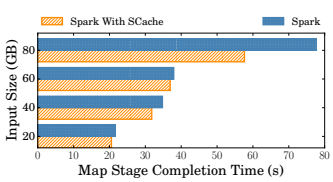
Figure 8: CPU Utilization and I/O Throughput of a Node During a Spark Single Shuffle Application

6. Evaluation

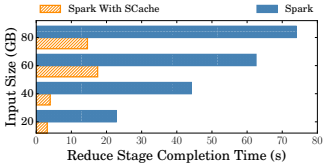
This section reveals a representative evaluation of SCache performance on both Spark and Hadoop MapReduce. In industrial big data, SQL-based big data technologies are widely used in these systems, such as data mining, predictive analytics, text analytics, and statistical analysis [23].

We first evaluate SCache with TPC-DS[8] to prove the performance gain of SCache in industry. TPC-DS is a standard benchmark which focuses on modeling industrial workload. Then we use a recognized shuffle intensive benchmark Terasort [24] to evaluate SCache with different data partition schemes. In summary, SCache decreases 89% time of Spark shuffle and improves 40% of the overall completion

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

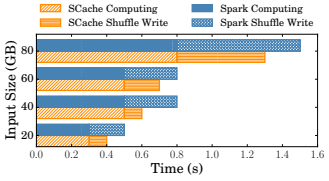


(a) Map Stage Completion Time

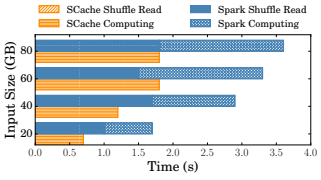


(b) Reduce Stage Completion Time

Figure 9: Stage Completion Time of Single Shuffle Test



(a) Median Task in Map Stages



(b) Median Task in Reduce Stages

Figure 10: Median Task Completion Time of Single Shuffle Test

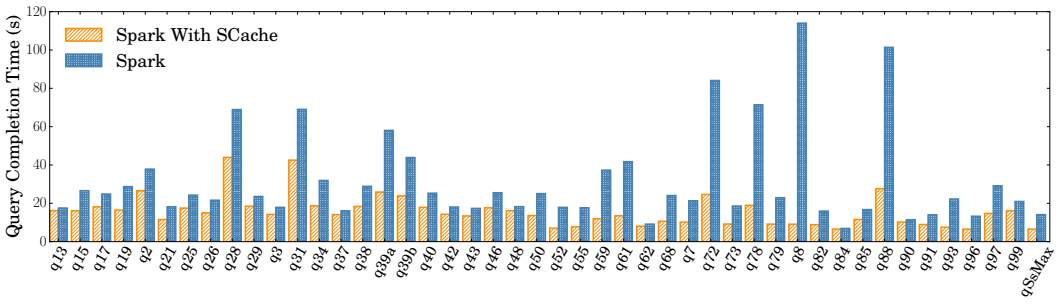


Figure 11: TPC-DS Benchmark Evaluation

time of TPC-DS on average. Meanwhile, Hadoop MapReduce with SCache optimizes job completion time by up to 15% and an average of 13%. We performed the experiments for about three to five times each variously, which ensured the accuracy of the evaluation.

6.1. Setup

We use Spark version 1.6.2 and Hadoop MapReduce version 2.8.5. The shuffle configuration of Spark is set to the default[9]. We run the experiments on a 50-node m4.xlarge cluster on Amazon EC2. Each node has 16GB memory and 4 CPUs. The network bandwidth of each node is about 300 Mbps.

6.2. Spark with SCache

6.2.1. Simple DAG Analysis

We first run Spark's *GroupByTest* on Amazon EC2. This job has 2 rounds of tasks for each node. As shown in Figure 8, the hardware utilization is captured from one node during the job. Note that since the completion time of the job with SCache is about 50% less than Spark without SCache, the duration of Figure 8b is only half of Figure 8a. As shown in 8a, the network transfer of shuffle data introduces an explicit I/O delay during *shuffle read*. And the several bursts of network traffic in *shuffle read* result in network congestion. As shown in Figure 8b, an overlap among CPU, disk, and network can be easily observed. 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.

In the map stage, the disk operations are replaced by the memory copies to decouple the shuffle write. It helps eliminate 40% of shuffle write time (Figure 10a), which leads to a 10% improvement of map stage completion time in Figure 9a. 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 [18].

In the reduce stage, most of the shuffle overhead is introduced by network transfer delay. By doing shuffle data pre-fetching, 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. As a result, the combination of these optimizations decreases 100% overhead of the shuffle read in a reduce task (Figure 10b). In addition, the heuristic algorithm can achieve a balanced pre-scheduling result, thus providing 80% improvement in reduce stage completion time (Figure 9b).

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

6.2.2. Industrial Production Workload

We also evaluate some queries from TPC-DS[10]. TPC-DS benchmark is designed for modeling multiple users submitting varied queries (e.g. ad-hoc, interactive OLAP, data mining, etc.). TPC-DS contains 99 queries and is considered as the standardized industry benchmark for testing big

[9] http://spark.apache.org/docs/1.6.2/configuration.html

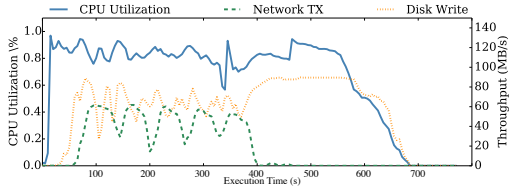
[10] http://www.tpc.org/tpcds/

	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	32 GB	1	4	0.65 GB/s	1 GB/s	0.47 GB/s	0.5	49.23	68.09 s	31.16 s	47.58 s	96.81 s	104 s	6.91%
	48 GB	1	6	0.65 GB/s	1 GB/s	0.47 GB/s	0.5	73.85	102.13 s	40.59 s	68.29 s	142.14 s	151 s	5.87%
	64 GB	1	8	0.65 GB/s	1 GB/s	0.47 GB/s	0.5	98.46	136.17 s	50.02 s	89.01 s	187.47 s	193 s	2.87%
Legacy	32 GB	1	4	0.65 GB/s	1 GB/s	0.47 GB/s	0.6	49.23 s	68.09 s	68.09 s	72.85 s	122.08 s	135 s	9.57%
	48 GB	1	6	0.65 GB/s	1 GB/s	0.47 GB/s	0.6	73.85 s	102.13 s	102.13 s	109.28 s	183.12 s	188 s	2.59%
	64 GB	1	8	0.65 GB/s	1 GB/s	0.47 GB/s	0.6	98.46 s	136.17 s	136.17 s	145.70 s	244.16 s	249 s	1.94%

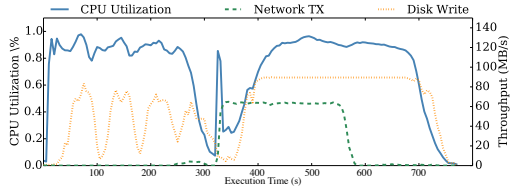
Table 1
Hadoop MapReduce on 4 nodes cluster in the FRQ model

	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 GB	1	5	1.15 GB/s	1.46 GB/s	1.4 GB/s	0.5	111.30	91.43	18.29	96.81	208.12	232	10.29%
	256 GB	1	5	1.15 GB/s	1.46 GB/s	1.4 GB/s	0.5	222.61	182.86	36.57	193.63	416.24	432	3.65%
	384 GB	1	5	1.15 GB/s	1.46 GB/s	1.4 GB/s	0.5	333.91	274.29	54.86	290.44	624.36	685	8.85%
Legacy	128 GB	1	5	1.15 GB/s	1.46 GB/s	1.4 GB/s	0.6	111.30 s	91.43 s	91.43 s	142.53 s	253.83 s	266 s	4.57%
	256 GB	1	5	1.15 GB/s	1.46 GB/s	1.4 GB/s	0.6	222.61 s	182.86 s	182.86 s	285.06 s	507.67 s	524 s	3.12%
	384 GB	1	5	1.15 GB/s	1.46 GB/s	1.4 GB/s	0.6	333.91 s	274.29 s	274.29 s	427.59 s	761.50 s	776 s	1.87%

Table 2
Hadoop MapReduce on 50 AWS m4.xlarge nodes cluster in the FRQ model



(a) Hadoop MapReduce with SCache



(b) Hadoop MapReduce without SCache

Figure 12: CPU utilization and I/O throughput of a node during a Hadoop MapReduce Terasort job

data systems. As shown in Figure 11, the horizontal axis is query name and the vertical axis is query completion time. Note that we skip some queries due to compatible issues. Spark with SCache outperforms the original Spark in almost all tested queries. Furthermore, in some shuffle-heavy 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.

6.3. Hadoop MapReduce with SCache

Unlike Spark, Hadoop MapReduce has only one Map and one Reduce phase. Such a simple workflow alleviates the defect of the naive scheduling schemes and reduces the benefits of *pre-scheduling*. However, SCache can still provide considerable optimization on shuffle-heavy workloads.

We use Terasort [24] benchmark to evaluate the performance gain of SCache and also prove the cross-framework ability of SCache. Terasort consists of two consecutive shuffles. The first shuffle reads the input data and uses a hash partition function for re-partitioning. The second shuffle partitions the data through a Spark `RangePartitioner`.

As shown in Figure 12b, Hadoop MapReduce without SCache starts shuffle and reduce simultaneously. The beginning part of reduce phase needs to wait for network transfer because a large amount of shuffle data reaches the network bottleneck. This causes the CPU resources to be idle. On the other hand, in Figure 12a, Hadoop MapReduce with SCache decouples shuffle from reduce and starts pre-fetching in the map phase. SCache utilizes the idle I/O throughput in the map phase. As shown in Figure 13, after decoupling, 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.

6.4. The FRQ Model Evaluation

To evaluate the FRQ model, we run Terasort with Hadoop MapReduce with two different environments. The parameters D, N, R, V_{Map}, $V_{Shuffle}$, and V_{Reduce} are set according to the application and the environment. We set K to 0.5 and 0.6 since SCache alleviates the defect on the reduce phase. Table 1 shows the results in the in-house environment. While enabling SCache, Terasort satisfies the situation in Figure 5 ($V_{Map} \times R \geq V_{Shuffle}$). While without SCache, $T_{P\_Shuffle}$ is equal to $T_{Shuffle}$(see Equation 4). $ExpT_{Job}$ represents the actual experiment job times and $Error$ represents the error between T_{Job} and $ExpT_{Job}$. Table 2 shows the results in Amazon EC2. In this environment, Terasort on Hadoop MapReduce satisfies the situation in Figure 7 ($V_{Map} \times R < V_{Shuffle}$), thus the formula of $T_{P\_Shuffle}$ is Equation 7.

As shown in Table 1 and 2, the experimental values are all larger than the calculated values. This is because the application has some extra overhead at runtime, such as net-

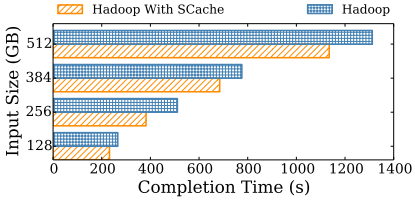


Figure 13: Hadoop MapReduce Terasort Completion Time

work 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.

7. Related Work

Modeling. Verma et al. [25] proposed the ARIA model to estimate the required resources based on the job information, the amount of input data and a specified soft deadline. Khan et al. [26] proposed the HP model which extends the ARIA model. The HP model adds scaling factors and uses a simple linear regression to estimate the job execution time on larger datasets. In [27], Herodotou proposed a detailed set of mathematical performance models for describing the five phases of a MapReduce job and combine them into an overall MapReduce job model. Chen et al. [28] proposed the CRESP model which is a cost model that estimates the performance of a job then provisions the resources for the job. Farshid Farhat et al.[29] proposed a closed-form queuing model which focuses on stragglers and tries to optimize them. However, the above models are not able to accurately describe the overhead caused by the shuffle process under different scheduling strategies.

Industrial big data. 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 [30]. Nowadays, the real challenge of big data is not how to collect it, but how to manage it logically and efficiently [31]. Several distributed computing frameworks become efficient tools in industrial big data analysis [1, 32, 33].

DAG Optimization. Slow-start from Hadoop MapReduce is a classic approach to handle shuffle overhead. Starfish [34] gets sampled data statistics for self-tuning system parameters (e.g. slow-start, etc). DynMR [35] dynamically starts reduce tasks in late map stage. iShuffle [21] 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 [36] aggressively pre-schedules tasks in multiple successive stages to start fetching shuffle. In data skew mitigation, Skewtune [37] partitions the data dynamically. When a straggler task happens, it re-partitions the data and forces them to transfer to other nodes. LIBRA [38] proposes a new sampling method for DAG frameworks and uses an innovative approach, which supports the split of large keys, to balance the load among the reduce tasks. ToF [39] proposed a DAG optimization system to address the performance and monetary cost optimizations in the cloud. Different from these works, SCache pre-schedules multiple shuffles and uses pre-fetching to gain optimization without breaking load balancing.

Network layer optimization. Varys [40] and Aalo [41] 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. According to [42, 43], we can also combine SDN with SCache to further improve the performance on the network later.

8. Conclusion

In this paper, we present SCache, a plug-in shuffle optimization for DAG computing frameworks. By task pre-scheduling and shuffle data pre-fetching with application context, SCache decouples the shuffle from computing tasks and mitigates the shuffle overhead. We also propose *Framework Resources Quantification* (FRQ) model to evaluate the shuffle process of DAG computing frameworks. The evaluations show that SCache can provide a promising speedup in industrial production workloads.

Acknowledgements

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] P. Lade, R. Ghosh, S. Srinivasan, Manufacturing analytics and industrial internet of things, IEEE Intelligent Systems 32 (2017) 74–79.

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

[3] Y. Chen, S. Alspaugh, R. Katz, Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads, Proceedings of the VLDB Endowment 5 (2012) 1802–1813.

[4] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, I. Stoica, Managing data transfers in computer clusters with orchestra, in: ACM SIGCOMM Computer Communication Review, volume 41, ACM, 2011.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: ACM SIGOPS operating systems review, volume 41, ACM, 2007.

[6] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, K. Li, A parallel random forest algorithm for big data in a spark cloud computing environment, IEEE Transactions on Parallel and Distributed Systems 28 (2017) 919–933.

[7] C. Xie, R. Chen, H. Guan, B. Zang, 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, PPoPP 2015, ACM, New York, NY, USA, 2015. doi:10.1145/2688500.2688508.

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

[9] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, I. Stoica, Tachyon: Reliable, memory speed storage for cluster computing frameworks, in: Proceedings of the ACM Symposium on Cloud Computing, 2014.

[10] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, 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.

[11] B. Heintz, A. Chandra, R. K. Sitaraman, J. Weissman, End-to-end optimization for geo-distributed mapreduce, IEEE Transactions on Cloud Computing 4 (2016) 293–306.

[12] D. Cheng, J. Rao, Y. Guo, C. Jiang, X. Zhou, Improving performance of heterogeneous mapreduce clusters with adaptive task tuning, IEEE Transactions on Parallel and Distributed Systems 28 (2017) 774–786.

[13] M. Wasi-ur Rahman, N. S. Islam, X. Lu, 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 28 (2017) 633–646.

[14] Z. Fu, T. Song, Z. Qi, H. Guan, Efficient shuffle management with scache for dag computing frameworks, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2018, pp. 305–316.

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

[16] 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, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (2016) 56–65.

[17] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, 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.

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

[19] B. Fitzpatrick, Distributed caching with memcached, Linux J. 2004 (2004) 5–.

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

[21] Y. Guo, J. Rao, D. Cheng, X. Zhou, ishuffle: Improving hadoop performance with shuffle-on-write, IEEE Transactions on Parallel and Distributed Systems 28 (2017) 1649–1662.

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

[23] M. Poess, T. Rabl, H.-A. Jacobsen, Analysis of tpc-ds: the first standard benchmark for sql-based big data systems, in: Proceedings of the 2017 Symposium on Cloud Computing, ACM, 2017.

[24] E. Higgs, A. Trivedi, J. Zhang, Spark terasort, 2017. URL: https://github.com/ehiggs/spark-terasort.

[25] A. Verma, L. Cherkasova, 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.

[26] M. Khan, Y. Jin, M. Li, Y. Xiang, C. Jiang, Hadoop performance modeling for job estimation and resource provisioning, IEEE Transactions on Parallel & Distributed Systems (2016) 441–454.

[27] H. Herodotou, Hadoop performance models, arXiv preprint arXiv:1106.0940 (2011).

[28] K. Chen, J. Powers, S. Guo, F. Tian, Cresp: Towards optimal resource provisioning for mapreduce computing in public clouds, IEEE Transactions on Parallel and Distributed Systems 25 (2014) 1403–1412.

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

[30] S. Yin, O. Kaynak, Big data for modern industry: challenges and trends [point of view], Proceedings of the IEEE 103 (2015) 143–146.

[31] Z. Lv, H. Song, P. Basanta-Val, A. Steed, M. Jo, Next-generation big data analytics: State of the art, challenges, and future research topics, IEEE Transactions on Industrial Informatics 13 (2017) 1891–1899.

[32] X. Li, J. Song, 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 84 (2016) 119–131.

[33] M. H. ur Rehman, E. Ahmed, I. Yaqoob, I. A. T. Hashem, M. Imran, S. Ahmad, Big data analytics in industrial iot using a concentric computing model, IEEE Communications Magazine 56 (2018) 37–43.

[34] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, S. Babu, Starfish: A self-tuning system for big data analytics., in: Cidr, volume 11, 2011.

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

[36] E. Elnikety, T. Elsayed, H. E. Ramadan, ihadoop: asynchronous iterations for mapreduce, in: Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on, 2011.

[37] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune: mitigating skew in mapreduce applications, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012.

[38] Q. Chen, J. Yao, Z. Xiao, Libra: Lightweight data skew mitigation in mapreduce, IEEE Transactions on parallel and distributed systems 26 (2014).

[39] A. C. Zhou, B. He, Transformation-based monetary costoptimizations for workflows in the cloud, IEEE Transactions on Cloud Computing 2 (2014) 85–98.

[40] M. Chowdhury, Y. Zhong, I. Stoica, Efficient coflow scheduling with varys, in: ACM SIGCOMM Computer Communication Review, volume 44, ACM, 2014, pp. 443–454.

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

[42] P. Qin, B. Dai, B. Huang, G. Xu, Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data, IEEE Systems Journal 11 (2017) 2337–2344.

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