PISCES: Optimizing Multi-Job Application Execution in MapReduce

Qi Chen, Jinyu Yao, Benchao Li, and Zhen Xiao, *Senior Member, IEEE*

Abstract—Nowadays, many MapReduce applications consist of groups of jobs with dependencies among each other, such as iterative machine learning applications and large database queries. Unfortunately, the MapReduce framework is not optimized for these multi-job applications. It does not explore the execution overlapping opportunities among jobs and can only schedule jobs independently. These issues significantly inflate the application execution time. This paper presents Pipeline Improvement Support with Critical chain Estimation Scheduling (PISCES), a critical chain optimization (a critical chain refers to a series of jobs which will make the application run longer if any one of them is delayed), to provide better support for multi-job applications. PISCES extends the existing MapReduce framework to allow scheduling for multiple jobs with dependencies by dynamically building up a job dependency DAG for current running jobs according to their input and output directories. Then using the dependency DAG, it provides an innovative mechanism to facilitate the data pipelining between the output phase (map phase in the Map-Only job or reduce phase in the Map-Reduce job) of an upstream job and the map phase of a downstream job. This offers a new execution overlapping between dependent jobs in MapReduce which effectively reduces the application runtime. Moreover, PISCES proposes a novel critical chain job scheduling model based on the accurate critical chain estimation. Experiments show that PISCES can increase the degree of system parallelism by up to 68 percent and improve the execution speed of applications by up to 52 percent.

Index Terms—MapReduce, job dependency, group scheduling, pipeline

1 Introduction

We are experiencing an age of big data. Internet applications such as search engines, social networks and mobile applications bring us not only high quality of service but also challenges of efficient large-scale data processing. MapReduce[1], a popular distributed parallel computing framework with efficiency, convenience and fault tolerance, has already been widely used in web indexing, log analysis, data warehousing, data mining, scientific computing, and other widespread applications.

Many of these applications in MapReduce are written as groups of jobs with dependencies among each other. Two common types of multi-job applications are iterative machine learning applications and large database queries. Iterative machine learning applications, such as the PageRank algorithm [2], break their computation into multiple iterations with each iteration being run as a MapReduce job. This generates a series of separated jobs which are executed one after another. Large database queries, like Pig or Hive scripts, are translated into a DAG in which each node represents a MapReduce job.

Despite the popularity of these multi-job applications, the existing MapReduce system is not optimized for them. First of all, it does not explore the execution overlapping opportunities between dependent jobs. It requires the jobs to have their input data ready before submission, which requires a job to wait for the completion of all its dependent jobs and creates a synchronization barrier between dependent jobs. MapReduce Online [3] and HPMR [4] provide execution overlapping between the map and the reduce stages of a job. MapReduce itself provides execution overlapping between independent jobs. However, none of previous work focuses on how to break the synchronization barrier between dependent jobs.

In addition, MapReduce can only schedule jobs independently. MapReduce itself does not maintain any dependency information among jobs. Although there has already been a great deal of work providing high quality job schedulers [5], [6], [7], [8], they only focus on how to schedule collections of independent jobs. Therefore, users need to arrange the submissions of the dependent jobs in the right order themselves and submit them one by one. There are also several pieces of software which provide job dependency analysis and scheduling at a higher level, such as Pig [9] for Apache Hadoop [10], Scope [11] for Microsoft Dryad [12], Tenzing [13] for Google MapReduce [1], Hive [14] and RoPE [15]. However, they only focus on how to translate a query to a good job dependency DAG and simply schedule jobs according to the dependency constraints. They cannot make effective job scheduling decisions since they lack some important information in MapReduce, such as the input data size, output data size and execution time of each task, the number of free slots in the system, and the performance of each worker, etc. Moreover, since these higher level softwares are independent of each other, they have to implement their own job scheduling strategy and cannot support job scheduling across different

- *The authors are with the Department of Computer Science, Peking University, Beijing 100871, China.*
  E-mail: {chenqi, yjy, lbc, xiaozhen}@net.pku.edu.cn.

Manuscript received 28 May 2015; revised 31 May 2016; accepted 12 Aug. 2016. Date of publication 26 Aug. 2016; date of current version 6 Mar. 2019.
Recommended for acceptance by P.B Gibbons.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2016.2603509

2168-7161 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

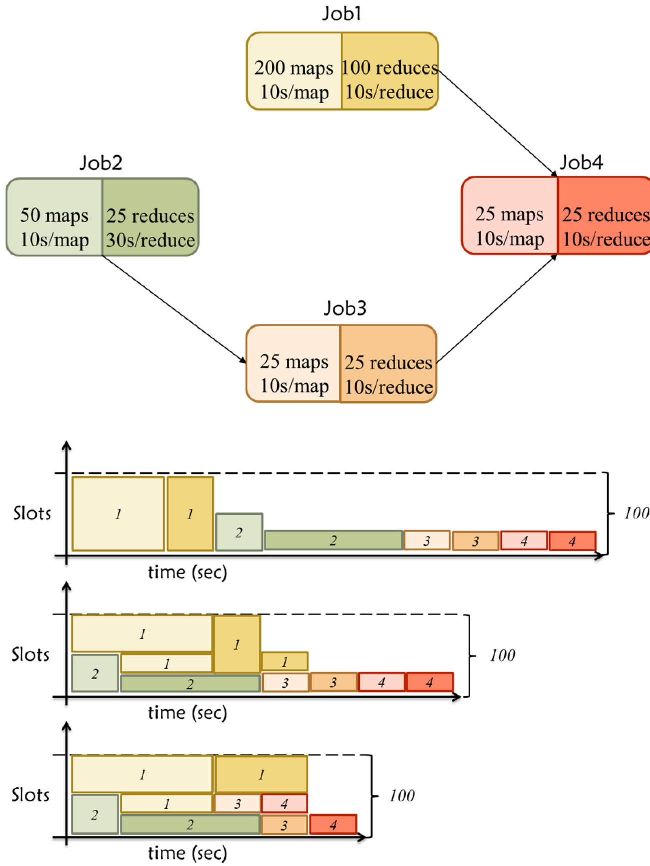


Fig. 1. The impact of job scheduling.

softwares. The closest previous work is FlowFlex [16] which provides a theoretical malleable scheduling for flows of MapReduce jobs. However, the fixed amount of work unit for each job in its model is hard to measure in the real computing environment. Moreover, it is hard to add the new overlapping feature of dependent jobs into its theoretical scheduling model.

We notice that an efficient job scheduling policy can have a significant impact on the execution time of multi-job applications. This is illustrated in a simplified example in Fig. 1 for a system with four jobs. The number of map tasks, reduce tasks and their required execution times are as shown in the figure. We can model the dependencies among the jobs as a DAG. Suppose the MapReduce cluster is deployed in a group of 50 workers and each worker is configured with two slots (minimal resource unit required by a task). If job 1 is executed first, it will occupy the entire cluster and all four jobs end up executing one after another (the first schedule in Fig. 1). It will take 110 seconds to finish the group of jobs. On the contrary, if job 2 is scheduled first, it takes up only half of slots. The remaining slots can be used to execute part of job 1 in parallel (the middle schedule in Fig. 1). In this case, the group of jobs can be finished in 80 seconds, a 37.5 percent improvement. If we further allow data pipelining between the output phase of an upstream job and the map phase of a downstream job and assume that the map phase of the downstream job can be finished soon after the completion of the output phase of the upstream job (the last schedule in Fig. 1), the group of jobs can be finished even earlier in about 60 seconds, a further 33.3 percent improvement.

It turns out that maximizing the possible execution overlapping among dependent jobs and giving a better job scheduling for those multi-job applications remain to be tough challenges for us. In this paper, we present a new critical chain optimization in MapReduce called Pipeline Improvement Support with Critical chain Estimation Scheduling (PISCES) to address these challenges. To the best of our knowledge, we are the first to propose this optimization. Compared to the previous work, we make the following contributions:

- We extend the existing MapReduce framework to allow scheduling for multiple jobs with dependencies among each other by building up a dynamic job dependency DAG for current running jobs.
- We creatively parallelize the output phase (map phase in the Map-Only job or reduce phase in the Map-Reduce job) of an upstream job and the map phase of a downstream job to offer a new execution overlapping between dependent jobs which effectively reduces the application runtime.
- We accurately predict the execution time for each job by using a new locally weighted linear regression (LWLR) model [17] based on the job execution histories. The prediction precision will reach above 80 percent in general.
- We build a novel "overlapping" critical chain job scheduling model based on the accurate critical chain estimation which further improve the performance of the system.

Experiment results show that our PISCES system can increase the degree of parallelism by up to 68 percent and run applications up to 52 percent faster.

The rest of the paper is organized as follows. Section 2 provides a background on MapReduce, and its data pipelining and job scheduling support. Section 3 describes the implementation of our PISCES system, the innovative pipeline optimization and the novel critical chain job scheduling algorithm. Section 4 presents a performance evaluation. Related work is described in Section 5. Section 6 concludes.

2 Background

2.1 MapReduce Overview

A typical MapReduce cluster consists of one master node and several worker nodes. The master node is responsible for receiving jobs, scheduling tasks, and managing all the worker nodes. A worker node executes the map and the reduce tasks issued by the master. A specific job execution procedure is as follows:

- A user uploads input data into a distributed file system (e.g., GFS [18]), and submits a job to the MapReduce framework.
- The MapReduce client divides the input data into multiple splits (each split is 64 MB by default), generates the split info (including the storage location, the start position and the real size of each split), and submits the job and split info to the master.
- The master generates multiple map tasks according to the split info (one map task for each split), and then schedules them to different worker nodes for parallel processing.
- Each map task converts input (k_1, v_1) pairs into intermediate (k_2, v_2) pairs according to the user defined

TABLE 1
Example of a Pig Script

T1 =	LOAD "visits.txt" AS (user, url, date);
T2 =	LOAD "pages.txt" AS (url, pagerank);
T3 =	FILTER T1 BY date = "2013/07/20";
T4 =	FILTER T2 BY pagerank $>=$ 0.5;
T5 =	JOIN T3 BY url, T4 BY url;
T6 =	GROUP T5 BY user;

map and *combine* functions, partitions them into multiple parts by their keys according to a user defined partitioner, and stores them onto a local disk.

- When the percentage of finished map tasks reaches a certain threshold, the master begins to issue reduce tasks.
- Each reduce task copies its input parts from each map task, sorts them into a single stream according to their keys by a multi-way merge after all map tasks complete, transforms the intermediate result into the final (k_3, v_3) pairs according to the user defined reduce function, and finally outputs them into the user specified directory in the distributed file system.

In the steps above, the master does not make job execution plans. It simply arranges jobs in queues according to the order of their submissions. It is the responsibility of the user to submit the jobs in the right order. There are several prior approaches focused on extending the MapReduce framework to support complex queries. They support easy-to-use query languages (e.g., SQL-like language) to simplify the programming of MapReduce. Complex queries written in the SQL language are analyzed and decomposed automatically to generate a job execution plan for MapReduce. Table 1 gives an example of a Pig script which consists of multiple Pig queries. The job execution plan made by Pig is shown in Fig. 2. From the figure, we can see that Pig divides these queries into four MapReduce jobs that have dependencies with each other.

Moreover, jobs running in the MapReduce framework must have their input data ready before the submission. This is because the MapReduce client will divide their input data into splits as soon as they are submitted in order to determine the number of map tasks needed before the jobs start running. Therefore, jobs whose input are not ready must wait until all of their dependent jobs complete.

2.2 Challenges for Multi-Job Applications

As introduced before, there are two challenges left for us: i) How to maximize the possible execution overlapping among dependent jobs, ii) How to provide better job scheduling for multi-job applications.

For the first challenge, the most difficult problem is that the downstream job needs to consume the output data of all its upstream jobs. Therefore, we cannot run the downstream job before all of its input data are produced. MapReduce Online [3] and HPMR [4] provide intermediate data pipeline support between the map and the reduce stages within a job, which creatively breaks the synchronization barrier between the map and the reduce stages of a job. Can we take this idea to the dependent jobs? Before we answer this question, we need to face the following challenges: i) Where are the output data of a job before its completion? ii) How to decide which jobs these data should be pipelined to?

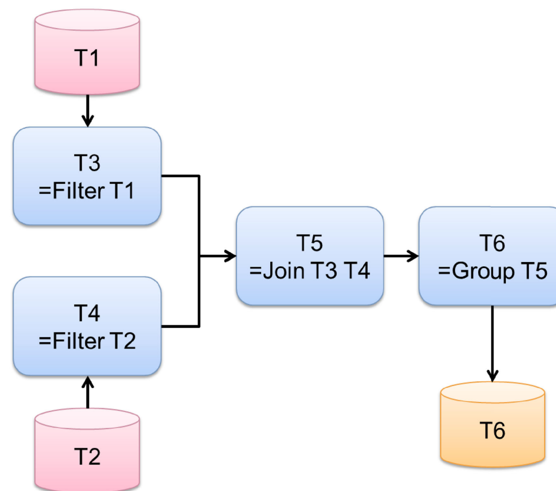


Fig. 2. Example of the Pig execution plan.

iii) How to make the downstream job run with dynamically growing input data? iv) How to ensure the execution correctness of the downstream job? v) How to provide fault tolerance for this pipeline mechanism?

For the second challenge, the difficulties are: i) How to build up a dynamic job dependency DAG for the current running jobs? ii) How to accurately predict the execution time for each task within a job so that we can find the critical chain correctly? iii) How to create a critical chain job scheduling model so that we can take advantage of the innovative job pipeline mechanism and finish the query as quickly as possible?

In the following, we show how PISCES can solve these challenges nicely.

3 The PISCES System

In this section, we present a new system called PISCES, a critical chain optimization, to address the challenges for multi-job applications. PISCES provides an innovative job pipeline mechanism and a new job scheduling strategy for data-intensive jobs that have dependency with each other. In the rest of this section, we will introduce PISCES system in detail.

3.1 System Overview

The MapReduce framework in which we choose to implement PISCES is the latest stable Hadoop-2.7.1 version. We improved the current Hadoop design to provide complex job workflow scheduling for the multi-job applications. We labeled the jobs of an application with the same workflow ID, submitted them concurrently and dispatched them to the same application master for scheduling. In the application master, we extended the architecture by adding following three new modules (shown in Fig. 3):

- Dependency Analyzer: analyze the dependency among a group of jobs according to their input and output directories.
- Job Time Estimator: estimate the execution time for each job according to the job configuration and the job execution history.
- Job Scheduler: make scheduling decisions for the submitted jobs according to their dependency DAG and estimated job execution time.

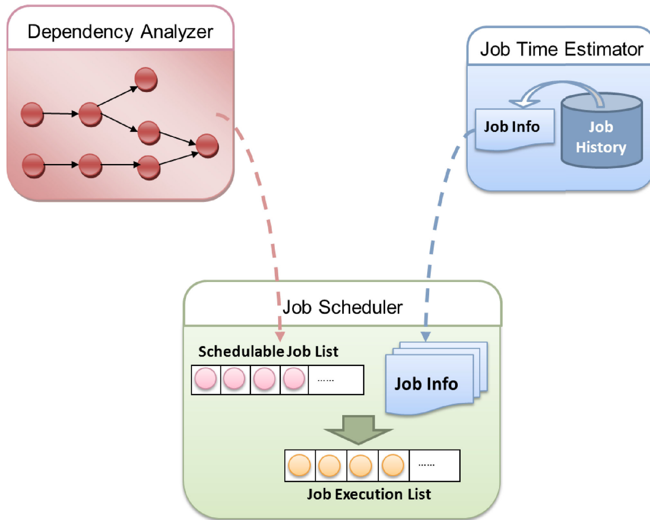


Fig. 3. System architecture.

3.2 Dependency Management and Data Pipelining

3.2.1 Dependency Management

Traditional job dependency analysis in higher level software like Pig [9], Scope [11] and Tenzing [13] usually contains the following two steps: i) translate a query into an internal parse tree, ii) convert the tree to a physical execution plan by traversing the tree in a bottom-up fashion. In PISCES, this procedure is much easier: i) A user submits a group of jobs with the same workflow ID in a parallel mode, ii) PISCES dispatches these jobs to the same application master, iii) PISCES builds up a dynamic job dependency DAG according to their input and output directories and maintains a schedulable job list which contains all the jobs whose whole input data are ready (the leaf jobs in the job dependency DAG) in the application master. Each time new jobs arrive or current running jobs complete, the dependency analyzer will update the job dependency DAG.

Note that there is a special case that we should also take into consideration when building the job dependency DAG: the user code directly accesses the HDFS through the DFSClient during the job execution. For example, in the KMeans clustering application, the input data for map tasks are the whole points set. However, each map task in each iteration should also read the whole cluster centers generated by the previous iteration to decide the current nearest cluster to which each point belongs. Therefore, the files storing the cluster centers cannot simply be added as the input of the MapReduce job. Instead, it should be read from the HDFS directly by each map task. In this case, there also exists a job dependency relationship between the previous iteration job and the current iteration job, although they do not have an input-output dependency. Therefore, we should also add a dependency edge between these two jobs. This can be done easily by comparing all the HDFS read paths of one job with the output paths of the other jobs. We call this kind of job dependency relationship "hard dependency".

3.2.2 Data Pipelining Among Jobs

In the current MapReduce framework, there exists a synchronization barrier between the upstream and the downstream jobs. This is because MapReduce itself does not maintain the job dependency DAG and cannot decide the downstream jobs to which a data block from an upstream job should be pipelined. Consequently, a job cannot be issued until all of its input data are ready, which means that a job needs to wait for the completion of all its dependent jobs.

Inspired by the intermediate data pipeline mechanism between the map and the reduce stages within a job provided by MapReduce Online [3] and HPMR [4], we wondered whether we could provide the same pipeline mechanism between dependent jobs. Since a map task in the downstream job only needs to deal with a continuous data block with a fixed size and all the map tasks within a job are independent with each other, the corresponding map task can be issued as soon as its input data block is produced by the upstream job. Therefore, it seems feasible in theory to provide the data pipeline mechanism between dependent jobs. But we first need to answer the five questions proposed in the Section 2.2.

i) Where are the Output Data of a Job Before Its Completion?

We notice that in Apache Hadoop [10], in order to support the roll back of failed jobs and tasks, the output of a reduce task is first written to a temporary directory in HDFS. It will be moved to the real destination directory when the task completes successfully. Moreover, as soon as a data block (default is 64 MB in Hadoop) is produced by a reduce task, it will be flushed to the HDFS. Therefore, we can read a data block from the temporary output file of a reduce task as soon as it is produced.

ii) How can we Decide the Jobs to Which These Data Should be Pipelined?

Since we have maintained a dynamic job dependency DAG for all the submitted jobs by using our dependency analyzer, we can simply pipeline a data block from the upstream job to all its downstream jobs according to the dependency relationships in the DAG (excluding the "hard dependencies").

iii) How to Make the Downstream Job Run with Dynamically Growing Input Data?

This is the most important question in the pipeline mechanism. We modify the Hadoop framework to allow a job to generate new map tasks dynamically during its execution as follows:

- For those submitted jobs which are not in the schedulable job list, we initialize the number of maps that each reduce task need to wait for to the maximum integer value so that we can limit all the reduce tasks to the shuffle phase. The purpose of this step is to ensure the correctness because the input data of these jobs has not been produced yet. It will not introduce performance penalty.
- When PISCES detects that a data block from an upstream job has been generated, it will notify all of its downstream jobs (excluding the "hard dependent" downstream jobs) to create a new map task. The notification includes the file name to read, the start offset of this block and the end offset of this block. Instead of only obtaining split information from the MapReduce client, PISCES can add extra split information dynamically during a job's lifetime.

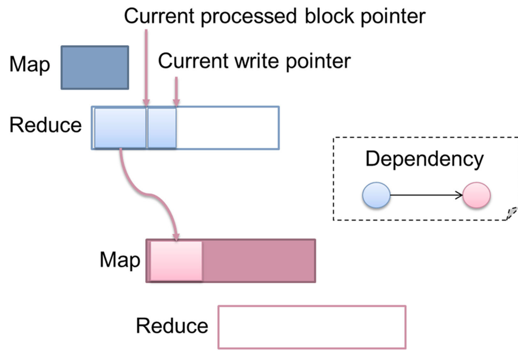


Fig. 4. Pipeline between an upstream job and a downstream job.

- In the dependency analyzer module, instead of waiting for all of its upstream jobs to complete, we add a job to the schedulable job list as soon as all of its upstream jobs enter the output phase (map phase in the Map-Only job or reduce phase in the MapReduce job) and have generated a certain amount of data (at least a data block). Note that if a job has "hard dependent" upstream jobs, we can only add it to the schedulable job list after all of its "hard dependent" upstream jobs complete and all of its other upstream jobs enter the output phase.
- After the completion of all its upstream jobs, PISCES notifies all the reduce tasks of a downstream job the correct number of maps. These reduce tasks can then enter the sort and the reduce phases for execution after copying all the intermediate data.

Fig. 4 illustrates this pipeline process. As we will see later in the experiments, the pipelining optimization improves the execution time of data intensive jobs significantly, especially for iterative data processing applications like PageRank.

iv) How to Ensure the Execution Correctness of the Downstream Job?

This can be separated into two parts: map correctness and reduce correctness. To ensure map correctness, we should ensure we will not miss any map inputs or add different map tasks that process the same input. To ensure this, we keep a current processed block pointer for each output task in the upstream jobs (shown in Fig. 4). Each time we increase the pointer, we will add a map task for all the downstream jobs. The pointer will never be decreased so that we can ensure each map task in a downstream job processes a different data block. Since each time the pointer will only move at most a data block length, we can ensure all the data blocks have been processed. To ensure reduce correctness, we first limit all the reduce tasks in the downstream jobs to the shuffle phase. They cannot enter the sort and the reduce phases until they receive the correct number of map outputs.

For those downstream jobs which have "hard dependent" upstream jobs, we will not pipeline the data blocks between their "hard dependent" upstream jobs and them since the whole outputs of their "hard dependent" upstream jobs may be read by each map task in these downstream jobs. Instead, we wait for the completion of all of their "hard dependent" upstream jobs. We will not add these downstream jobs to the schedulable job list until all of their "hard dependent" upstream jobs complete and all of their other upstream jobs enter the output phase. It is also reasonable to ignore these "hard dependencies" when considering pipeline between dependent jobs since this kind of dependent data are nearly always small and therefore will not dramatically impact the job execution time.

v) How to Provide Fault Tolerance for this Pipeline Mechanism?

With our new job pipeline mechanism, the map tasks in a downstream job will first read the output data from the temporary directory of an upstream job. Those map tasks and the re-executed map tasks (e.g., failed or slow tasks) will experience read errors when the temporary data are moved to their final destinations later. To tackle this problem, we implement a new hard link feature into HDFS to allow multiple inodes (metadata structure in file systems which stores all the information about a file object, such as ownership, creation time, data block locations, etc.) to point to the same data blocks. When the reduce task finishes, we create a hard link file in the destination directory to point to the file in the temporary directory. The basic semantics of a hard link is that multiple files (including the original file and the linked files) share the same data block information. The shared data blocks will not be deleted if there are any files referencing them. By doing so, we can avoid the read errors in the map tasks of the downstream jobs caused by the migration of the temporary data. The temporary data of a job will be finally deleted when all of its downstream jobs complete. Moreover, when an output task in the upstream job fails, the temporary output data of this task will not be deleted at once. Instead, they will be kept until the temporary directory of the job is removed. We will not generate new map tasks for the downstream jobs until the current write point of the new reduce task has caught up with the current processed block point and assign new map tasks using the new temporary output data.

To support hard links in HDFS, we add a series of hard link interfaces which include create, update, delete, and du operations. We give the hard link files the same permission as the source file. In addition, we also modify the FSImage save and restore operations to add the hard link information into the file system metadata image so that the HDFS can restore from failures successfully. The implementation details are outside the scope of this paper.

After solving the five questions above, we successfully parallelize the output phase of an upstream job and the map phase of a downstream job to offer new execution overlapping between dependent jobs. This job pipeline mechanism can take full advantage of not only the computing resources in the cluster but also the file cache of the operating system by consuming the output data as soon as it is produced. The reason why we still force the output data to be written to the HDFS is that redoing previous jobs is very expensive compared to just redoing some map tasks in the inter-job data pipeline mechanism. In order to provide fault tolerance and high data availability at a low cost, we write each data block to the HDFS before pipelining it to the downstream jobs.

3.3 Job Scheduling in PISCES

With our innovative job pipeline mechanism between dependent jobs, the job scheduling in MapReduce can be further improved by taking data flow information and the pipeline feature into consideration. Instead of scheduling jobs in the higher level softwares, such as Pig, we implement our

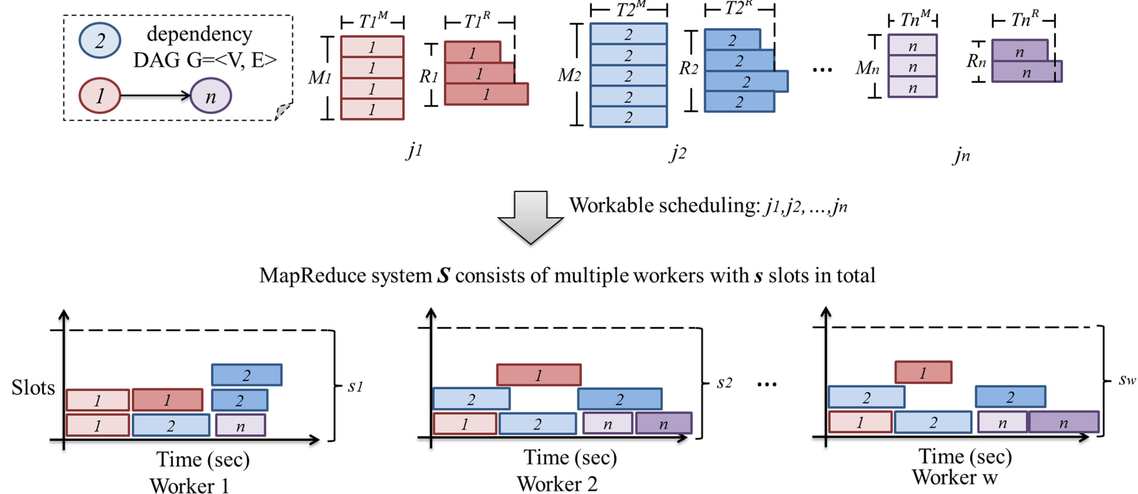


Fig. 5. Scheduling problem definition.

PISCES job scheduling directly into the MapReduce framework. The reason is that we can get some detailed job execution information easily and take advantage of the innovative job pipeline mechanism among dependent jobs. Such information is difficult to obtain at the higher level softwares. Moreover, this architecture allows our job scheduling method to be widely used in a variety of softwares, such as Pig, Cloud9 [19], Mahout [20] and even user defined job groups. Users can define their applications across different softwares without worrying about how to arrange them. Job scheduling in PISCES consists of the following steps:

- A user submits a group of jobs with the same workflow ID into PISCES. PISCES modifies the interface of job submission to allow jobs to be submitted in parallel.
- PISCES dispatches these jobs to the same application master. After receiving the submission of the jobs, the application master updates the job dependency DAG and estimates the job execution time for these jobs.
- Whenever there are enough free resources in the system and the current job running queue is empty, the job scheduler will choose a new job for running using a critical chain model described later.

3.3.1 Scheduling Problem Definition

We first formalize our job scheduling problem as follows. Suppose we have n jobs to be scheduled. Job i contains M_i map tasks and R_i reduce tasks. Its average execution time is T_i^M for a map task and T_i^R for a reduce task. We define a DAG $G = <V, E>$, where each vertex in V represents a job ($|V| = n$). The directed edge $<u, v> \in E$ if and only if job v is dependent on the output of job u. The MapReduce system S consists of multiple workers with s slots (minimal resource unit required by a task) in total. A workable scheduling scheme for job dependency graph G can be expressed as a list of job vertexes: $j_1, j_2, \ldots, j_n$, which satisfies the dependency property. The definition is shown in Fig. 5. The goal of our job scheduling is to provide an execution order of jobs to maximize the parallelism of the system and minimize the total execution time for all jobs.

The optimization of job scheduling is a NP hard problem no matter what policy is used to schedule tasks within a job. In the simplest case where each job contains only one map task and there is no dependency among the jobs, the job scheduling problem is equivalent to the job shop scheduling problem in the literature which has been shown to be NP-complete [21]. Moreover, new jobs with the same workflow ID can be submitted dynamically, which will cause the job dependency DAG changing all the time. Meanwhile, unexpected events can happen in real systems, for example, the straggler tasks[1] caused by system failures or resource competition among the tasks. If we allow the speculative execution to backup those straggler tasks, it will affect the optimal solution. Therefore, we simplify the job scheduling problem in the following two ways:

- We separate job scheduling from task scheduling within a job and only make decision on the job execution order.
- Instead of calculating the optimal solution at the beginning and then schedule jobs accordingly, we use an iterative method to get the job scheduling list so that our model can meet the requirements of dynamic job submissions and tolerate the uncertainty during real execution. We invoke our scheduling algorithm whenever there are free slots in the system and the current job running queue is empty. During each iteration, we sort all the schedulable pending jobs and choose the most appropriate one to join the running job list.

3.3.2 Estimate Job Execution Time

After the submission of a group of jobs, we first need to estimate the runtime for each job in order to decide how to schedule them. Although there has already been some work focusing on how to dynamically estimate the completion time of a job during its execution[22], [23], [24], they cannot meet our scheduling requirement: the execution time of all jobs should be estimated before running. Estimating the execution time for a new job is very difficult since it is impacted by multiple factors. The following is a list of such factors:

- *the problem the job is solving and the implementation complexity*: Different MapReduce algorithms and different implementations play the decisive role for job execution time.

- *the input and output data scale*: In general, the job execution time is also related to the input and the output data sizes. For fixed cluster resources, the more data to be processed, the longer time it will take to execute the job.
- *the distribution of the input data*: In large scale data processing applications, data distribution determines the complexity of computation and the output data scale [25]. For example, the join of two tables with uniform distribution is much faster than the join of two tables with Zipf distribution even when the data scale of the latter is smaller. The reason is the join of tables with Zipf distribution are more likely to generate uneven reduce load and increase the scale of the output data exponentially.

How can we determine the complexity of a program? The simplest solution is to let the users provide it for us. However, this will cause a loss of transparency and difficulty for the user. Another solution is to use the existing code library to check for the code reuse [26] for those commonly used MapReduce applications. This solution seems more reasonable. However, it can only be applied for a limited number of applications. For those user defined applications, it cannot provide proper estimations.

How can we get the input and output data scale? The input and output data scale is determined by the data set. Different data sets will result in different output data scales although they may have the same input data size. For example, joining two tables will result in no output when there are no identical keys. It can also result in a large scale output when the records in these tables share the same key. Therefore, it is hard for us to judge the input and output data scale.

How can we obtain the distribution of the input data? There are several previous approaches launching some pre-run sampling jobs to collect the data distribution statistics. They can then make a balanced partition decision according to the estimated data distribution [10], [27], [28]. However, these pre-run sampling jobs will bring too much overhead and can only be used in some special applications whose intermediate keys are the same as the input keys, such as join, sort, etc..

Since all of the three factors above are difficult to estimate, we cannot make an accurate estimation of job runtime. Moreover, the only information we can get before the job execution is the job configuration, its binary execution code and its input data size. Then how should we estimate the job execution time? We notice that some real world applications with large data scale often run periodically when the input data changes or when new data arrives. Such kind of applications include log analysis, database operations, machine learning, data mining, etc.. RoPE [15] has observed that these recurring jobs have accounted for 40.32 percent of all jobs and 39.71 percent of all cluster hours. Since the program complexity is the same for the same job and the data distribution will not change too much for the same application, we believe that the execution speed and the ratio between the input and output size will not change too much for the same recurring application, either.

PISCES makes use of this property by building a job history database to record the execution information of each job. This information is then used to estimate the execution time of the recurring jobs. We use job name to separate the different kinds of jobs since once the application code is written, the name of the application will not be changed generally.

In order to estimate the runtime for a job and the input data size of its downstream jobs, the metrics we need to predict for each job are the *processing speed of each phase in map and reduce tasks* (the ratio between the input data size and the duration of a phase) and the *output data size*. Popescu et al. [29] have observed that there are strong correlations between input data size and output data size, and between input data size and processing speed. They use uni-variate linear regression model (UVLR) to estimate the runtime and the output data size according to the input size, which can achieve less than 25 percent runtime prediction errors for 90 percent of predictions. However, we notice that the execution time of reduce tasks and the output data size can be super-linear correlated with the input data size. For example, the reduce computational complexity of the self-join application is $O(n^2)$. Therefore, we use a novel regression model called locally weighted linear regression [17] to make the estimation. Instead of using the whole observation set, LWLR estimates the target y at x only using the q observations whose x_i values are closest to x and weights these q observations according to their distance from x. That means we give those observations whose input data size are closed to the current input data size higher weight in the estimation. Moreover, in order to make accurate estimation, we always prefer those latest observations since they represent the current state of the system. Therefore, the weight we use for the observation (y_i, x_i) is

$$w_i(x) = \begin{cases} \frac{1}{rank(d(x,x_i))+time(x,x_i)}, & rank(d(x,x_i)) \le q \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $rank(d(x, x_i))$ represents the rank of the distance between observation x_i and x (long distance has large rank value), $time(x, x_i)$ represents the difference between current observation id and observation id i. We only weight the q observations whose x_i values are closest to x. According to our weight function, we can easily find that observations that are close to x in both time and distance will obtain high weight in the estimation. As we will see later in the experiments, our novel estimation model can achieve about 90 percent precision in both linear and super-linear applications after a few runs.

To estimate the runtime of a group of dependent jobs, the first step is to estimate the input data size for each job. For those jobs whose input data are ready, we can easily obtain their input data size and estimate the runtime of each phase and output data size using our novel LWLR model. For those jobs whose input data have not been produced yet, we first estimate their input data sizes according to the output data sizes of their dependent jobs, and then use the input data size to estimate the runtime of each phase and output data size. When a job has no execution history, we use the average value of all existing jobs to estimate its runtime of each phase and output data size since we cannot get any other information.

Generally speaking, a typical MapReduce job includes a large number of map tasks and several reduce tasks. Each map task contains two phases: a map phase and a combine phase, while each reduce task contains three phases: shuffle, sort, and reduce. We can estimate the job execution time as follows:

$$T_{job} = R_{map} * (T(\text{Map}) + T(\text{Combine})) + R_{reduce} * (T(\text{Shuffle}) + T(\text{Sort}) + T(\text{Reduce})), \quad (2)$$

where $T(X)$ represents the execution time of the X phase. R_{map} and R_{reduce} represent the number of rounds of the Map and the Reduce tasks, respectively.

Note that in the existing MapReduce framework, when a certain percentage of map tasks have completed (e.g., 5 to 10 percent in Hadoop), the shuffle phase of reduce tasks can start copying its data from the map tasks. In other words, the shuffle phase can be executed in parallel to the map tasks. In practice, the shuffle phase of the first round reduce tasks typically finishes around the same time as the map tasks. For the remaining rounds of reduce tasks, since all map tasks have completed, the shuffle phase can be executed very quickly. Therefore, we can ignore the shuffle time and simplify the equation as follows:

$$T_{job} = R_{map} * (T(\text{Map}) + T(\text{Combine})) + R_{reduce} * (T(\text{Sort}) + T(\text{Reduce})). \quad (3)$$

Moreover, if a job has several rounds to execute the map or the reduce tasks, we can consider the system as running at full capacity (i.e., no free slots) except the last round. Therefore, we only need to consider the execution time of the map or the reduce tasks in the last round when making scheduling decisions. Our goal is to avoid the substantial decrease of the system parallelism when the number of tasks in the last round is small and their execution time is long. We schedule the critical job chain with the longest total last round runtime first so that we can have more choices to fill up the blanks in these last rounds. For this reason, we use the execution time of the map and the reduce tasks in the last round to represent the execution time of the job

$$T_{job} = T(\text{Map}) + T(\text{Combine}) + T(\text{Sort}) + T(\text{Reduce}). \quad (4)$$

Furthermore, with our innovative job pipeline optimization, the map tasks of a job can be executed in parallel with the output tasks in the upstream jobs. Therefore, either the execution time of reduce task in the upstream job or the execution time of map task in the downstream job needs to be calculated in our job scheduling. We modify the estimation of the job execution time accordingly. We divide T_{job} into two parts.

$$T_{job_map} = T(\text{Map}) + T(\text{Combine})$$
$$T_{job_reduce} = T(\text{Sort}) + T(\text{Reduce}). \quad (5)$$

3.3.3 *Job Scheduling Algorithm*

After estimating the job execution time for each submitted job, we can make a job scheduling decision to determine which job runs first whenever there are free slots in the system and the current job running queue is empty. PISCES uses the modified Critical Path Method (CPM) [30] to schedule a group of jobs with dependencies. The goal of the Critical Path Method is to differentiate the critical activities which affect the progress of the project from those non-critical activities which can be delayed without making the project longer. Then it mainly optimizes the critical activities during its scheduling decision. We redefine the critical path as the job chain that has the longest total last round task runtime. We run the schedulable jobs in the critical chains first when there are free slots in the system so that we can have more choices to fill up the blanks in these last rounds.

The input of our scheduling algorithm are a group of jobs and their dependency DAG $G = <V, E>$, and the estimated task execution time for each job. Suppose that there are n jobs in the graph G (i.e., $|V| = n$) and that the estimated execution time of map and reduce task in job i is $T_{job_map}(i)$ and $T_{job_reduce}(i)$. We add a new virtual source vertex s ($T_{job_map}(s) = T_{job_reduce}(s) = 0$) and a directed edge from s to each vertex whose incoming degree is zero. We also add a new virtual sink vertex t ($T_{job_map}(t) = T_{job_reduce}(t) = 0$) and a directed edge from each vertex whose outgoing degree is zero to t.

We order the new graph G into a list by using a topological sort. Then we calculate the earliest start time for each job (we consider the start time of sort and reduce phases in job i as the start time of job i) in the list as follows:

$$b(i) = \max_{<j,i>\in E}\{b(j) + \max\{T_{job_reduce}(j), T_{job_map}(i)\}\}, \quad (6)$$

where the boundary condition is $b(s) = 0$. The meaning of this equation is that the earliest start time of a downstream job is equal to the maximum value of the earliest finish time of all its upstream jobs, because a downstream job cannot enter the sort and the reduce phases until its last upstream job complete. The earliest finish time of a job can be calculated by the sum of its earliest start time and its execution time. Here we consider the execution time of an upstream job as the duration between the start of its reduce stage and the completion of the map stage in the current downstream job since these two stages will be executed overlapped and cannot be calculated twice.

Then we calculate the latest finish time for each job in inverse topological sequence as follows:

$$e(i) = \min_{<i,j>\in E}\{e(j) - T_{job_reduce}(j) - \max\{0, T_{job_map}(j) - T_{job_reduce}(i)\}\}, \quad (7)$$

where the boundary condition is $e(t) = b(t)$. This means that the latest finish time of an upstream job is equal to the minimum value of the latest start time of all its downstream jobs, because it should not delay any of its downstream jobs. The latest start time of a job is calculated according to its latest finish time and its execution time. Here the execution time of a downstream job contains two parts: its reduce execution time and the extra map execution time after the completion of the current upstream job.

We define those jobs which satisfy the condition $b(i) + T_{job_reduce}(i) = e(i)$ as the critical jobs. Since the jobs in the critical chain may affect the system parallelism, we give them higher priority during scheduling so that we can have more choices to fill up the blanks. There may be multiple critical chains in the DAG in which case the schedulable job list may contain multiple critical jobs. For example, for the job dependency DAG in Fig. 6, Table 2 shows the calculation of its critical chains. From the table we can see that job 1 and job 3 are both critical and schedulable jobs. In this case, we schedule the job with the shortest execution time first (i.e., job 1). The rationale is to minimize the influence due to inaccurate execution time estimation and the uncertainty during actual execution.

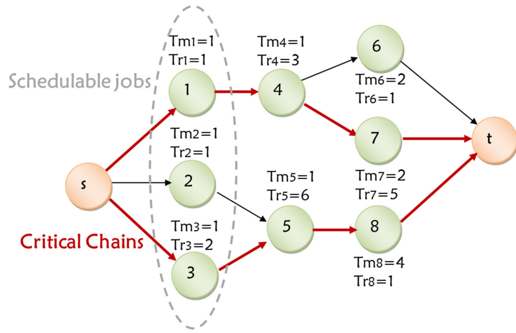


Fig. 6. Example of a job dependency DAG.

One problem of the job scheduling method is starvation: since new jobs are submitted to the system continuously, early jobs that are never in the critical chain will be delayed forever. To tackle this problem, we take the job priority into consideration when making scheduling decision. The priorities of the submitted jobs are increased automatically as time goes by. During job scheduling, we first pick the jobs with the highest priority as the scheduling candidates and then use the CPM method to choose one for running. A user can also increase the priority of some jobs if they are urgent to be executed.

As we will see later in the experiments, PISCES can make significant improvement in not only the application execution time but also the resource utilization of the system.

4 Evaluation

In this section, we evaluate the performance of our PISCES system. Here is a highlight of our findings:

1) PISCES can run data intensive iterative applications much faster than the original MapReduce system. It improves the system parallelism by 31 percent and runs application 41 percent faster (Section 4.2).
2) PISCES can schedule a group of jobs more effectively. It increases the degree of system parallelism by up to 68 percent and improves the execution speed of the PigMix application by up to 52 percent (Section 4.3).
3) PISCES can make effective use of system resources such as the file cache of the operating system (Section 4.3) and deal with "hard dependency" among jobs well (Section 4.4).

TABLE 2
Critical Chain Calculation

	s	**1**	2	**3**	4	5	6	7	8	t
$T_{job_map}(i)$	0	**1**	1	**1**	1	1	2	2	4	0
$T_{job_reduce}(i)$	0	**1**	1	**2**	3	6	1	5	1	0
$b(i)$	0	**1**	1	**1**	2	3	5	5	9	10
$e(i)$	0	**2**	3	**3**	5	9	10	10	10	10

4) PISCES can estimate the task execution time and output data size according to the job history accurately. The precision can reach above 80 percent in general (Section 4.5).

4.1 Experiment Environment

We set up our experiments on a Hadoop cluster with 30 nodes on 15 servers. Each server has dual-Processors (2.4 GHz Intel Xeon E5620 with 8 physical cores), 24 GB of memory, and two 150 GB disks. They are connected by 1 Gbps Ethernet and controlled by the OpenStack Cloud Operating System [31]. We use the KVM virtualization software [32] to construct two virtual machines on each server. Each virtual machine is of medium size with two virtual cores, 4 GB memory and 30 GB disk space. We configure the memory capacity for each worker node to 3 GB. The memory usages for each map and reduce task are 1 and 2 GB, respectively. Other configurations we use for HDFS, Yarn and MapReduce are the default configurations except as otherwise noted. We evaluate PISCES in the following three kinds of applications:

- *Data-intensive iterative application:* We use PageRank [2] as the data-intensive iterative application. PageRank is the best-known link analysis algorithm used in the web search area. It assigns a numerical weight (rank) to each vertex in a hyperlinked document graph by iteratively aggregating the weights from its incoming neighbors. We apply it to the free ClueWeb09 web graph full dataset [33]. The total input data size is about 30 GB.
- *Database operations:* We use the PigMix2 benchmark [34] which is a set of queries used to test pig performance as the database applications. This benchmark contains some most commonly used operations, such as select, count, group, join, union, etc. We run it on a

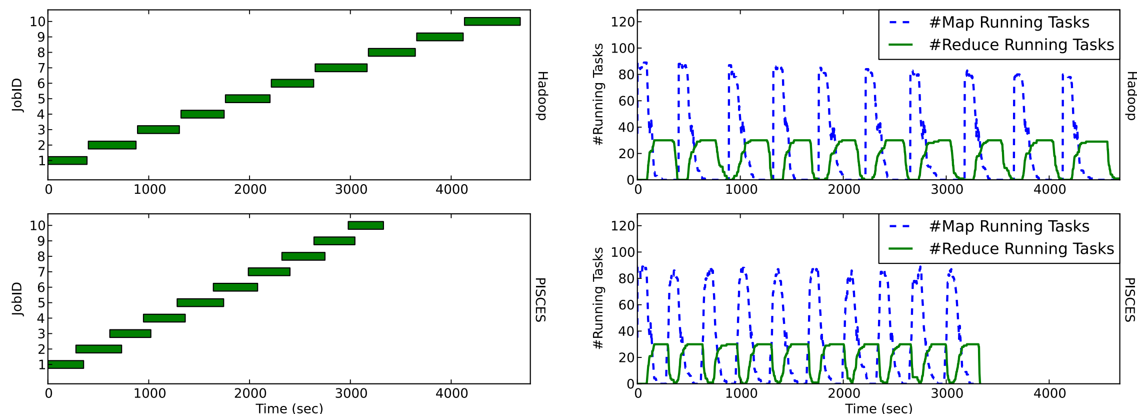


Fig. 7. Job execution time (left) and parallelism of the system (right) in PageRank.

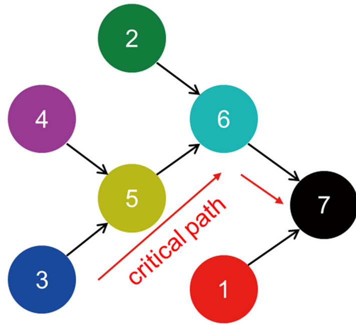


Fig. 8. Job dependency DAG.

data set of 80 GB. The queries we used are the combination of some simple queries.

- *Machine Learning application with hard dependency:* We use Naive Bayes example in Mahout [20] as the machine learning application. Naive Bayes algorithm is a simple classifier based on Bayes theorem which assumes the features have strong independence with each other. In Mahout example, it is used to classify 20 news groups for the 45 MB news data set.

We execute each test case at least three times and take their average value. We compare PISCES with Hadoop and PIG. We take the total execution time of the job group and the parallelism of the system as our primary metrics to measure the effectiveness of our PISCES system.

4.2 Data-Intensive Iterative Application

To demonstrate the advantage of the pipeline optimization in PISCES, we run the PageRank application ten iterations. The execution time of each iteration and the parallelism of the system is shown in Fig. 7. From the figure we can see that the degree of parallelism in PISCES is 31 percent higher than that in Hadoop. The execution speed in our system is 41 percent faster than that in Hadoop. Moreover, we can see that PISCES decreases the interval between each iteration. This is because the map phase of a downstream job can be started much earlier when its upstream job has generated a certain amount of output data. As a result, we can take full advantage of the resources in the system during the execution. The average and the standard deviation (stdev) of resource usage among all virtual machines are shown in Fig. 10. As we can see from the figure, PISCES makes more effective use of resources (e.g., CPU).

4.3 Database Operations

To evaluate how PISCES can do better in job scheduling than PIG, we run the PigMix2 benchmark with 80 GB data. The dependency DAG of the job group is shown in Fig. 8. The execution time of each job and the parallelism of the system is shown in Fig. 9. From the figure, we find that the improvement of PISCES is significant: the execution speed in PISCES is 52 percent faster than that in PIG. Even without pipeline optimization, PISCES can run 22 percent faster than PIG. The degree of system parallelism (measured by the number of concurrent running tasks) in PISCES is 68 percent higher than in PIG.

Interestingly, the figure shows that some downstream jobs run substantially faster in PISCES than in PIG. An example is job 6 (colored blue) in the figure. To understand the reason, we collect the resource usage statistics of the system using *vmstat* tool every three seconds. The average and the standard deviation (stdev) of resource usage among all virtual machines are shown in Fig. 11. From the CPU and I/O usage figures, we can see that at around 300 seconds, the average CPU usage and I/O bandwidth are very high while the stdev is low, which means PISCES can make more effective use of CPU resource and I/O bandwidth. From the swap usage figure, we can see that at around 300 and 600 seconds, there are two swap peaks in PIG. The reason that PIG needs more swap operations is that the upstream job and the downstream job in PIG are not scheduled next to each other. Therefore, it cannot take advantage of the file cache of the operating system. In contrast, in PISCES as soon as an output data block of an upstream job is written to the file system, it can be executed by the map task of a downstream job. This allows PISCES to take full advantage of the file cache.

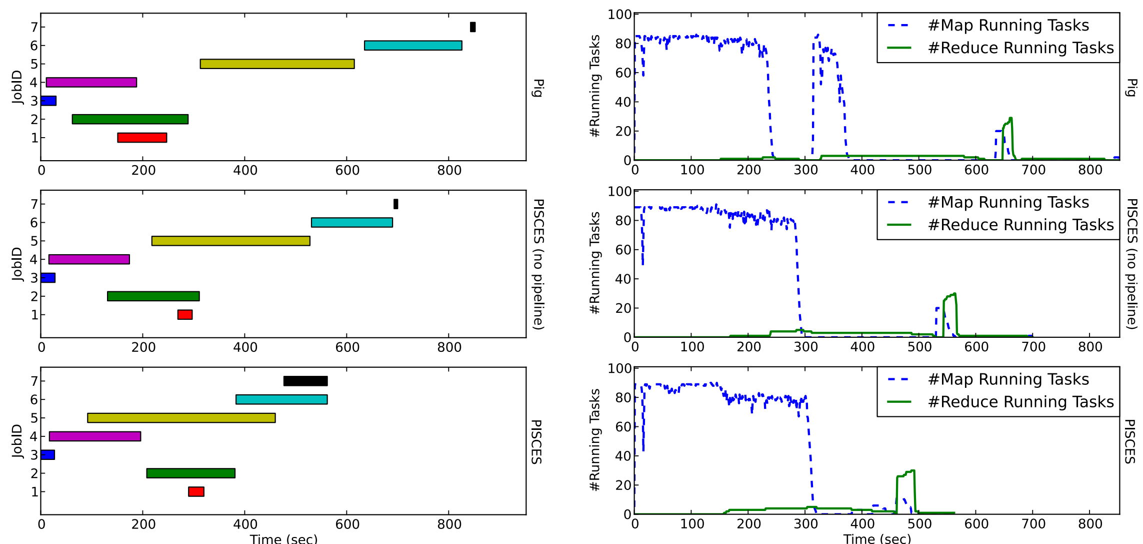


Fig. 9. Job execution time (left) and parallelism of the system (right) in PigMix.

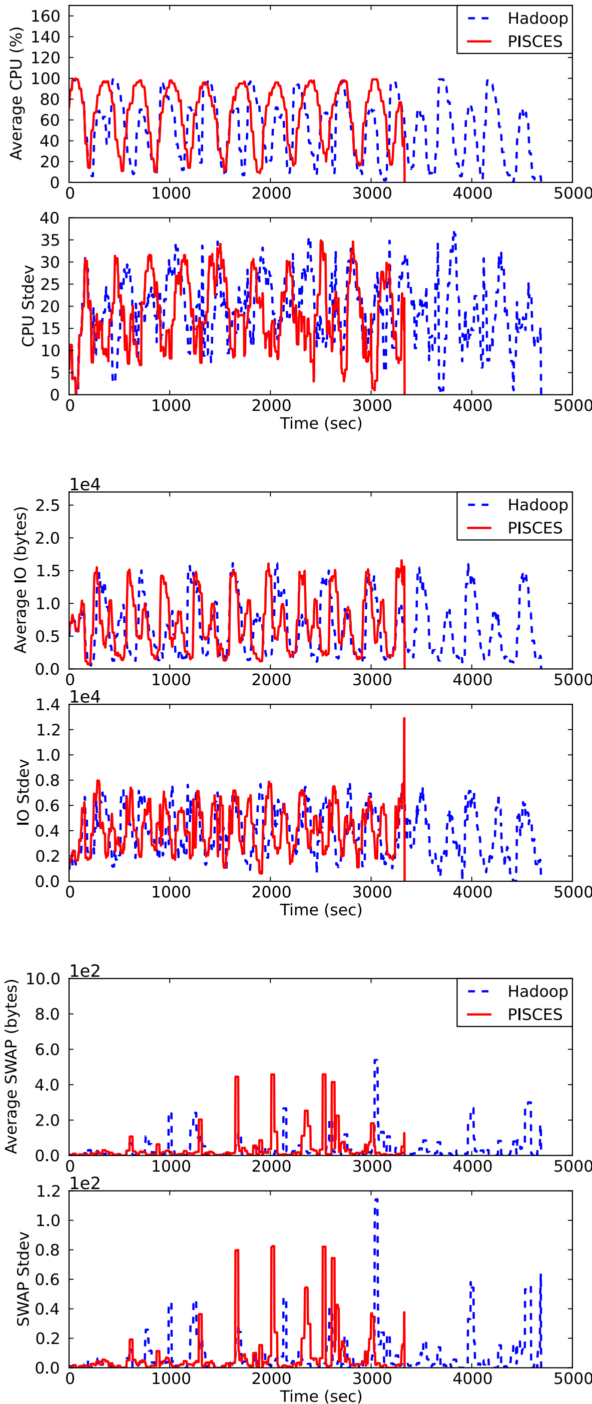


Fig. 10. Resource usage of the system in PageRank.

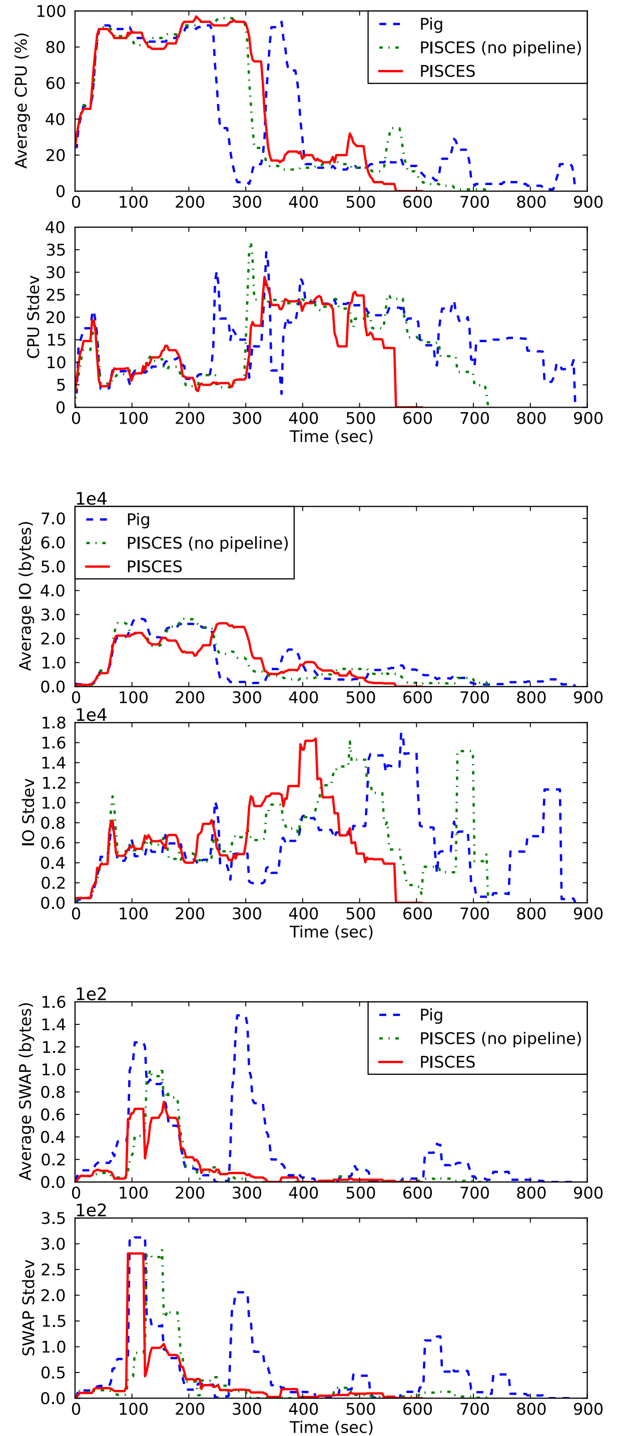


Fig. 11. Resource usage of the system in PigMix.

Moreover, the figure shows that the pipeline optimization not only improves the performance of PISCES but also optimizes its job scheduling decisions. Without pipelining, PISCES schedules job 2 (colored green) first after job 4 (colored purple) while with pipelining it schedules job 5 (colored yellow) first. This is more desirable since job 5 is on the critical chain. The reason for this difference is the following: Job 4 released some free slots at around 100 seconds as some (but not all) of its reduce tasks complete. Without pipelining, PISCES cannot issue job 5 at this moment because job 4 (which it depends on) has not yet entirely finished. In contrast, with pipelining PISCES can initiate job 5 since job 4 has already produced a certain amount of output data.

4.4 Machine Learning Application

To demonstrate the effectiveness of PISCES in machine learning applications with "hard dependency", we run the Naive Bayes application in Mahout. The execution time of each job is shown in Fig. 12. From the figure, we can see that in Naive Bayes application, even through there exist some hard dependencies between the jobs, we can still achieve 30 percent improvement on the execution speed.

4.5 Accuracy of Prediction

As described in Section 3.3.2, we use the locally weighted linear regression model to estimate the runtime of the

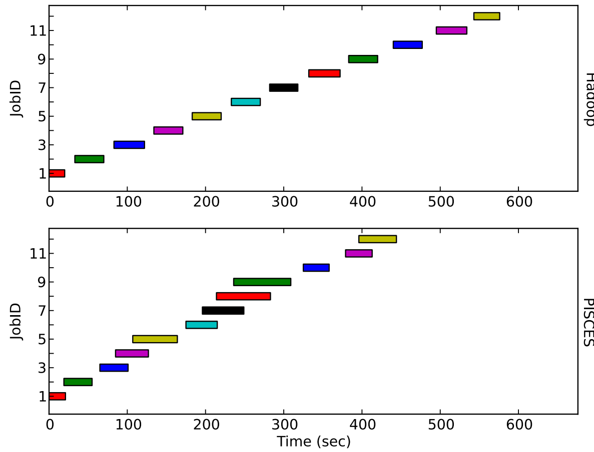


Fig. 12. Job execution time in Naive Bayes application.

submitted jobs. To evaluate how accurate our predictions are, we first run the PigMix2 benchmark with 30 random data sizes. For each data size, we run L1 to L17 pig scripts (most of which are input size and runtime linear-correlated applications) and then calculate the average precision of the execution time and the output size estimations. The result is shown in the left-hand side of Fig. 13. From the figure, we can see that the precision of both uni-variate linear regression and locally weighted linear regression can reach above 80 percent after a few runs. To demonstrate the precision of the predictions for the super-linear applications, we run the self-join application with 30 random data sizes (each data set is uniformly distributed). The result is shown in the right of Fig. 13. From the result, we can see that the precision of our LWLR model can reach above 90 percent after a few runs. However, the precision of reduce time and the output size in UVLR model are much worse than that in our LWLR. It can result in poor prediction accuracy even after many runs. The reason is that uni-variate linear regression cannot deal with super-linear applications well.

5 Related Work

Scheduling is a common and important problem in the distributed and parallel computing area. In MapReduce, it can be divided into two categories: fine-grain task scheduling and coarse-grain job scheduling. For fine-grain task scheduling, there are several prior studies attempting to provide special strategies for multiple purposes. Fair Scheduling [10] focuses on the fairness among the users. Capacity Scheduling [10] supports the fairness and resource-intensive jobs. Quincy [7] and Delay Scheduling [8] consider the tradeoff between fairness and data locality. The approach in [6] focuses on prioritizing the users, the stages and the bottleneck components by dynamically configuring the priority of different phases. Multiple resource scheduling [35] studies heterogeneous resource demands and the fairness among the users. LATE [24], Mantri [36], and MCP [23] tackle stragglers. All of the above approaches only consider how to schedule tasks within a job. Since MapReduce itself does not support the dependency analysis among the jobs, these approaches use only the simplest queue based FIFO strategy to schedule jobs.

In order to make MapReduce support complex queries, there are several approaches providing easy-to-use SQL like languages to simplify the programming of MapReduce and generating multi-job workloads. For example, Pig [9] for Apache Hadoop, Scope [11] for Microsoft Dryad, and Tenzing [13] for Google MapReduce use traditional query optimization strategies from the database area to transform a query into an internal parse tree and then translate the tree directly to a physical job execution plan by traversing the tree in a bottom-up fashion. These approaches provide a simple coarse-grain job scheduling to satisfy the dependency relationship among multiple jobs at a higher level than MapReduce. Therefore, they cannot schedule jobs according to the detailed job execution information or take advantage of the general job pipeline mechanism between dependent jobs.

There are also multiple work focusing on optimizing the coarse-grain job scheduling for these multi-job workloads. HiveQL [14], YSmart [37], MRShare [38], Starfish [39], RoPE [15] and Stubby [40] all use a series of hand-crafted rules to further optimize the job execution plans. HiveQL performs multiple passes over the logical plan using some optimization rules. For example, it combines multiple joins that share the same join key into a single multi-way join, provides repartition operators for some MapReduce operators, and puts predicates closer to the table scan operators in order to cut away columns early and minimize the amount of data transferred between jobs. YSmart translates queries based on four job primitive types: selection-projection, aggregation, join and sort. It then merges the MapReduce jobs according to their primitives to minimize the total number

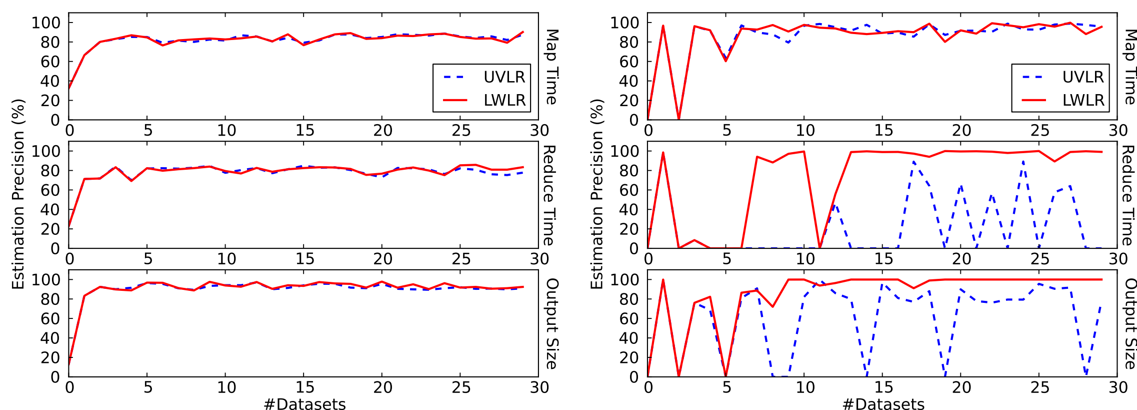


Fig. 13. The accuracy of job runtime estimation in linear (left) and super-linear applications (right).

of jobs. MRShare merges the similar jobs into a group and evaluates each group as a single job based on a cost model. Starfish provides only a cost-based configuration optimization for a group of jobs. RoPE [15] adapts execution plans based on estimates of code and data properties. It collects properties at many locations in each single pass over data, matches statistics across sub-graphs and adapts execution plans by interfacing with a query optimizer. For instance, it can coalesce operations with little data into a single physical operation, reorder commutative operations so that the more selective operation (one with a lower output to input ratio) runs first, and re-optimize future invocations for recurring jobs. Stubby is a cost-based optimizer that provides vertical packing (converting some special jobs into Map-only jobs and merging Map-only jobs with previous jobs), horizontal packing (just like MRShare), partition function (just like HiveQL and RoPE) and configuration transformations (just like RoPE and Starfish) to optimize the execution plan for a group of jobs. These rule-based approaches focus only on how to simplify and optimize a job execution plan and care less about the real execution of the plan. This is because they work at a higher level in the software stack than MapReduce and thus do not have access to detailed job execution information. Therefore, they cannot accurately estimate job execution time and figure out the critical chain in a series of jobs. Moreover, they cannot take advantage of the general job execution overlapping among dependent jobs for the underlying MapReduce system knows little dependency information among jobs. However, our PISCES is implemented in the MapReduce framework. Therefore, we can further improve the real execution of the optimized job execution plan by overlapping the output phase of the upstream jobs and the map phase of the downstream jobs, estimating the accurate execution time for all jobs, and running the critical jobs preferentially. FlowFlex [16] provides a theoretical malleable scheduling for multi-job workloads. However, the fixed amount of work unit for each job in its model is hard to measure in the real computing environment. Additionally, it is hard to add our new overlapping feature of dependent jobs into its theoretical scheduling model.

MapReduce Online [3] and HPMR [4] provide intermediate data pipelining between the map and the reduce stages in a job in order to adjust their load dynamically. They do not support the kind of parallelism in execution among dependent jobs the way we do. MapReduce Online supports online aggregation by applying the reduce function to the partial map intermediate data received so far, generating an intermediate output snapshot (different from the final output), and pipelining the intermediate output snapshot to the downstream job for execution. It will generate many intermediate output snapshots for each job on different percentage of the map intermediate data. Each time a new intermediate output snapshot is generated, it will re-execute all the downstream jobs. This is quite different from our pipeline mechanism. In fact, MapReduce Online cannot save the total execution time of the group of jobs because when the final intermediate output snapshot (i.e., the final output) of an upstream job is generated, all the downstream jobs still need to be re-executed.

For iterative applications, Haloop [41] provides an extension for MapReduce to support multiple stages in a job instead of the original map and reduce two stages and does loop-aware task scheduling by adding a variety of caching mechanisms. It is quite different from our optimization. We can add our data pipeline mechanism into Haloop to further improve the efficiency of the system.

6 Conclusion

Job scheduling is essential to the multi-job applications in MapReduce. This paper has presented PISCES, a system that implements an innovative pipeline optimization among dependent jobs and a critical chain job scheduling model in an existing MapReduce system to minimize the application execution time and maximize resource utilization and global efficiency of the system. PISCES extends the MapReduce framework to allow scheduling for multiple jobs with dependencies by building a dependency DAG for the running job group dynamically. It innovatively overlaps the output phase of the upstream jobs and the map phase of the downstream jobs, accurately estimates the job execution time based on detailed historical information, and effectively schedules multiple jobs based on a critical job chain model. Performance evaluation demonstrates that the improvement of PISCES is significant and that PISCES can make effective use of system resources such as the file cache of the operating system.

Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work was supported by the National High Technology Research and Development Program (“863” Program) of China (Grant No.2013AA013203) and the National Natural Science Foundation of China (Grant No. 61572044). The contact author is Zhen Xiao.

References

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Comput. Netw. ISDN Syst.*, vol. 30, no. 1–7, pp. 107–117, 1998.

[3] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, “MapReduce online,” in *Proc. 7th USENIX Conf. Networked Syst. Des. Implementation*, 2010, pp. 21–21.

[4] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, “HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment,” in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–8.

[5] J. Wolf, et al., “On the optimization of schedules for MapReduce workloads in the presence of shared scans,” *VLDB J.*, vol. 21, no. 5, pp. 589–609, Oct. 2012.

[6] T. Sandholm and K. Lai, “MapReduce optimization using regulated dynamic prioritization,” in *Proc. 11th Int. Joint Conf. Meas. Modeling Comput. Syst.*, 2009, pp. 299–310.

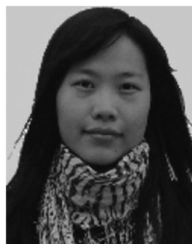
[7] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair scheduling for distributed computing clusters,” in *Proc. ACM 22nd Symp. Operating Syst. Principles*, 2009, pp. 261–276.

[8] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling,” in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: A not-so-foreign language for data processing,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.

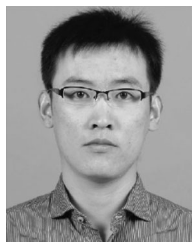
[10] [Online]. Available: *Apache hadoop*, http://hadoop.apache.org/

[11] R. Chaiken, et al., "SCOPE: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endowment*, vol. 1, pp. 1265–1276, Aug. 2008.
[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.
[13] B. Chattopadhyay, et al., "Tenzing a SQL implementation on the MapReduce framework," *Proc. VLDB Endowment*, vol. 4, pp. 1318–1327, Sep. 2011.
[14] A. Thusoo, et al., "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endowment*, vol. 2, pp. 1626–1629, Aug. 2009.
[15] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proc. 9th USENIX Conf. Networked Syst. Des. Implementation*, 2012, pp. 281–294.
[16] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum, "FlowFlex: Malleable scheduling for flows of MapReduce jobs," in *Proc. ACM/IFIP/USENIX 14th Int. Middleware Conf.*, 2013, pp. 103–122.
[17] W. S. Cleveland and S. J. Devlin, "Locally weighted regression: An approach to regression analysis by local fitting," *J. Amer. Statistical Assoc.*, vol. 83, no. 403, pp. 596–610, 1988.
[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 29–43.
[19] J. Lin, "Cloud9: A MapReduce library for hadoop. [Online]. Available: http://www.umiacs.umd.edu/~jimmylin/cloud9/docs/index.html
[20] Apache mahout: Scalable machine learning and data mining. [Online]. Available: https://mahout.apache.org/
[21] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics Operations Res.*, vol. 1, no. 2, pp. 117–129, 1976.
[22] J. Polo, et al., "Performance-driven task co-scheduling for MapReduce environments," in *Proc. IEEE Netw. Operations Manage. Symp.*, 2010, pp. 373–380.
[23] Q. Chen, C. Liu, and Z. Xiao, "Improving MapReduce performance using smart speculative execution strategy," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 954–967, Apr. 2014.
[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 29–42.
[25] M. Lakshmi and P. Yu, "Limiting factors of join performance on parallel processors," in *Proc. 5th Int. Conf. Data Eng.*, 1989, pp. 488–496.
[26] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School Comput. Queen's Univ., Kingston, ON, Canada, Tech. Rep. 2007-54, 2007.
[27] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 949–960.
[28] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 75–86.
[29] A. Popescu, V. Ercegovac, A. Balmin, M. Branco, and A. Ailamaki, "Same queries, different data: Can we predict runtime performance?" in *Proc. IEEE 28th Int. Conf. Data Eng. Workshops*, 2012, pp. 275–280.
[30] J. E. Kelley, Jr and M. R. Walker, "Critical-path planning and scheduling," in *Proc. Eastern Joint IRE-AIEE-ACM Comput. Conf.*, 1959, pp. 160–173.
[31] Open stack cloud operating system. [Online]. Available: http://www.openstack.org/
[32] K. Avi, K. Yaniv, L. Dor, L. Uri, and L. Anthony, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.
[33] Clueweb09 data set. [Online]. Available: http://boston.lti.cs.cmu.edu/clueweb09/wiki
[34] Pigmix2. [Online]. Available: https://issues.apache.org/jira/browse/pig-200
[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Networked Syst. Des. Implementation*, 2011, pp. 323–336.
[36] G. Ananthanarayanan, et al., "Reining in the outliers in Map-Reduce clusters using Mantri," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 265–278.
[37] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, "YSmart: Yet another SQL-to-MapReduce translator," in *Proc. 31st Int. Conf. Distrib. Comput. Syst.*, 2011, pp. 25–36.
[38] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "MRShare: Sharing across multiple queries in MapReduce," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 494–505, Sep. 2010.
[39] H. Herodotou, et al., "Starfish: A self-tuning system for big data analytics," in *Proc. 5th Biennial Conf. Innovative Data Syst. Res.*, Jan. 2011, pp. 261–272.
[40] H. Lim, H. Herodotou, and S. Babu, "Stubby: A transformation-based optimizer for MapReduce workflows," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1196–1207, Jul. 2012.
[41] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 285–296, Sep. 2010.



Qi Chen received the bachelor's degree from Peking University, in 2010. She is currently working toward the doctoral degree at Peking University. Her current research focuses on the cloud computing and parallel computing.

Jinyu Yao received the bachelor's degree from Peking University, in 2010. He is currently working toward the master's degree in the School of Electronics Engineering and Computer Science, Peking University. His current research focuses on the cloud computing and parallel computing.



Benchao Li received the bachelor's degree from Beijing University of Posts and Telecommunications, in 2014. He is currently working toward the master's degree in the School of Electronics Engineering and Computer Science, Peking University. His current research focuses on the cloud computing and parallel computing.

Zhen Xiao received the PhD degree from Cornell University, in January 2001. He is a professor in the Department of Computer Science, Peking University. After that he worked as a senior technical staff member with AT&T Labs, New Jersey and then a research staff member with IBM Thomas J. Watson Research Center. His research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of the ACM and the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.