Dependency-aware Network Adaptive Scheduling of Data-Intensive Parallel Jobs

Shaoqi Wang, *Student Member, IEEE,* Wei Chen, *Student Member, IEEE,*
Xiaobo Zhou, *Senior Member, IEEE,* Liqiang Zhang, *Member, IEEE,*Yin Wang, *Senior Member, IEEE*

Abstract—Datacenter clusters often run data-intensive jobs in parallel for improving resource utilization and cost efficiency. The performance of parallel jobs is often constrained by the cluster's hard-to-scale network bisection bandwidth. Various solutions have been proposed to address the issue, however, most of them do not consider inter-job data dependencies and schedule jobs independently from one another. In this work, we find that aggregating and co-locating the data and tasks of dependent jobs offer an extra opportunity for data locality improvement that can help to greatly enhance the performance of jobs. We propose and design Dawn, a dependency-aware network-adaptive scheduler that includes an online plan and an adaptive task scheduler. The online plan, taking job dependencies into consideration, determines where (i.e., preferred racks) to place tasks in order to proactively aggregate dependent data. The task scheduler, based on the output of online plan and dynamic network status, adaptively schedules tasks to co-locate with the dependent data in order to take advantage of data locality. We implement Dawn on Apache Yarn and evaluate it on physical and virtual clusters using various machine learning and query workloads. Results show that Dawn effectively improves cluster throughput by up to 73% and 38% compared to Fair Scheduler and ShuffleWatcher, respectively. Dawn not only significantly enhances the performance of jobs with dependency, but also works well for jobs without dependency.

Index Terms—Adaptive task scheduler, Network adaptive, Job dependency, Data-parallel clusters.

1 Introduction

Over the past decade, big data computing clusters with hundreds or thousands of servers have become increasingly common. Such data-parallel clusters usually deploy highly scalable computing frameworks like MapReduce [1] and Apache Tez [2] to run data-intensive parallel jobs for improving utilization and cost efficiency [3], [4], [5].

The performance of parallel jobs is often constrained by the cluster's hard-to-scale network for several reasons. First, jobs use network to read input data, as the input is randomly spread across several machines in a cluster. Second, jobs usually consist of intermediate data shuffle phases such as shuffle and join. Finally, when data flows generated by consecutive jobs are dependent, network is also used for transferring dependent data between jobs. Meanwhile, recent researches have shown that cross-rack network traffic is much higher than the cross-node traffic [6], [7], [8], [9] in clusters. Thus, our goal is to reduce cross-rack network traffic by improving rack-level data locality.

To improve the locality, several previous job schedulers (e.g., FIFO, Capacity, and Fair) employ techniques like delay scheduling [10] to optimize input data locality, but do not address other network-intensive phases (e.g., shuffle). Recent effort ShuffleWatcher [3] attempts to localize the shuffle phase of a MapReduce job by scheduling reducers on the same rack with most mappers. Results show that ShuffleWatcher achieves better locality for a single job with no dependency. However, it does not consider inter-job data dependencies, scheduling them independently from one another.

Many real-world workloads inherently exhibit strong job dependencies. Within the workload, a job often relies on the output of others, and it has to wait until all the parent jobs are finished. Such dependencies are usually presented as Directed Acyclic Graph (DAG). For instance, machine learning (ML) [11] and complex query [12] workloads usually have jobs exhibiting multiple levels of dependencies. In each level, specific jobs need to read the output from jobs in prior level and transfer result to jobs in subsequent level. However, without considering job dependencies, a traditional task scheduler, e.g., delay scheduling or ShuffleWatcher, would blindly place tasks, which generate the output of jobs in prior level, in different racks. Such placement may result in dependent data being spread over the cluster randomly so that the subsequent jobs have to read data using cross-rack network. This incurs significant network traffic and prolongs the overall completion time.

In fact, the availability of dependency profile allows us to carefully place dependent data to improve data locality. We can plan online and determine where, e.g., in which particular racks, tasks generating dependent data should be scheduled in the cluster. Thus, the dependent data can be aggregated. By adaptively co-locating aggregated data and tasks in the subsequent jobs, we can then improve the locality and free up more network bandwidth.

However, leveraging the dependency profile should address several challenges. First, complex dependency rela-

- *S. Wang, W. Chen and X. Zhou are with the Department of Computer Science, University of Colorado, Colorado Springs, CO 80918. E-mail: {swang, cwei, xzhou}@uccs.edu.*
- *L. Zhang is with the Department of Computer & Information Sciences, Indiana University South Bend, South Bend, IN 46615. E-mail: liqzhang@iusb.edu.*
- *Y. Wang is with the Department of Compute Science, Tongji University, Shanghai 201804. Email: yinw@tongji.edu.cn.*

tionship makes it difficult to determinate the optimal aggregation locations. For example, for certain dependent data, it could be the input of several jobs. Each of these jobs could rely on other dependent data. Second, co-locating data and tasks should adapt to dynamic network status [3] in clusters so as to further reduce network traffic. A detailed discussion is described in Section 2.4.

With this intuition, we propose Dawn, a Dependency-AWare Network adaptive scheduler that takes job dependencies into consideration and utilizes the dynamic network status to improve rack-level data locality. Dawn includes an online plan and an adaptive task scheduler. The online plan determines the preferred racks for tasks of each job based on their input data locations and job dependencies. The adaptive task scheduler, when it sees a rack has free resource (e.g., slot in Hadoop1.x and container in Apache Yarn [13]), tries to pick the most suitable job to schedule on that rack based on the current network status.

In a nutshell, Dawn tries to assign tasks generating dependent data in a gathered manner, in contrast to existing schedulers where tasks are scattered in the cluster. In its online plan, Dawn relies on two components, proactive task aggregation and dependency-aware task assignment, to determine preferred racks for tasks in different stages of the job (e.g., map and reduce stages) respectively. The proactive task aggregation aims to aggregate tasks in early stages (e.g., map stage) to realize intermediate data aggregation. The dependency-aware task assignment is to determine preferred racks for tasks in the final stage (e.g., reduce stage). It not only leverages the aggregated intermediate data to exploit data locality, but also tries to proactively maximize dependent data aggregation to further benefit subsequent jobs that are dependent on this current job.

Network load is highly dynamic in clusters [3], which can be utilized to guide schedule regarding when to schedule the task to its preferred racks so as to achieve more balanced network load. During unsaturated network periods. the adaptive task scheduler leverages the available network bandwidth to schedule a task that aggregates intermediate/dependent data. During saturated periods, it schedules a task that benefits from data aggregation, e.g., its dependent data has been aggregated on it preferred racks by tasks in prior stages.

We implement Dawn on Apache Yarn, and evaluate it in physical and virtual clusters using various MapReduce and Tez benchmark workloads that contain ML and complex queries. Results show that Dawn effectively improves cluster throughput by up to 73% and 38%, and reduces average workload completion time by up to 66% and 35%, compared to Fair Scheduler and ShuffleWatcher, respectively. Dawn also works well for jobs without dependency.

Our contributions mainly lie in the design, development, and evaluation of a new dependency-aware network adaptive job scheduler that significantly improves the cluster throughput and job performance for popular data-intensive parallel jobs with strong dependency.

A preliminary version of this paper appeared in the Proc. of IEEE ICAC'2017 [14]. In this significantly extended version, we have analyzed job dependencies in representative ML and query workloads and redesigned Dawn with online plan balancing algorithm. We have performed new experiments on a physical cluster to evaluate the performance of Dawn. We have also extended experiments to evaluate the sensitivity to cluster load.

The rest of this paper is organized as follows. Section 2 gives case study and motivations. Section 3 describes the design of Dawn. Section 4 gives the system implementation. Section 5 and Section 6 present the experimental setup and evaluation results. Section 7 reviews related work. Section 8 concludes the paper.

2 Motivation

We first introduce the importance of reducing cross-rack network traffic, and then analyze representative ML and query workloads to illustrate the job dependencies. After that, we provide a case study to show the potential gain of designing a dependency-aware scheduler. Finally, we validate dynamic network status in clusters and show that the dependency-aware scheduler can be further improved when adapting to the network status.

2.1 Network Bottleneck and Cross-rack Bandwidth

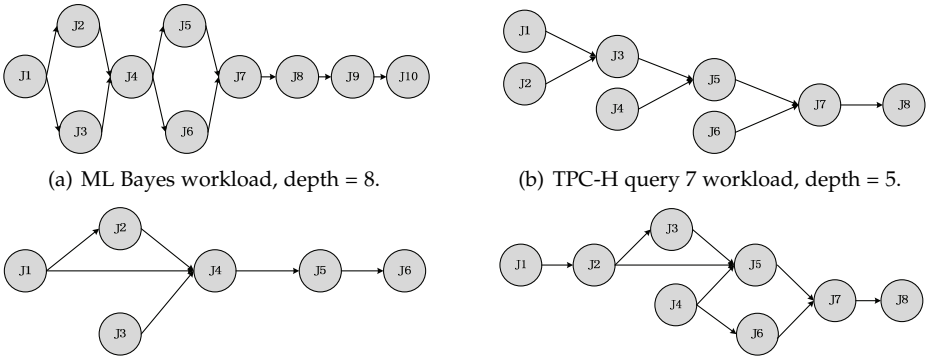
Datacenter clusters often run data-intensive jobs in parallel for improving resource utilization and cost efficiency. However, those jobs can cause a great volume of network traffic, imposing a serious constraint on the efficiency of data analytic applications [15], [16]. For instance, data is frequently transferred between nodes because of input data read, intermediate data shuffle, and dependent data transfer during the map, shuffle, and reduce stages [17], [18], [19]. As a result, network could become a potential bottleneck of a cluster with multiple data-intensive jobs running [18], [20].

Datacenters typically employ bandwidth over-subscription in the network for cost saving. Large datacenters usually have the bandwidth over-subscription ratios ranging from 2:1 to 20:1 or even higher [7], [8], [9], [21], meaning that the bandwidth at the aggregation layer (i.e., inter-rack layer) of network topology could be lower than the intra-rack bandwidth. When all nodes in clusters are concurrently communicating, the cross-rack bandwidth available for each node is much less than the bandwidth available within a rack [8], [10], [22] (e.g., 1-40 Gbps). Thus, our goal is to improve rack-level data locality so as to reduce cross-rack network traffic.

2.2 Dependency Analysis

In many workloads, data flows generated between consecutive jobs are dependent. In other words, a job often relies on the output of other jobs, and it has to wait until all parent jobs finish. To understand the job dependencies, we analyze two types of workloads: (a) five ML workloads from HiBench [23], namely, PageRank [24], K-means [25] and Bayes Classification [26], and (b) TPC-H query workloads [27].

Our analysis shows that these workloads typically exhibit strong job dependencies, as illustrated in Figure 1. The output of one job could become the input of several other jobs, and the input of one job could also come from the output of several other jobs. Figure 2 further plots the DAG depth of each workload. We can see that the median DAG has a depth of five.



(a) ML Bayes workload, depth = 8.

(b) TPC-H query 7 workload, depth = 5.

(c) TPC-H query 15 workload, depth = 5.

(d) TPC-H query 22 workload, depth = 6.

Fig. 1. Job DAGs in one ML workload and three query workloads. Each circle denotes a MapReduce job. The depth means number of jobs in the longest path from first job to last one in the DAG.

Among all the jobs analyzed, 80% of them depend on the output of other jobs, and 47% of them depend on at least two jobs. Dependent data aggregation benefits these 80% subsequent dependent jobs at the expense of proactive map task aggregation conducted in the rest 20% first-level non-dependent jobs.

Figure 3 compares the dependent and intermediate data size of the subsequent jobs with the input data size of the first-level jobs by plotting their percentages in the total size. Specifically, the dependent and intermediate data size is divided into several parts based on the sequence of the DAG level. For example, the first part from the bottom in the dependent data size represents the data size in DAG's second level jobs. The result shows that both dependent data size and intermediate data size are much larger than the input data size, indicating the potential gain of the proactive data aggregation. For instance, the average ratio of dependent data size and intermediate data size to the input data size in machine learning workloads is about 1/12. In iterative ML workloads such as PageRank and K-means, DAG contains iterative jobs across different levels. Thus, dependent and intermediate data size repeats across different levels of jobs (i.e., the 3^{rd} to the 10^{th} levels in K-means). In TPC-H queries, the intermediate data size varies across different levels of jobs. For instance, the intermediate data size in some jobs (e.g, 3^{rd} level jobs in query 11) is much smaller than the dependent data size due to query operations such as SELECT and INTERSECT. There also exist query operations such as JOIN and ORDER in which intermediate data size is equal to or larger than the dependent data size [28], [29].

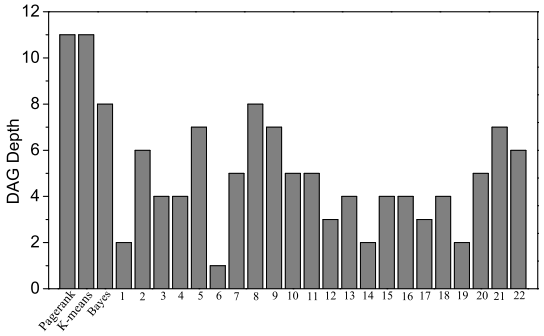


Fig. 2. DAG depth of workload, the first three are ML workloads, the rest represents 22 query workloads in TPC-H.

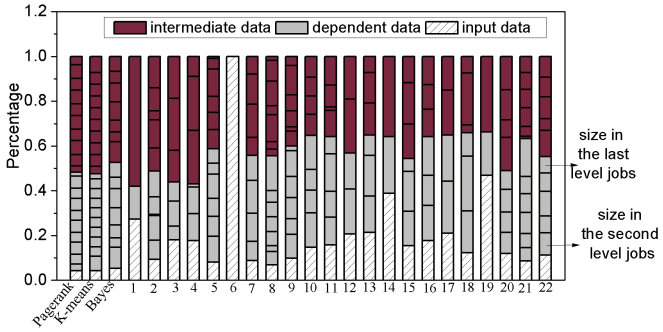


Fig. 3. Data size comparison.

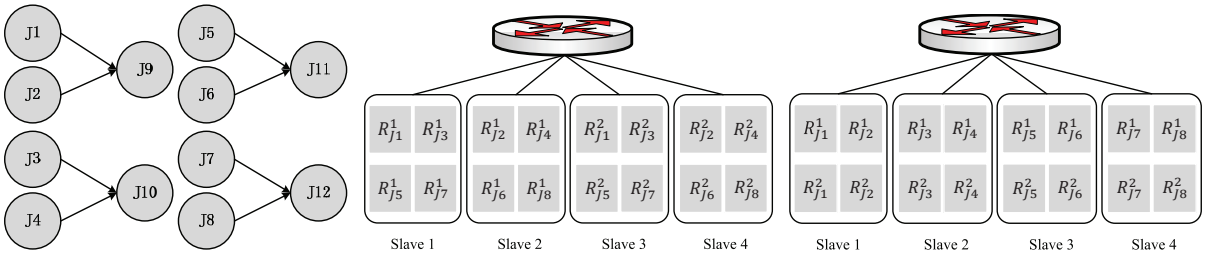
2.3 Dependency-Aware Case Study

Without loss of generality, we simulate cross-rack traffic via a small scale cross-node case study. The cross-node bandwidth is set to be 50 MBps. We created a 5-node cluster and ran 12 parallel MapReduce jobs with dependency. The cluster was configured with one master and four slave nodes. Figure 4(a) shows the dependency relationship between jobs. The relationship that one job depends on the output from other two jobs is common in ML workloads and TPC-H queries as shown in Figure 1. Specifically, jobs 1 to 8 belong to first level in the DAG and jobs 9 to 12 are in the second level. Each job contains 4 tasks in map stage and 2 tasks in reduce stage. To make it simple, the map task has 1 GB input data and the input size for reduce task is 2 GB. For each job in the first level, input data is evenly distributed in the cluster (i.e., one slave contains the input for one task).

The case study contains two parts: (1) we assume that 4 map tasks of each first level job are evenly distributed in the cluster and each task processes its input locally, and focus on how to leverage the dependency to schedule reduce tasks so as to reduce network traffic. (2) we further analyze the impact of map task scheduling to the dependency-aware scheduler in the first part.

2.3.1 Reduce Task Scheduling

After map tasks finished, intermediate data of each first level job is distributed in the cluster. Figure 4(b) shows the schedule of reduce tasks by existing schedulers (e.g., delay

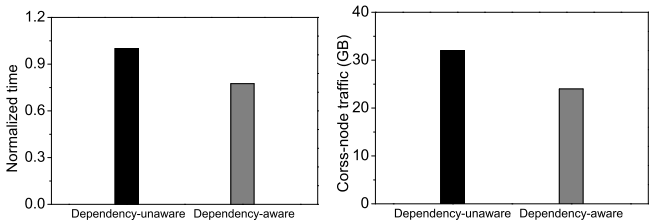


(a) DAG of Jobs with dependency. (b) Dependency-unaware task placement. (c) Dependency-aware task placement.

Fig. 4. A dependency-aware case study.

TABLE 1
Network traffic (GB) in each stage.

Scheduler	Map in job 1	Reduce in job 1	Map in job 9	Reduce in job 9
Dependency-unaware	0	3	0	2
Dependency-aware	0	3	0	0
With map aggregation	3	0	0	0



(a) Overall job completion time. (b) Cross-node traffic.

Fig. 5. Performance comparison.

scheduling and ShuffleWatcher) that do not consider the dependency. These schedulers will randomly place reduce tasks in the first level jobs, thus the dependent data (i.e., output of first level jobs) is spread out in the cluster. Note that the randomness comes from dependency unawareness. For example, job 1's reduce tasks schedule is viewed as random with respect to that of job 2, and vice versa. Here R^1_{Ji} and R^2_{Ji} refer to the output data from reduce tasks 1 and 2 in job i. For the jobs in level two, such dependent data distribution requires cross-node data read in either map or reduce stage.

In fact, if we carefully place reduce tasks, the output data, on which jobs in next level depend, can be aggregated within one node as shown in Figure 4(c). We can then co-locate the tasks of subsequent jobs with the data they depend on. By doing this, the jobs can run entirely on the same node to utilize data locality (i.e., running job 9 on node 1 and job 10 on node 2). In Figure 5, we compare these two different schedulers, i.e., dependency-unaware vs. dependency-aware, in terms of normalized overall job completion time and cross-node traffic. Results show that job completion time can be reduced by 22% and the cross-node traffic can be reduced by 25% by making scheduler dependency aware.

Table 1 gives the detailed network traffic in each stage of job 1 and job 9. For dependency-aware scheduler, the traffic incurred by reduce tasks in job 1 remains the same as the dependency-unaware scheduler. However, the traffic incurred by job 9 is zero. In contrast, dependency-unaware scheduler leads to more network traffic in reduce stage since the dependent data is spread in the cluster (we assume that all tasks in map stage can achieve locality).

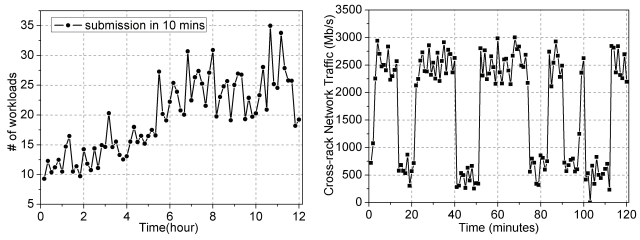
2.3.2 Map Task Scheduling

In this section, we analyze the impact of map task scheduling to the dependency-aware scheduler. If map tasks are scheduled to one slave in order to aggregate intermediate data, the remote input read leads to network traffic. However, benefit is that the traffic from reduce tasks becomes zero when we schedule the tasks to the slave with aggregated intermediate data. Detailed network traffic of job 1 and job 9 in this scene is shown in Table 1. Note that, the total network traffic is not further reduced compared to the dependency-aware scheduler without map aggregation. This is because we assume that, without map aggregation, all map tasks can achieve data locality. However, previous researches have shown that map stage locality is hard to be fully achieved [3], [10]. Thus, map task scheduling in aggregate fashion could further reduce total network traffic.

Map aggregation incurs the remote input read when the input data is spread in the cluster. In fact, for all jobs within a DAG, only a few of them, which does not depend on other jobs, contain the spreading input data. For most jobs that depend on others (80% as analyzed in Section 2.2), their input data (i.e. dependent data), which comes from the output of prior jobs, has been aggregated through the reduce task scheduling. Thus, for these jobs, the map aggregation would not introduce traffic anymore. Such benefit would far exceed the network traffic sacrifice of map aggregation in the few jobs.

Overall, with dependency-aware scheduler, we proactively aggregate intermediate/dependent data through map/reduce task aggregation to improve data locality in subsequent jobs. Although the task aggregation in map stage may incur extra network traffic, it is worthwhile noting the resulted higher data locality in reduce stage and jobs in the subsequent level.

In above case study, we only focus on where to schedule tasks based on the dependency. In a more complex scenario, we need to know when to schedule tasks to their aggregated destinations. Next, we discuss dynamic network status and how to adapt to such dynamics in order to reduce the impact of extra traffic caused by map aggregation.



(a) Workload submission rate. (b) Network traffic profile.

Fig. 6. Analysis of the Google trace.

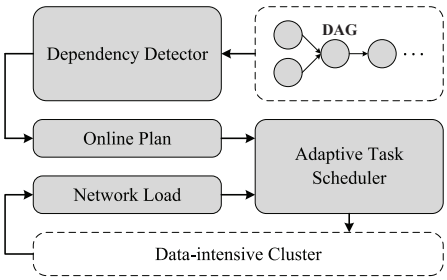


Fig. 7. The architecture of Dawn.

2.4 Dynamic Network Status in Clusters

Data-intensive parallel jobs not only involve network-intensive phases such as remote input data read and intermediate data shuffle, but also contain network non-intensive phases which do not exacerbate the load on cluster network. These non-intensive phases refer to CPU, memory or disk intensive phases such as computation and sort in reduce tasks. The overlapping of network-intensive phases results in high network load in a cluster. Meanwhile, the overlapping of the network non-intensive phases leads to low network load.

Previous work has shown that cross-rack network load is quite saturated during some periods, while unsaturated during others when the job submission follows an exponential distribution [3]. To understand the network load in clusters with production workloads, we analyze the publicly available trace from Google datacenters [30], [31]. Specifically, we run mixed ML workloads from Table 2 using Fair scheduler in a 25-node physical cluster with 6 racks. The workload submission follows the submission rate in the Google trace for a period of 12 hours as shown in Figure 6(a). In the 12-hour period, 1,417 workloads containing total 38,572 jobs are submitted. Figure 6(b) shows the measured cross-rack network traffic in the final two hours. From the figure, we can see that network load is quite bursty and saturated during some periods (e.g., 4th min to 14th min), while leaving it under-utilized during other periods (e.g., 40th min to 52th min). These unsaturated periods offer an opportunity for network adaptive scheduling.

Summary: To leverage the opportunity of dynamic network load, we can schedule a task (e.g., map) that incurs network traffic during unsaturated periods. Meanwhile, we can schedule the task that benefits from data aggregation to improve data locality and free up network bandwidth during saturated periods. Thus, we need an online plan to determine suitable locations for data aggregation, and a network adaptive scheduler to select the most suitable task from parallel jobs to schedule in real time, in order to maximize data locality while improving network utilization and cluster throughput.

3 System Design

In this section, we present the design of Dawn that co-locates aggregated data and tasks to achieve good rack-level data locality based on job dependencies. Dawn first detects job dependencies from the workload DAG. It then employs an online plan to determine the preferred racks for tasks in each job. Based on network status and online plan, Dawn adaptively schedules the most suitable task to the rack with free resource. Figure 7 shows the architecture of Dawn. We describe the functionality of each component as follows:

- **Dependency detector** takes the workload DAG as input and extracts the dependent relationship among jobs.
- **Online plan** determines the preferred racks of each job based on its current stage (e.g., map or reduce). Tasks in a job are only scheduled to preferred racks of the job. For each preferred rack, online plan also estimates the network traffic when tasks are scheduled to the rack.
- **Adaptive task scheduler** monitors the network status of the cluster and schedules tasks according to preferred racks in the online plan. When there is a free resource in a rack available for task scheduling, the scheduler selects a job whose preferred racks include this rack and schedules one task from the job.

3.1 Dependency Detector

In a workload, job dependencies are often described in the form of Directed Acyclic Graph which is available in the workload documentation. Each vertex represents a job and each edge a dependency. As one example, we can interpret the edge (*x*,*y*) as job *y* relies on the output of job *x*. For a non-iterative workload (e.g, query), DAG depth is deterministic. For an iterative workload (e.g., iterative ML), DAG depth is in direct proportion to the number of iterations. If the convergence threshold is pre-defined by users, the number of iterations would be fixed up to the threshold and thus the depth is deterministic. For ML workloads evaluated in this paper, we set the fixed number of iterations based on the default convergence threshold in the benchmarks.

The dependency detector extracts the complex dependent relationship among jobs in two steps. First, we identify these inter-job dependencies in the DAG and store the dependencies in matrix *D*, where the non-zero element $d_{x,y}$ refers to the dependency between job *x* and *y*. Second, for each job in the DAG, we profile the dependency of this job. For certain job *x*, several next level jobs could read its output data (i.e., dependent data). Meanwhile, the next level jobs may also read the output/dependent data from other jobs.

For ease of explanation, we introduce some notations. We use $Next_x$ to represent the set of subsequent jobs that depend on job *x*: $Next_x = \{\text{job } y \mid d_{x,y} = 1\}$. For each job *y* in $Next_x$, its dependent data may come from several previous jobs. So we use $Prev_{x,y}$ to present the set of the previous jobs except job *x*: $Prev_{x,y} = \{\text{job } z \mid d_{z,y} = 1, z \neq x\}$. The dependency of job *x* consists of the set $Next_x$ and all sets $Prev_{x,y}$ of each job *y*. To highlight the output of

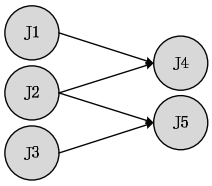


Fig. 8. Two levels DAG workload.

dependency detector, Figure 8 shows a simple example with 5 jobs at two levels. For job 2, the set $Next_2$ = {job 4, job 5}, which represents next jobs. Meanwhile, the dependent data of job 4 and job 5 also comes from one extra job. So the $Prev_{2,4}$ = {job 1} and $Prev_{2,5}$ = {job 3}.

3.2 Online Plan

Online plan determines which racks are preferred to schedule tasks during each stage within the job. It is comprised of two components: (a) proactive task aggregation and (b) dependency-aware task assignment. The former component focuses on aggregating tasks in early stages to realize intermediate data aggregation within the job, so that the tasks in the subsequent stages can be assigned to the same racks to achieve data locality. The latter one determines the preferred racks for tasks in the final stage. It leverages the aggregated intermediate data to exploit data locality and proactively maximizes dependent data aggregation to further benefit the proactive task aggregation of future jobs.

Both two components are to maximally exploit the rack level data locality that refers to input data locality and intermediate data locality. That is to minimize the cross-rack network traffic. Therefore, given *R*-rack cluster, we formulate it as a network optimization problem:

$$minimize \ \ Network\_Cost_r \tag{1}$$

where $Network\_Cost_r$ represents the estimated traffic that will be incurred when we schedule tasks in rack *r*. We model $Network\_Cost_r$ based on different job stages.

For a MapReduce job, the proactive task aggregation is performed in map stage while the dependency-aware task assignment is performed in reduce stage. Below, we first present the two components for MapReduce jobs in map and reduce stage separately, and then show how to extend the components to support general jobs.

3.2.1 Proactive Task Aggregation in Map Stage

The execution of a MapReduce job consists of two network intensive stages: (a) the map stage which reads input data and (b) the reduce stage which shuffles the intermediate data generated by map stage. Since each job gets only a fraction of the cluster resources for its execution in parallel jobs cluster, there is low intra-job concurrency between different stages. So we assume here that these stages run sequentially for simplicity as previous works [32].

In traditional map stage, task is assigned to a slot/container wherever there is available resource. Thus, the intermediate data is scattered in the cluster and reduce tasks often need to read data remotely. In our plan, we aim to proactively aggregate map tasks within one rack, so that tasks in the subsequent stage can be assigned to the same rack to achieve data locality.

Network traffic model: For *R*-rack cluster with *N* machines per rack, the location of job *x* input data in the cluster is defined as an array $\overrightarrow{In_x^{Rack}}$ that contains *R* array elements $\overrightarrow{In_{x,r}^{Node}}$ (*r* ranges from 1 to *R*) representing the location within each rack *r*. For popular file system HDFS [33] with three input data replications, the calculation of $\overrightarrow{In_{x,r}^{Node}}$ also involves the extra replications. We then define the $\left|\overrightarrow{In_x^{Rack}}\right|$ as total input blocks number in the cluster and $\left|\overrightarrow{In_{x,r}^{Node}}\right|$ as the number of different input blocks in the *r*th rack. When we aggregate the map tasks in rack *r*, $Network\_Cost_r$ can be modeled as the sum of blocks that are not available on rack *r*:

$$Network\_Cost_r = R\_Cost_r^x(In) = \left|\overrightarrow{In_x^{Rack}}\right| - \left|\overrightarrow{In_{x,r}^{Node}}\right| \tag{2}$$

Preferred rack: To minimize $Network\_Cost_r$, we select the rack max_map with max available blocks ($max\{\left|\overrightarrow{In_{x,i}^{Node}}\right|, i = 1 \ ... \ R\}$) as preferred location. So the minimum traffic $R\_Cost_{min}^x(In)$ can be represented as:

$$R\_Cost_{min}^x(In) = \left|\overrightarrow{In_x^{Rack}}\right| - \left|\overrightarrow{In_{x,max\_map}^{Node}}\right| \tag{3}$$

For example, for jobs which benefit from the dependency-aware task assignment in previous jobs, their input data has been aggregated to a certain rack, so that the minimum traffic $R\_Cost_{min}^x(In)$ is zero. For other jobs whose input data scatters in the cluster, the $R\_Cost_{min}^x(In)$ could result in extra traffic. However, the dynamic network status allows us to reduce the impact of such traffic through scheduling it during network unsaturated periods.

3.2.2 Dependency-ware Task Assignment in Reduce Stage

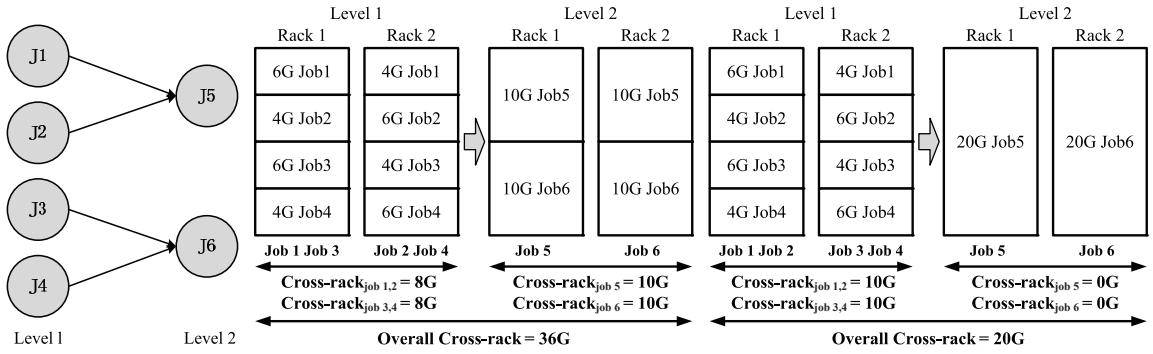
In reduce stage, the placement of reduce tasks not only decides the network traffic comes from the intermediate data shuffle, but also decides the location of output data which could constitute dependent data of jobs in next level. Therefore, we determine the preferred racks with two objectives: minimizing intermediate transfer traffic for current job and maximizing dependent data aggregation for jobs in next level.

Network traffic model in intermediate transfer traffic: We use $\left|\overrightarrow{Inter_{x,r}^{Node}}\right|$ to represent the size of intermediate data in rack *r*. So the $Network\_Cost_r$ is modeled as the sum of intermediate data stored in other racks due to no replications.

Preferred rack: To minimize intermediate transfer traffic, we select the rack max_reduce which contains the maximum intermediate data ($max\{\left|\overrightarrow{Inter_{x,i}^{Node}}\right|, i = 1 \ ... \ R\}$) as preferred one. The minimum traffic $R\_Cost_{min}^x(Inter)$ is:

$$R\_Cost_{min}^x(Inter) = \sum_{i=1, i\neq max\_reduce}^{R} \left|\overrightarrow{Inter_{x,i}^{Node}}\right| \tag{4}$$

Network traffic model in dependent data aggregation: For maximizing dependent data aggregation, we first obtain



(a) Dependencies. (b) Dependency-unaware scheduling. (c) Dependency-aware scheduling.

Fig. 9. Cross-rack network traffic reduction with online plan.

the set $Next_x$ and all sets $Prev_{x,y}$ from the dependency detector. The location of output data for job x is defined as $\overrightarrow{Out_x^{Rack}}$. For each job y in $Next_x$, its input data comes from both job x and the sum of each job i in $Prev_{x,y}$. So the input data location is:

$$\overrightarrow{In_y^{Rack}} = \overrightarrow{Out_x^{Rack}} + \sum_{i \in Prev_{x,y}} \overrightarrow{Out_i^{Rack}} \quad (5)$$

We then obtain the $R\_Cost_{min}^{y}(In)$ of each job y in $Next_x$ based on each $\overrightarrow{In_y^{Rack}}$. Note that, the sum of these jobs input cost represents the network cost related to the dependent data from job x. So the $Network\_Cost_r$ is:

$$Network\_Cost_r = \sum_{i \in Next_x} R\_Cost_{min}^{i}(In) \quad (6)$$

Preferred rack: To minimize $Network\_Cost_r$, we need to know the $\overrightarrow{Out_i^{Rack}}$ of each job in advance. However, when we schedule tasks, only the output of finished jobs are available. So we use a greedy algorithm to solve the problem. We calculate input data based on currently finished jobs. That is, job i in $\sum_{i \in Prev_{x,y}} \overrightarrow{Out_i^{Rack}}$ must be finished. We select the rack dep_reduce that minimizes the current $Network\_Cost_r$ as preferred location. The greedy strategy is to achieve currently optimal dependent data aggregation and make such choice at each schedule to find the global optimum.

After obtaining the rack dep_reduce, we calculate the network traffic caused by intermediate transfer in job x:

$$R\_Cost_{dep}^{x}(Inter) = \sum_{i=1, i \neq dep\_reduce}^{R} \left| \overrightarrow{Inter_{x,i}^{Node}} \right| \quad (7)$$

Summary: Both dep_reduce and max_reduce are preferred racks in reduce stage. For jobs whose $dep\_reduce = max\_reduce$, scheduling tasks on preferred racks results in both minimum network traffic and maximum data aggregation. For other jobs whose $dep\_reduce \neq max\_reduce$, $R\_Cost_{dep}^{x}(Inter)$ is greater than $R\_Cost_{min}^{x}(Inter)$ and scheduling tasks on dep_reduce brings more traffic. However, the dynamic network status allows us to reduce the impact of such traffic, achieving better data aggregation to benefit the proactive task aggregation in further jobs.

For example, Figure 9 illustrates a simple example showing online plan's cross-rack traffic reduction for 6 jobs. The data size ratio between different stages is 1:1 for simplicity. Figure 9(a) shows the dependency relationship between these jobs. The cluster level 1 profiles in Figures 9(b) and 9(c) show the intermediate data locations on two racks for level 1 jobs. When scheduling the one reduce task of each level 1 job, ShuffleWatcher assigns the reduce task on the rack with more intermediate data (i.e., job 1 on rack 1, job 2 on rack 2) as shown in Figure 9(b). While this schedule can reduce the traffic within the two jobs and only cause 8G+8G cross-rack intermediate data shuffle, the dependent data for the next level jobs is evenly distributed as shown in the cluster level 2 profile. So jobs 5 and 6 have to read 10G+10G data using cross-rack links and the overall cross-rack data is 36G. Note that the 10G+10G traffic does not necessarily come from map stage. If jobs 5 and 6 are launched without map locality, the traffic comes from map tasks. If map locality is achieved, the traffic comes from reduce tasks due to shuffling.

Figure 9(c) shows the preferred rack choice considering dependency. While the schedule in transferring intermediate data sacrifices part of intra-job locality and results in more traffic (i.e., 10G + 10G), jobs 5 and 6 benefit from the dependent data aggregation through co-locating their tasks with dependent data. With the online plan, the jobs only need to transfer 20G data, reducing the traffic by 44%.

3.2.3 General Jobs

General jobs refer to those generated by frameworks such as Hive and Apache Tez, which usually contain complex DAG of task stages [2]. Such job DAG contains more than two stages of tasks (job DAG differs from workload DAG), while MapReduce job only contains two stages of tasks (i.e., map stage and reduce stage). There are different communication patterns between adjacent stages such as one-to-one, broadcast and scatter-gather as shown in Figure 10.

The input stage and the output stage in general jobs can be modeled as map stage and reduce stage separately. The rest stages shuffle intermediate data from previous stages, just like shuffle operation in MapReduce jobs. However, they only generate intermediate data. They do not generate dependent data for subsequent jobs. Thus, we determine preferred locations of tasks in these stages to minimize intermediate transfer traffic.

To this end, we obtain the intermediate data array $\overrightarrow{Inter_x^{Rack}}$ according to the task placement in the previous stage. We then determine the preferred racks based on the

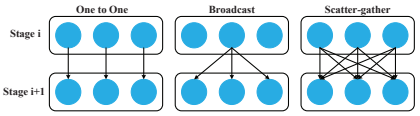


Fig. 10. Three communication patterns between stages.

objective of minimizing intermediate transfer traffic as the model in Eq. (4).

The intermediate data shuffle benefits from the task aggregation in the previous stage, so that the network traffic in a single job without dependency can also be alleviated. This is more evident when the general job contains multiple stages since all subsequent stages, except the input data read stage, can benefit from alleviated network traffic. Though Dawn makes use of dependency to alleviate network traffic, it also works well for a single job without dependency.

3.2.4 Assumption and Discussion

The online plan makes several assumptions. First, we use the whole network traffic between stages to estimate the traffic for each task. This is valid for broadcast and scatter-gather communication pattern. It is also robust for one-to-one case as the data is gathered together most time. Second, input and intermediate data size of each job is assumed to be known. Specifically, the input size (i.e., $\left|\overrightarrow{In_{x,i}^{Node}}\right|, i = 1 \ ... \ R\}$) is obtained from the HDFS since we submit the job after finishing the previous jobs on which it depends. The intermediate data size (i.e., $\left|\overrightarrow{Inter_{x,i}^{Node}}\right|, i = 1 \ ... \ R$) is estimated based on input/intermediate size ratio instead of assuming that intermediate data size is equal to the input size. Finally, we assume that the job output data will be stored in the same rack where the final tasks are executed. The HDFS supports this assumption [34].

Instead of leveraging job relationship at two levels, one can plan ahead with more job levels. However, we found that very few jobs in the third level or beyond benefit from the online plan since the currently preferred racks become stale before their execution. It only incurs more complex online plan calculation without bringing much further improvement.

3.3 Adaptive Task Scheduler

The adaptive task scheduler coordinates task placement with dynamic network status to achieve better rack-level data locality and more balanced network load. It keeps monitoring cross-rack load and compares it to a threshold to determine whether the network is saturated. When there are free resources in the cluster, it employs a heuristic algorithm, as shown in Algorithm 1, to adaptively schedule tasks.

Algorithm 1 contains three steps. First, it updates online plan and obtains the latest preferred racks of each job when the received heartbeat indicates free resources (line 3). Second, it balances online plan if necessary (line 4). Finally, it selects a job based on the network status (lines 5 to 32) and chooses one task from the job based on its current stage.

3.3.1 Online plan update

The online plan is updated according to Algorithm 2. The algorithm first models different $Network\_Cost_r$ for each job based on its current stage. It then determines preferred racks (i.e., max_x and dep_r) and estimates network traffic (i.e., $R\_Cost^x_{min}(In)$, $R\_Cost^x_{min}(Inter)$ and $R\_Cost^x_{dep}(Inter)$). Specifically, for a new submitted job, we calculate its *max_map* and estimate the traffic (lines 3 to 5). For a job that finishes map tasks, we calculate *max_reduce* and *dep_reduce* based on $\sum_{i \in Prev_{x,y}} \overrightarrow{Out_i^{Rack}}$ estimated from currently finished jobs (lines 6 to 12). We also monitor new finished jobs since the last update and recompute *dep_reduce* when the finished jobs belong to the set of previous jobs $Prev_{x,y}$ (lines 13 to 16).

3.3.2 Online plan balancing

For each job, online plan identifies the optimal rack for assigning its tasks to. Naturally, for parallel jobs in clusters, different jobs would have different preferred racks and spread over the cluster. However, there is a possibility that several racks (hotspots) are preferred by most jobs. In this case, for the job that prefers the hotspots, its execution time increases due to intense competition for resources in the hotspots, prolonging the execution time of the workload to which the job belongs. In the extreme case when the hotspots are preferred by all jobs and no job prefers the rest racks, job parallelism and resource utilization in clusters decrease.

To address this issue, we balance the preferred racks in Algorithm 3. The algorithm calculates the number of jobs that prefer rack i (i.e., P_i) and the number of jobs that prefer rack *max* with maximum preferences (i.e., P_max). If P_max is larger than P_i to an extent (decided by balancing factor b), rack *max* is regarded as the hotspot, following which rack i is added into the preferred racks of the job that prefers the rack *max* (lines 8 to 11).

The balancing factor b decides the balancing frequency. The range of b is [0, 1]. In default experiment, b is set to 0 so that online balancing is only triggered when $P\_i = 0$. We explore the impact of b in Section 6.1.4.

3.3.3 Job selection

If the network is saturated, the algorithm sorts jobs in increasing order based on the estimated traffic they would cause (lines 5 to 12). In detail, since each job may contain several preferred racks, the algorithm uses the rack with the smallest traffic to decide the caused traffic for each job in the sorting. From the sorted list, we choose the first job whose preferred racks contain the rack i. That is, we schedule the task from a job which would cause minimum network traffic during the saturated period. If the network is not saturated, the jobs are sorted in decreasing order and a job with the maximum network traffic is selected (lines 13 to 21). Then, the algorithm adds more jobs whose preferred racks contain the rack i with the same amount of traffic as the first job, into the selected jobs set (lines 22 to 27).

If the set only contains one job, we just return the job (lines 28 to 29). Otherwise, we consider the node level data distribution within the rack i (lines 31). That is, we select the job which would cause the minimum or maximum cross-node traffic. For simplicity, we skip the details as it adopts the same principle as the rack-level selection.

Algorithm 1 Adaptive task scheduler

```
Receiving heartbeats from all nodes;
if A heartbeat from the node in rack i indicating free resource then
    Update online plan through Algorithm 2;
    Balance online plan through Algorithm 3;
    if NetworkSaturated is True then
        for each job in the parallel jobs do
            Select the preferred rack with the smallest traffic;
        end for
        Sort jobs in increasing order of selected rack's traffic;
        do
            Remove first job in sorted list;
        while the preferred rack is not rack i
    else
        for each job in the parallel jobs do
            Select the preferred rack with the highest traffic;
        end for
        Sort jobs in decreasing order of selected rack's traffic;
        do
            Remove first job in sorted list;
        while the preferred rack is not rack i
    end if
    Add the latest removed job into the selected jobs set;
    for job i in remaining jobs do
        if job i traffic == traffic of selected jobs then
            Add the job i into the set;
        end if
    end for
    if the set contains one job then
        return the job
    else
        return the job based on node level data distribution
    end if
end if
```

Algorithm 2 Online plan update

```
Set New = new finished jobs from last update;
for each job x in the parallel jobs do
    if max_map is none then
        calculate max_map;
        estimate R_Cost^x_min(In);
    end if
    if map tasks finished and max_reduce is none then
        calculate max_reduce;
        estimate R_Cost^x_min(Inter);
        calculate sum_{i in Prev_{x,y}} Out_i^{Rack};
        calculate dep_reduce;
        estimate R_Cost^x_dep(Inter);
    else if New contains jobs ∈ sets Prev_{x,y} then
        update sum_{i in Prev_{x,y}} Out_i^{Rack};
        update dep_reduce;
        update R_Cost^x_dep(Inter);
    end if
end for
```

3.4 Starvation, Fairness and Fault Tolerance

Algorithm 1 in adaptive task scheduler sorts jobs in increasing or decreasing order of network traffics based on network status. So a job will not always get a low priority, thus starvation is avoided. When a job needs to shuffle data but the current network is saturated, it gets a low priority. However, this situation does not last long since the job can get a high priority when the network becomes unsaturated.

The objective of Dawn, as ShuffleWatcher, is to improve the cluster throughput without considering the scheduling fairness at the job level. Dawn can support user-level fairness by allocating scheduling opportunity for each user based on a fairness policy.

In the default Fair scheduler, if an entire rack fails, for each affected job, it is possible that only a small fraction of tasks have to be repeated since the job spreads its tasks in the cluster. In Dawn, it is possible that all tasks of the affected job have to be repeated since the tasks are aggregated. However, fewer number of jobs would be affected in Dawn due to task aggregation. Indeed, the total number of tasks to be repeated is same for the two schedulers.

Algorithm 3 Online plan balancing

```
Set b = balancing factor;
Obtain online plan results and the rack i with free resources;
Scan the preferred racks of each job;
P_i = # of jobs that prefer rack i;
P_max = # of jobs that prefer the rack max with maximum preferences;
if P_max * b ≤ P_i then
    return
else
    /* Balance online plan */
    Add rack i into the preferred racks of the job that prefer the rack max;
    Estimate the corresponding traffic;
    return
end if
```

4 Implementation

We implemented Dawn on Apache Yarn (version 2.7.2) by modifying the `yarn.server.resourcemanager`.

Dependency detector: We added a new member `yarn.job.dependency` to store the dependency matrix of each workload. The matrix stems from the benchmark documentation which provides the workload DAG (i.e, jobs dependency relationship).

Online plan: We collected the previous task placement from Apache Yarn and stored the output information into a new class called `yarn.job.distribution`. This class also received the result from dependency detector. We detected the different stages of the job through `priority` label provided by Apache Yarn. The calculation in online plan happened in the class `yarn.job.preferredracks`.

Adaptive task scheduler: We used Linux `netstat` to monitor the network status and implemented a distributed monitoring tool based on Remote Procedure Call (RPC) protocol using Python language. The tool provides an interface so that the Java language in Apache Yarn can call its built-in functions. The core scheduling algorithm was implemented in `yarn.server.resourcemanager.scheduler`.

5 Evaluation Setup

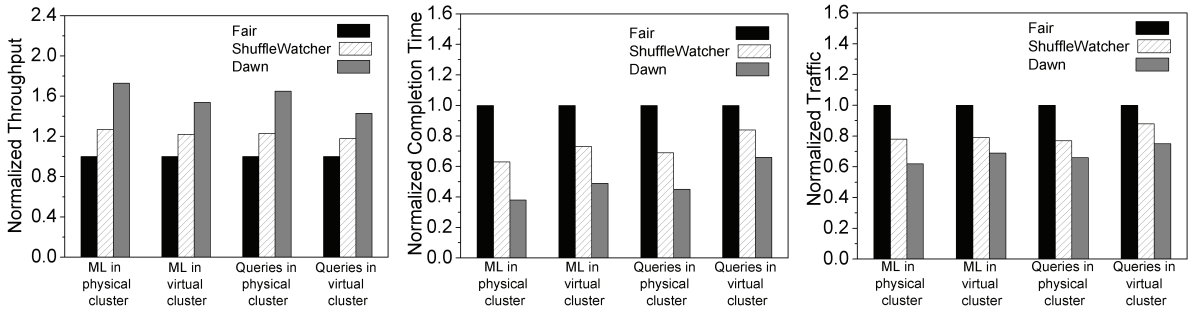
5.1 Testbed

We built two clusters to evaluate the performance of Dawn. **Physical Cluster** contains 25-node physical machines with one master node and 24 slave nodes. We divided the cluster into 6 sub-clusters, each with 4 slave nodes, and identify the sub-clusters by IP addresses. Each machine has 8-core Intel Xeon processors, 132GB of RAM and 5*1-TB hard drivers configured as RAID-5. We configure slave machine to run 6 Yarn containers. The aggregate bandwidth from one sub-cluster to another is 500 MBps (as that in ShuffleWatcher [3]) the bandwidth within each sub-cluster is 10 Gbps.

Virtual Cluster is a 37-node testbed in our university cloud. The cloud runs on 8 HP BL460c G6 blade servers

TABLE 2
ML workloads with different input size.

Benchmark	Label	Input size(GB)	Input data	DAG depth
K-means	B1 / B2 / B3	2.5 / 6.5 / 10.2	HiBench	11
Bayes Classification	B4 / B5 / B6	5.8 / 7.8 / 11.9	HiBench	8
PageRank	B7 / B8 / B9	3.4 / 6.6 / 8.8	HiBench	11
Matrix Factorization (MF)	B10 / B11 / B12	4.5 / 5.5 / 7.3	Mahout	21
Latent Dirichlet Allocation (LDA)	B13 / B14 / B15	6.2 / 8.4 / 10.1	Mahout	18



(a) Throughput. (b) Average WCT. (c) Cross-rack traffic.

Fig. 11. Performance comparison on jobs with dependency in two clusters.

interconnected with 10Gbps Ethernet. The cluster contains one master node and 36 slave nodes. We divided it into 6 sub-clusters, each with 6 slaves. Each slave node was configured to run 2 Yarn containers. We use the network utility tools *tc* and *iptables* to limit the aggregate bandwidth from one sub-cluster to another to 500 MBps, without limiting the bandwidth within each sub-cluster.

5.2 Workloads

We first use workloads consisting of five ML benchmarks drawn from HiBench and Mahout [11] as shown in Table 2. MF [35] uses expectation maximization based alternating least squares algorithm. LDA [36] uses collapsed variational Bayes algorithm. Each benchmark contains several MapReduce jobs with dependency. To emulate a scenario that many different types of ML applications are running in a cluster, we build multiple copies with different input sizes. Benchmark submission follows the submission rate in the Google trace for a period of 4 hours. The virtual cluster uses the first four hours in Figure 6(a) that contains 310 workloads. The physical cluster uses the final four hours that contains 581 workloads. In each submission, we randomly choose a benchmark (i.e., B1 to B15).

To evaluate the benefits of Dawn for complex queries, we then ran 22 queries with 50GB input size from TPC-H benchmark, using Hive 2.0.0. We execute these queries via MapReduce engine, so that each query contains several MapReduce jobs with dependency. Similar to Corral [32], the queries are submitted over a period of 10 minutes in the physical cluster and 25 minutes in the virtual cluster, with arrival times chosen uniformly at random.

As mentioned in Section 3.2.3, Dawn also works for general jobs with multiple stages. To evaluate this, we ran the same TPC-H queries on Apache Tez [2] framework and each Tez job contains multiple stages.

We conducted experiments against each of the three workloads. Each experiment lasted for a steady-state 4 hours [3] and ran five times. We report the average results.

5.3 Baselines and Metrics

We compare Dawn with two online schedulers: Apache Yarn Fair Scheduler (Fair) and ShuffleWatcher. The primary metrics include cluster throughput, average workload completion time (average WCT) and cross-rack network traffic [3], [32]. The cluster throughput denotes the number of finished workloads during the 4 hours experiment. Since number of finished workloads under different schedulers is different, cross-rack network traffic refers to the size of data transferred across different racks per workload.

6 Evaluation

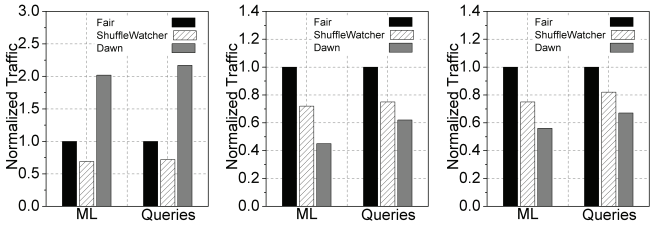
We first show the benefits of using Dawn to schedule MapReduce jobs with dependency against Fair and ShuffleWatcher including sensitivity of Dawn to cluster load and its overhead. We then show the performance improvement on general Apache Tez jobs with multiple stages. Finally, we present the discussion about Dawn.

6.1 MapReduce Jobs with Dependency

6.1.1 Overall Result

Figure 11 compares Dawn against Fair and ShuffleWatcher in three metrics. Overall, Dawn achieves significant improvements over other two schedulers in two clusters. Moreover, performance improvement in the virtual cluster is better than that in the physical cluster. Because there is more background network traffic in the virtual cluster since it is deployed in a shared cloud. The traffic impacts the accuracy of network status estimation using `netstat`.

In the physical cluster, with ML workloads, Dawn achieves significant improvement over Fair, with 73% higher throughput, 66% lower average WCT time and 34% lower cross-rack traffic. The experiment on query workloads shows similar results (e.g., 54% throughput improvement and 48% lower average WCT). Fair only employs delay scheduling technique to optimize map tasks locality. Dawn's



(a) Input data. (b) Dependent data. (c) Intermediate data.

Fig. 12. Cross-rack traffic comparison on jobs with dependency.

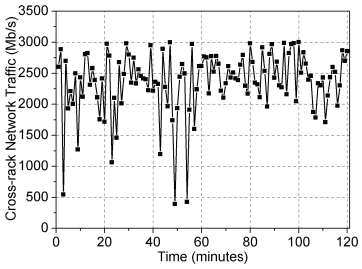
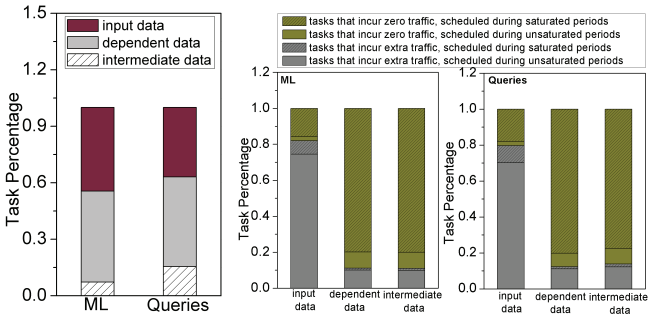


Fig. 13. Network traffic profile under Dawn in the physical cluster.



(a) Tasks percentage. (b) Tasks scheduled during network saturated and unsaturated periods.

Fig. 14. Tasks of Dawn in the physical cluster.

improvements result from better locality for tasks in both subsequent stages and future jobs. In virtual cluster, Dawn improves throughput over Fair by 65% and 43% for ML and query workloads, respectively.

ML workloads achieve better performance improvement than queries. The reasons are twofold. On the one hand, the submission rate in the Google trace results in that the clusters running ML workloads have high load. Section 6.1.4 further evaluates Dawn's sensitivity to cluster load. On the other hand, data size of subsequent jobs in ML workloads is larger than that in queries as shown in Section 2.2 so that more data locality in ML workloads can benefit from dependent data aggregation in previous jobs.

ShuffleWatcher optimizes each job individually for both map and reduce stages and it results in an obvious improvement over Fair. However, such scheduling in ShuffleWatcher ignores data dependency between jobs, so the dependent data is spread over the cluster. This leads increased network traffic and completion time for all jobs in the subsequent levels. Thus, in the physical cluster, Dawn results in higher throughput (38% and 25%) and lower average WCT (35% and 31%) compared to ShuffleWatcher for two workloads as shown in Figures 11(a) and 11(b) respectively. For cross-rack traffic, although the scheduling in Dawn may sacrifice part intra-job locality as described in Figure 9, the overall cross-rack traffic is still 38% and 32% lower than ShuffleWatcher respectively. In the virtual cluster, Dawn improves throughput over ShuffleWatcher by 34% and 21% for ML and query workloads, respectively.

6.1.2 Cross-Rack Traffic

Figure 12(a) shows the measured cross-rack traffic for input data read in the first level jobs in the physical cluster. The traffic by Dawn is higher than that by Fair and ShuffleWatcher, because the input data is spread over the cluster so that data aggregation inevitably incurs extra network traffic. However, Dawn can reduce the impact of such traffic through aggregating input data during network unsaturated periods.

Figure 12(b) shows the traffic for dependent data transfer in subsequent dependent jobs. The traffic by Dawn is the lowest since the most dependent data is aggregated in the cluster. Thus, data aggregation, which aggregates map tasks and co-locates the tasks with dependent data, incurs significant less network traffic than that by Fair and ShuffleWatcher. In contrast, dependent data by Fair and ShuffleWatcher is spread over the cluster, leading extra network traffic in the data transfer.

Figure 12(c) shows the traffic for intermediate data shuffle in all jobs. Dawn incurs the lowest traffic since it aggregates intermediate data and co-locates most reduce tasks with the data. Note that traffic reduction in dependent data transfer is more significant than that in intermediate data shuffle, because reduce task scheduling is also responsible for maximizing dependent data aggregation that could fail the co-location and incur extra network traffic. Overall, based on data size comparison in Figure 3 and cross-traffic comparison in Figure 12, network traffic reduction by Dawn in dependent data and intermediate data is more than the extra traffic in input data read due to data aggregation.

Figure 13 shows the measured cross-rack network traffic of Dawn in the physical cluster. Comparing this profile to that in Figure 6(b), we see that the network traffic with Dawn is relatively balanced. Similar cross-rack traffic, omitted in this paper, holds for the virtual cluster.

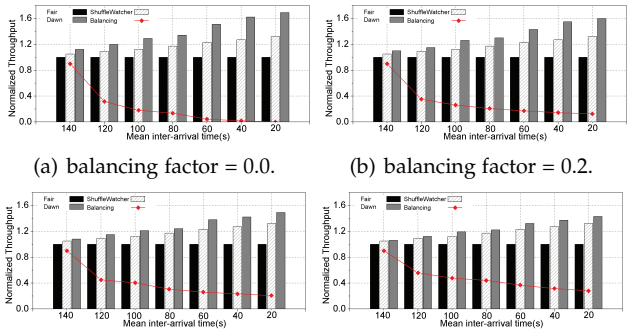
Figure 14(a) shows the percentages of ML and query tasks that process input data, dependent data, and intermediate data separately in Dawn. The results show that tasks that process dependent data and intermediate data significantly outnumber tasks that process input data. Figure 14(b) further plots the percentages of tasks that are scheduled during network saturated and unsaturated periods respectively. The result shows that, for tasks that process input data, 79% of them incur extra network traffic since the input data is spread in the cluster. In contrast, for tasks that process intermediate and dependent data, only 11% of them incur extra network traffic due to data aggregation. Meanwhile, 89% tasks that incur extra network traffic are scheduled during network unsaturated periods. 90% tasks that incur no traffic are scheduled during network saturated periods.

TABLE 3
Average completion time (seconds) of ML workload in the physical cluster.

Scheduler	K-means	Bayes	PageRank	MF	LDA
Fair	1022	819	792	2194	1752
ShuffleWatcher	595	485	499	1265	1037
Dawn	393	321	333	694	562
Time Reduction Over Fair	61%	62%	58%	69%	68%
Time Reduction Over ShuffleWatcher	33%	34%	33%	44%	45%

TABLE 4
Average completion time (seconds) of TPC-H query workload with depth larger than 4 in the physical cluster.

Scheduler	Query									
	2	5	7	8	9	10	11	20	21	22
Fair	139	281	542	425	760	227	80	245	839	151
ShuffleWatcher	95	198	388	322	643	161	62	177	614	110
Dawn	68	149	260	192	422	122	41	125	421	79
Time Reduction Over Fair	50%	47%	52%	55%	44%	46%	47%	49%	50%	47%
Time Reduction Over ShuffleWatcher	28%	25%	32%	40%	34%	24%	32%	29%	31%	29%



(a) balancing factor = 0.0. (b) balancing factor = 0.2.

(c) balancing factor = 0.4. (d) balancing factor = 0.6.

Fig. 15. Sensitivity to cluster loads of ML workloads.

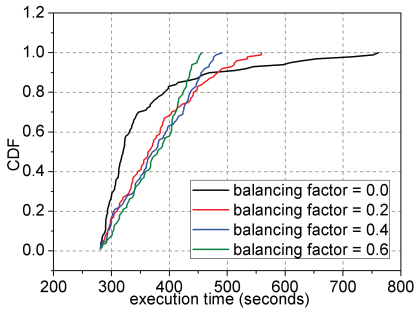


Fig. 16. The CDF of PageRank workload's execution time.

6.1.3 Average Workload Completion Time

Table 3 and Table 4 show the average completion time of five ML workloads and query workloads in TPC-H with depth larger than 5 in the physical cluster, respectively. We can see that Dawn performs better on workloads with deeper DAG (e.g., MF and query 8). Also, the performance enhancement on ML workloads is more significant than on query workloads. On average, DAG depth of each ML workload is 13 which is more than the depth of 6 in query workloads. This leads to more data dependencies and gives Dawn more opportunities to excel. Similar results, omitted in this paper, hold for the virtual cluster.

6.1.4 Sensitivity to Cluster Load

Dawn calculates each job's preferred racks in online plan and then exploits the options among parallel jobs to adapt to the network status. Online plan balancing, i.e., Algorithm 3, is triggered when there are relatively fewer jobs to choose for the rack with free resources. When this happens, Dawn performs only sub-optimally without enough options to exploit in the cluster. In this section, we evaluate Dawn's sensitivity to load using ML workloads.

The evaluation with default balancing factor is conducted in the virtual cluster and the result is shown in Figure 15(a). For ML workloads, we submit the benchmark based on the exponential distribution and vary the mean benchmark inter-arrival time from 20 to 140 seconds to simulate different cluster load conditions. Note that, we increase input size in this evaluation since the exponential submission pattern is different from the Google trace pattern. In addition, we report the normalized online plan balancing frequency under different loads. The figure shows that the performance improvement brought by Dawn is more evident at higher loads due to the increased network traffic under Fair and ShuffleWatcher. Also, online plan balancing is rarely triggered under high loads since the preferred racks from the original online plan can cover all racks in the cluster (i.e., $P\_i > 0$). As the load goes lower, the benefit of Dawn tends to decrease as more network bandwidth becomes available. At the lowest load, Dawn achieves similar performance compared to Fair. In this case, frequent online plan balancing is performed, catering for better cluster resource utilization and job parallelism. Results also show that ShuffleWatcher is sensitive to the load of cluster.

To examine the impact of the factor b, we set it to several nonzero values (i.e, 0.2, 0.4, 0.6) and evaluate Dawn's sensitivity to load respectively. Figures 15(b) 15(c) 15(d) show that, during high cluster loads (20 to 80 seconds), the balancing frequency increases when the factor value increases. This is because a larger factor value leads to a looser condition in Algorithm 3 to balance the online plan. Meanwhile, the throughput decreases when the factor value increases as a larger factor results in more sub-optimal scheduling so that jobs are not scheduled to their preferred

TABLE 5
Overhead (seconds) of Dawn in two clusters.

Overhead	Physical cluster		Virtual Cluster	
	ML	Queries	ML	Queries
Online plan	13.9	9.1	11.1	7.3
Monitor	9.2	9.8	8.9	8.5
Scheduler	9.8	6.9	8.5	5.6
Total	32.9	25.9	28.5	21.4

racks in Dawn.

Figure 16 further plots the completion time distribution of PageRank workload with different factor values when the mean benchmark inter-arrival is 60 seconds. The result shows that the 99^{th} percentile workload completion time decreases when the factor value increases. Therefore, when the workload execution time is constrained by the predefined deadline, we choose a nonzero value for the factor. There is an inverse correlation between the factor value and the deadline value. When we aim to improve the overall cluster throughput instead of meeting workload execution time constraint, we empirically set balancing factor to 0.

6.1.5 Overhead

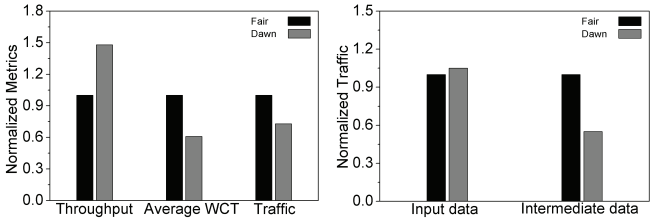
The overhead of Dawn comes from three sources: online plan update, network monitor, and adaptive task scheduler. Online plan uses enumeration method to solve the optimization problem. The RPC communication in network monitor uses a bitvector to indicate network status. Algorithms 1 and 3 in adaptive task scheduler also have low time complexity. To better understand the overhead, we measure the time cost on these three components in two clusters as shown in Table 5. Overall, the overhead is 32.9 and 25.9 seconds for all ML or Query workloads in the physical cluster. For each ML workload, the average overhead is about 0.06 second that is negligible compared to workload execution time shown in Table 3. Similar result holds for each Query workload. Also, the overhead of online plan update and adaptive task scheduler in the physical cluster is larger than that in the virtual cluster since more workloads are submitted to the physical cluster.

6.2 General Jobs with Multiple Stages

We ran the workloads on Apache Tez engine with TPC-H queries to evaluate Dawn's performance on general jobs with multiple stages in the virtual cluster. Note that ShuffleWatcher only supports MapReduce framework, so we compare Dawn with Fair in this experiment.

Figure 17(a) shows Dawn's improvement over Fair under three metrics. From the figure, we see that Dawn is 48%, 39%, and 47% better than Fair in throughput, average WCT and cross-rack traffic respectively. Moreover, compared to the same TPC-H query workloads running on MapReduce framework, the overall improvement on Apache Tez is higher. This is because Apache Tez optimizes the data transfer on disks compared to MapReduce, shifting the bottleneck further towards the network side. Therefore, Dawn shows an even bigger advantage.

Figure 17(b) compares the cross-rack traffic in different stages. The result shows that Dawn achieves more intermediate data shuffle reduction on Apache Tez compared



(a) Performance. (b) Cross-rack traffic.

Fig. 17. Performance improvement on Apache Tez job.

to MapReduce framework. This is because Apache Tez contains multi-stage tasks, so more intermediate data shuffle in subsequent stages can benefit from data aggregation in previous stage.

For the performance evaluation on non-dependent MapReduce jobs, please refer to the supplemental material.

6.3 Discussions

The targeting workload of Dawn is one that contains dependent jobs. For instance, ML and complex query workloads implemented through MapReduce framework have jobs exhibiting multiple levels of dependencies. In those workloads, the input data is processed by the first level jobs and then transmitted among the subsequent dependent jobs so that the communication traffic is non-negligible.

Data aggregation in Dawn is a good performance choice for these workloads since Dawn is aware of network status. For a job in which data aggregation causes extra network traffic, Dawn can reduce the impact of such traffic through scheduling this job during network unsaturated periods.

ML and query workloads can also be implemented through other computing frameworks, such as Spark [37]. Such an implementation only contains one job with multiple stages [38]. Data aggregation may not be able to improve the performance. In the future, we plan to augment Dawn for Spark implementation.

7 Related Work

Improving data-locality in big data clusters has become the focus of several recent works. The techniques in Dawn are related to the following research.

Task placement techniques: Techniques like delay scheduling [10] try to improve the locality of individual tasks (e.g., maps) by scheduling them close to their input. iShuffle [39] reduces the time cost of remote data transfer through pro-actively pushing map output data to nodes via a novel shuffle-on-write operation. FlexMap [40] schedules reduce tasks to the machine that stores most intermediate data to avoid inter-machine shuffling.

ShuffleWatcher [3] achieves better map or reduce locality within a single MapReduce job. Compared to ShuffleWatcher, Dawn's main novelty lies in its dependency awareness, and its technical contributions in the online plan and the adaptive task scheduler. Specifically, in its online plan, Dawn determines the preferred task locations by considering inter-job data dependency. In its adaptive task scheduler, Dawn designs an online plan balancing algorithm that can avoid hotspots and idle racks.

Carbyne [41] and Graphene [42] aim at jobs that have a complex dependency structure and heterogeneous resource demands. In former work, jobs yield fractions of their allocated resources that are rescheduled to improve performance and cluster efficiency. The latter one first schedules troublesome tasks and then schedules the remaining tasks without violating dependencies so as to improve job completion time and cluster throughput. However, these two works focus on intra-job task dependency and their techniques cannot be applied to inter-job data dependency. For instance, Graphene schedules tasks based on task characteristics including task input size, task resource requirement and task execution time. Such task level scheduling strategy cannot be applied to job level, because the basic scheduling unit in Yarn (e.g., Yarn container or Hadoop slot) can run one task only. Graphene is unable to schedule an entire job at a time in Yarn clusters.

Input data placement techniques: The study in [43] proposes a grouping-blocks strategy for Hadoop systems. Such strategy can greatly improve both map and reduce locality. Corral [32] tries to coordinate input data placement with job task placement so that the most jobs can be run entirely on one rack with all their tasks achieving rack-level data locality. The Corral improves job performance upon Fair and ShuffleWatcher as it separates large shuffles from each other and reduces network contention in the cluster. However, for a large DAG, it is trivial that Corral needs to utilize extra network to offline aggregate input data for every job. In contrast, Dawn can dynamically aggregate dependent input data for each job and co-locate the task with data without offline operation.

Data replication techniques: HDFS randomly stores 3 replications of each input block across the entire cluster to be fault-tolerant and improve data locality [1]. Scarlett [44] proposes a proactive replication design that periodically replicates files based on predicted popularity. DALM [45] accommodates data-dependency in a MapReduce framework and proposes a dependency-aware replication strategy for general real-world input data that can be highly skewed and dependent. Unlike Dawn which is the online scheduler, the offline extra replications in these techniques, however, require more disc to store data and network to transfer data.

8 Conclusion

In today's clusters, many real-world workloads inherently exhibit strong inter-job data dependency. However, existing network-aware job schedulers ignore the dependencies and schedule the jobs independently to one another. Driven by the intuition that aggregating and co-locating the data and tasks of dependent jobs offers an extra opportunity of data locality, we proposed and developed Dawn, a dependency-aware network adaptive scheduler for data-intensive parallel jobs with dependencies. Dawn is able to determine the optimal racks for each job and adaptively choose the most suitable job to schedule based on the network status. This helps it to maximally exploit rack-level data locality while at the same time smooth network traffic to improve utilization. We implemented Dawn on Apache Yarn. Our experiments show that Dawn can significantly enhance the job performance compared to two state-of-art approaches. Moreover, our results show that Dawn not only significantly helps the jobs with dependency, but also improves the performance of jobs without dependency.

In future work, we plan to extend Dawn to other computing frameworks (e.g., Apache Spark).

Acknowledgments

Wang, Chen, and Zhou (corresponding author) were supported in part by NSF grant CNS-1422119 in the research.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[2] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1357–1369.
[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 1–12.
[4] W. Chen, J. Rao, and X. Zhou, "Preemptive, low latency datacenter scheduling via lightweight virtualization," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 251–263.
[5] W. Chen, A. Pi, S. Wang, and X. Zhou, "Characterizing scheduling delay for low-latency data analytics workloads," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2018.
[6] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, "Scale-out networking in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 29–41, 2010.
[7] J. Mudigonda, P. Yalagandula, and J. C. Mogul, "Taming the flying cable monster: A topology design and optimization framework for data-center networks." in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 101–114.
[8] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proc. ACM SIGCOMM Comput. Commun.*, 2015, pp. 183–197.
[9] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A survey on data center networking (dcn): Infrastructure and operations," *IEEE communications surveys & tutorials*, vol. 19, no. 1, pp. 640–656, 2017.
[10] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM/USENIX Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[11] "Mahout," https://mahout.apache.org.
[12] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," in *Proc. VLDB Endowment*, 2009, pp. 1626–1629.
[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proc. ACM Symp. Cloud Comput.*, 2013, art. no. 5.
[14] S. Wang, X. Zhou, L. Zhang, and C. Jiang, "Network-adaptive scheduling of data-intensive parallel jobs with dependencies in clusters," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2017.
[15] H. Ke, P. Li, S. Guo, and M. Guo, "On traffic-aware partition and aggregation in mapreduce for big data applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 818–828, 2016.
[16] Q. Xie, M. Pundir, Y. Lu, C. L. Abad, and R. H. Campbell, "Pandas: robust locality-aware scheduling with stochastic delay optimality," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 2, pp. 662–675, 2017.
[17] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Transactions on Networking (TON)*, vol. 24, no. 1, pp. 190–203, 2016.
[18] W. Chen, I. Paik, and Z. Li, "Tology-aware optimal data placement algorithm for network traffic optimization," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2603–2617, 2016.

[19] F. J. Clemente-Castelló, B. Nicolae, M. M. Rafique, R. Mayo, and J. C. Fernández, "Evaluation of data locality strategies for hybrid cloud bursting of iterative mapreduce," in *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, 2017, pp. 181–185.

[20] C. X. Cai, S. Saeed, I. Gupta, R. H. Campbell, and F. Le, "Phurti: Application and network-aware flow scheduling for multi-tenant mapreduce clusters," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 161–170.

[21] Y. Xia, X. S. Sun, S. Dzinamarira, D. Wu, X. S. Huang, and T. Ng, "A tale of two topologies: Exploring convertible data center network architectures with flat-tree," in *Proc. Conf. ACM Special Interest Group on Data Communication*, 2017, pp. 295–308.

[22] S. Legtchenko, N. Chen, D. Cletheroe, A. I. Rowstron, H. Williams, and X. Zhao, "Xfabric: A reconfigurable in-rack network for rack-scale computers." in *Proc. USENIX Symp. Networked Syst. Des. ImplementationNSDI*, 2016, pp. 15–29.

[23] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai, "Hibench: A representative and comprehensive hadoop benchmark suite," in *Proc. Int. Conf. Data Engineering Workshops*, 2010.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[25] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Analysis Machine Intelligence*, vol. 24, no. 7, pp. 881–892, 2002.

[26] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *Proc. AAAI*, 1998, pp. 41–48.

[27] "Tpc-h benchmark," http://www.tpc.org/tpch/.

[28] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[29] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proc. Int. Conf. Extending Database Technology*, 2010, pp. 99–110.

[30] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, 2012.

[31] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," in *Proc. VLDB Endowment*, 2012, pp. 1802–1813.

[32] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: plan when you can," in *Proc. ACM SIGCOMM Comput. Commun.*, 2015, pp. 407–420.

[33] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. ACM SIGOPS Operating Syst.*, 2003, pp. 29–43.

[34] "Hadoop distributed file system," https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[35] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 42–49, 2009.

[36] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal Machine Learning Research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 15–28.

[38] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015.

[39] Y. Guo, J. Rao, D. Cheng, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1649–1662, 2017.

[40] W. Chen, J. Rao, and X. Zhou, "Addressing performance heterogeneity in mapreduce clusters with elastic tasks," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2017, pp. 1078–1087.

[41] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters." in *Proc. Symp. Operating Syst. Des. Implementation*, 2016, pp. 65–80.

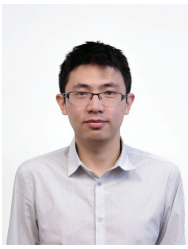
[42] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. Symp. Operating Syst. Des. Implementation*, 2016, pp. 81–97.

[43] X. Yu and B. Hong, "Grouping blocks for mapreduce co-locality," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2015, pp. 271–280.

[44] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proc. ACM/USENIX Eur. Conf. Comput. Syst.*, 2011, pp. 287–300.

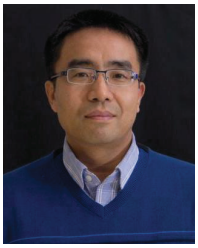
[45] X. Fan, X. Ma, J. Liu, and D. Li, "Dependency-aware data locality for mapreduce," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2014, pp. 408–415.

Shaoqi Wang received the B.S degree in engineering from Anhui Normal University in 2012. He received the M.S degree in engineering from the University of Science and Technology of China in 2015. Currently, he is working toward the Ph.D degree in computer science at the University of Colorado, Colorado Springs. His research interests include big data processing and distributed machine learning. He is a student member of the IEEE.

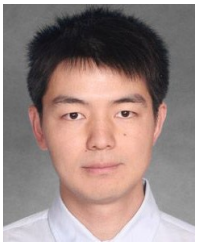


Wei Chen received the B.S. and M.S. degrees in Computer Science from Tongji in 2012 and 2014, respectively. he is working toward the Ph.D degree in computer science at the University of Colorado, Colorado Springs. His research interests include OS virtualization and resource management in datacenter. He is a student member of the IEEE.

Xiaobo Zhou obtained the B.S., M.S., and Ph.D degrees in Computer Science from Nanjing University, in 1994, 1997, and 2000, respectively. Currently he is a Professor with the Department of Computer Science, University of Colorado, Colorado Springs. His research lies in Cloud computing and Datacenters, BigData parallel and distributed processing, autonomic and sustainable computing. He was a recipient of the NSF CAREER Award in 2009. He is a senior member of the IEEE.



Liqiang Zhang received the Ph.D. in Computer Science from Wayne State University in 2005. Currently he is an Associate Professor in the Department of Computer and Information Sciences at Indiana University South Bend. His research interests are in wireless networking and mobile computing, quality of service, distributed and embedded systems, and network security. He is a member of the IEEE.



Yin Wang received the B.S degree in electrical engineering from Shanghai Jiaotong University in 2000, and the Ph.D degree in electrical engineering from the University of Michigan, Ann Arbor, in 2009. He is Professor of the Computer Science Department at Tongji University, Shanghai. His current research interest lies in machine learning and BigData scalable systems. He is a senior member of the IEEE.