Network-Adaptive Scheduling of Data-Intensive Parallel Jobs with Dependencies in Clusters

Shaoqi Wang*, Xiaobo Zhou*, Liqiang Zhang† and Changjun Jiang‡

*Department of Computer Science, University of Colorado, Colorado Springs, USA
†Department of Computer & Information Sciences, Indiana University South Bend, South Bend, USA
‡The Key Laboratory of Embedded System and Service Computing, Tongji University, China
Email addresses: swang@uccs.edu, xzhou@uccs.edu, liqzhang@iusb.edu, cjjiang@tongji.edu.cn

***Abstract*—The performance of data-intensive parallel jobs is often constrained by the cluster's hard-to-scale network bisection bandwidth. Previous solutions 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 offers an extra opportunity of data locality that can help to greatly enhance the performance of clusters and jobs. We propose and design Dawn, a network-adaptive scheduler that includes an online plan and an adaptive task scheduler for jobs with dependencies. The online plan, taking job dependencies into consideration, determines preferred locations (e.g., racks) for tasks to proactively aggregate dependent data. The task scheduler, based on the output of online plan and dynamic network bandwidth status, adaptively schedule 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 clusters using various benchmark workloads. Results show that Dawn effectively improves system throughput by 42-51% and 21-28% compared to Fair Scheduler and ShuffleWatcher, respectively.**

I. Introduction

Data-parallel clusters usually deploy highly scalable computing frameworks like Hadoop [1], Tez [2] or Spark [3] to run data-intensive parallel jobs for improving utilization and cost efficiency. However, the performance of parallel jobs is often constrained by the cluster's hard-to-scale network for several reasons. First, jobs use cross-rack bandwidth to read input data, as input data is randomly spread across several racks in a cluster. Second, jobs usually consist of intermediate data transfer stages such as shuffle and join. The above two network-intensive stages transfer a large amount of data across the network. Finally, when the data flows generated by consecutive jobs are dependent, cross-rack bandwidth can also be used for data transfer between jobs.

Several previous job schedulers (e.g., FIFO, Capacity [4] and Fair [5]) try to overcome this problem by employing the technique like delay scheduling [6] to optimize input data locality, but do not address other network-intensive stages such as shuffle and join. Recent effort ShuffleWatcher [7] 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 within a single job. However, it does not consider data dependency among jobs, but schedules them independently from one another.

Many real-world workloads, such as iterative machine learning (ML) jobs [8] and complex queries [9], inherently exhibit strong dependency. Within such workloads, a job often relies on the output of other jobs, and it has to wait until all the parent jobs are finished. Such dependencies are usually presented as Directed Acyclic Graph (DAG). However, without considering inter-job data dependency, previous schedulers would blindly place the final tasks from the prior level in different racks. Such placements may result in dependent data being spread over the cluster randomly, therefore the subsequent jobs have to read data using cross-rack links. This incurs significant network traffic and prolongs the average job completion time.

In this paper, we propose Dawn, a Network-adaptive scheduler that takes job dependency into consideration and utilizes opportunistic network bandwidth to improve data locality. The key is to assign tasks generating dependent data in a gathered manner, in contrast to existing approaches where tasks scatter in the cluster.

Dawn includes an online plan and an adaptive task scheduler. The online plan decides the preferred racks for the tasks of each job in different stages based on their data locations and job dependencies. The decision aggregates tasks in early stages to realize intermediate data aggregation. For tasks in the final stage, the decision leverages the aggregated intermediate data to exploit data locality, and proactively maximizes dependent data aggregation to further benefit future jobs that are dependent on this current one. The adaptive task scheduler, when it sees a rack has free resource, tries to pick the most suitable job to schedule on that rack. It leverages the available network bandwidth to schedule tasks in a gathered manner to aggregate intermediate or dependent data during unsaturated network periods. During saturated periods, it achieves data locality and frees up more network bandwidth through co-locating the aggregated data with tasks in the subsequent stages or jobs.

We implement Dawn on Apache Yarn, and evaluate it in clusters using various MapReduce and Tez benchmark workloads that contain iterative ML applications and complex queries. Results show that Dawn effectively improves job throughput by 51% and 28%, and average job completion time by 42% and 21%, compared to those by Fair Scheduler and ShuffleWatcher, respectively.

2474-0756/17 $31.00 © 2017 IEEE
DOI 10.1109/ICAC.2017.13

IEEE computer society

II. Case Study and Motivation

A. Dynamic Network in Clusters

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

Previous work has shown that network load is quite saturated during some periods, while unsaturated during others [7]. Cross-rack network traffic is much higher than cross-node since large clusters usually have bandwidth over-subscription ratios ranging from 5:1 to 20:1 [10], [11]. These unsaturated periods offers an opportunity for creating a more smooth and balanced cluster network load.

B. Case Study

In many real-world 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 demonstrate that the scheduling for jobs with dependencies can achieve better data locality, we simulate cross-rack traffic via a small scale cross-node case study. We created a 5-node cluster in our university cloud testbed which ran 12 parallel MapReduce jobs with dependencies. The cluster was configured with one master and four slave nodes. Figure 1(a) shows the dependency relationship between jobs. Each job contains 4 map and 2 reduce tasks. Figure 1(b) shows the schedule of reduce tasks by existing schedulers (e.g., delay scheduling and ShuffleWatcher) that do not consider dependency. Those schedulers will randomly place reduce tasks in prior jobs, thus the dependent data 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/dependent data from reduce tasks 1 and 2 in job i. Such dependent data distribution requires cross-node data reading in the subsequent jobs.

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 1(c). Then we can 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). We compared these two different schedulers, i.e., dependency-unaware vs. dependency-aware, and experimental results show that the dependency-aware scheduler can reduce the completion time by **29.5%** and the cross-node traffic by **26.5%**.

With the scheduler for jobs with dependencies, we proactively aggregate dependent data. Although this aggregation may incur network traffic, we reduce its impact on network load by arranging the aggregation to happen only during unsaturated periods. It is worthwhile noting that the resulted higher data locality will also help to free up network bandwidth. Nevertheless, we need a careful online plan to determine suitable locations for data aggregation, and a network-adaptive scheduler to pick the most suitable jobs to schedule in real time, in order to maximize data locality while improving network utilization.

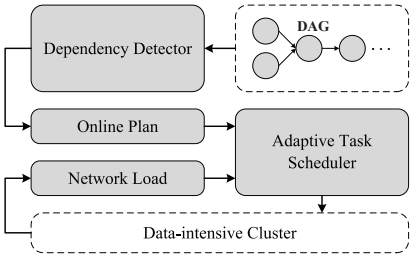


Fig. 2. The architecture of Dawn.

III. System Design

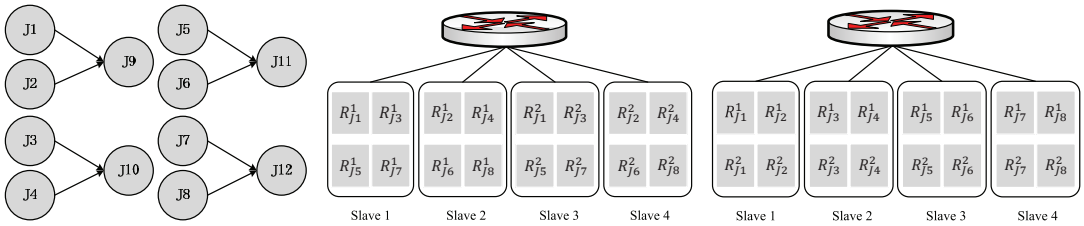
In this section, we present the design of Dawn which co-locates aggregated data and tasks to achieve better rack-level data locality based on job dependencies. Figure 2 shows the architecture of Dawn. We describe the functionality of each component as follows:

- **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).
- **Adaptive task scheduler** monitors the network status of the cluster and uses online plan's output as guidance to adaptively schedule tasks.

A. Dependency Detector

In a workload, job dependencies are often described in DAG form that 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*. We identify such inter-job dependencies by profiling the DAG and store in matrix *D*, where the non-zero element $d_{x,y}$ refers to the dependency between job *x* and *y*.

The dependency detector aims to extract the complex dependent relationship among jobs, so that online plan can use this information to guide the job tasks schedule. For ease of explanation, we introduce some notations. We use the set $Next_x$ to represent the next jobs depending 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 one or more previous jobs. We define the set $Prev_{x,y}$ as these previous jobs except job *x*: $Prev_{x,y}$ = $\{\text{job } z \mid d_{z,y} = 1, z \neq x\}$. The set $Next_x$ and all sets $Prev_{x,y}$ of each job *y* represent the relationship between dependent data and the output of job *x*.



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

Fig. 1. A case study of jobs with dependencies.

B. Online Plan

Online plan 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. The goals of both two components are to maximally exploit the rack level data locality that refers to both input data locality and intermediate data locality. That is to minimize the cross-rack network traffic $Network\_Cost$. We model such traffic 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 (e.g., Hive or Tez).

1) Proactive Task Aggregation in Map Stage: We assume here that map and reduce stages run sequentially for simplicity since there is low intra-job concurrency between them in parallel jobs clusters [12]. In our plan for map stage, 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.

For *R*-rack cluster with *N* machines per rack, we define the $\left|\overrightarrow{In_{x,r}^{Node}}\right|$ as the number of input blocks in the *r*th rack and $\left|\overrightarrow{In_{x}^{Rack}}\right|$ as the total blocks number in the cluster. When we aggregate map tasks in rack *r*, traffic $Network\_Cost_r^{In}$ can be modeled as the sum of blocks that are not available in this rack: $\left|\overrightarrow{In_{x}^{Rack}}\right| - \left|\overrightarrow{In_{x,r}^{Node}}\right|$.

To minimize the traffic, we select the rack max_r with max available blocks ($max\{\left|\overrightarrow{In_{x,i}^{Node}}\right|, i = 1 \cdots R\}$) as preferred location. So the minimum traffic is:

$$Network\_Cost_{Min}^{In} = \left|\overrightarrow{In_{x}^{Rack}}\right| - \left|\overrightarrow{In_{x,max\_r}^{Node}}\right| \quad (1)$$

2) Dependency-ware Task Assignment in Reduce Stage: In reduce stage, the network traffic comes from the intermediate data transfer. We use $\left|\overrightarrow{Inter_{x,r}^{Node}}\right|$ to represent the size of intermediate data in each rack. When we place reduce tasks in rack *r*, the traffic $Network\_Cost_r^{Inter}$ is modeled as the sum of intermediate data stored in other racks. Meanwhile, the placement of reduce tasks decides the location of output data which could constitute dependent data of jobs in next level. So 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.

For minimizing intermediate transfer traffic, we select the rack max_r which contains the maximum intermediate data ($max\{\left|\overrightarrow{Inter_{x,i}^{Node}}\right|, i = 1 \cdots R\}$) as preferred one. So the minimum traffic is:

$$Network\_Cost_{Min}^{Inter} = \sum_{i=1, i \neq max\_r}^{R} \left|\overrightarrow{Inter_{x,i}^{Node}}\right| \quad (2)$$

For maximizing dependent data aggregation, we first obtain 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:

$$\overrightarrow{In_y^{Rack}} = \overrightarrow{Out_x^{Rack}} + \sum_{i \in Prev_{x,y}} \overrightarrow{Out_i^{Rack}} \quad (3)$$

From $\overrightarrow{In_y^{Rack}}$, we can get the number of input blocks in each rack and the total blocks number in the cluster. Then we obtain the input traffic $Network\_Cost_{Min}^{In}$ of each job *y* in $Next_x$ based on the proactive task aggregation. Note that, the sum of these jobs input traffic represents the network cost related to the dependent data from job *x*. It is modeled as:

$$Network\_Cost_{Dep}^{In} = \sum_{i \in Next_x} Network\_Cost_{Min}^{In}(i) \quad (4)$$

Based on enumeration method, we select the rack dep_r that minimizes $Network\_Cost_{Dep}^{In}$. Then we calculate the network traffic caused by current intermediate transfer in job *x*:

$$Network\_Cost_{Dep}^{Inter} = \sum_{i=1, i \neq dep\_r}^{R} \left|\overrightarrow{Inter_{x,i}^{Node}}\right| \quad (5)$$

Both max_x and dep_r are preferred racks in reduce stage. Their corresponding traffics are $Network\_Cost_{Min}^{Inter}$ and $Network\_Cost_{Dep}^{Inter}$ separately. The adaptive scheduler chooses one of them based on current network status as shown in Section III-C.

3) General Jobs: General jobs generated by frameworks (e.g., Hive and Tez) usually contain complex DAG of tasks [2]. Note that such job DAG contains several stages of tasks which are not the same as the workload DAG. There are different communication patterns between adjacent stages such as one-to-one, broadcast and scatter-gather.

The input stage and the output stage in general jobs can be modeled as map stage and reduce stage separately. The rest stages perform proactive task aggregation since they only generate intermediate data just like the tasks in map stage. To determine preferred locations of tasks in these stages, we obtain $\left|\overrightarrow{Inter_{x,r}^{Node}}\right|$ from the tasks placement in previous stage. Using this information, we determine the preferred racks based on the objective of minimizing intermediate transfer traffic as the model in Eq. (2).

C. Adaptive Task Scheduler

The adaptive task scheduler coordinates the tasks 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 current 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 components. First, it updates online plan and obtains the latest preferred racks of each job (line 2). Then, it determines the selected jobs based on the network status (lines 3 to 25). If the network is saturated, the algorithm sorts jobs in increasing order based on the estimated traffics they would cause. 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 rack contain free resources. That is, we schedule tasks 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 we select the job with maximum network traffic. After choosing the first job, the algorithm will add more jobs, whose preferred racks also contain the rack *i* with the same amount of traffic as the first job, into the selected jobs set.

Finally, we schedule one job from the set (lines 26 to 30). If the set only contains one job, we just return the job. Otherwise, we consider the node level data distribution within the rack *i*. 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.

D. Implementation

We implemented Dawn on Apache Yarn (version 2.7.2) by modifying the `yarn.server.resourcemanager`. We added a new member `yarn.job.dependency` to store the dependency matrix in workload. The matrix stems from the benchmark documentation which provides the workload DAG (i.e, jobs dependency relationship). For 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. The calculation in online plan happened in the class `yarn.job.preferredracks`. 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`.

Algorithm 1 Adaptive task scheduler

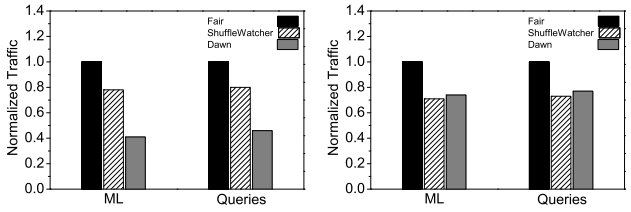
```
1: if A heartbeat received from rack i indicating free resource then
2:     Update online plan
3:     if NetworkSaturated is True then
4:         for each job in the parallel jobs do
5:             Select the preferred rack with the smallest traffic
6:         end for
7:         Sort jobs in increasing order of selected rack's traffic
8:         do
9:             Remove first job in sorted list
10:        while the preferred rack is not rack i
11:    else
12:        for each job in the parallel jobs do
13:            Select the preferred rack with the highest traffic
14:        end for
15:        Sort jobs in decreasing order of selected rack's traffic
16:        do
17:            Remove first job in sorted list
18:        while the preferred rack is not rack i
19:    end if
20:    Add the latest removed job into the selected jobs set
21:    for job i in remaining jobs do
22:        if job i traffic == traffic of selected jobs then
23:            Add the job i into the set
24:        end if
25:    end for
26:    if the set contains one job then
27:        return the job
28:    else
29:        return the job based on node level data distribution
30:    end if
31: end if
```

TABLE I
ML WORKLOAD.

Benchmark	Label	Input size(GB)	Input data
K-means	B1/B2/B3	18.5/37.5/75	HiBench
Bayes classification	B4/B5/B6	34.8/52.8/69.9	HiBench
PageRank	B7/B8/B9	32.4/51.6/64.8	HiBench

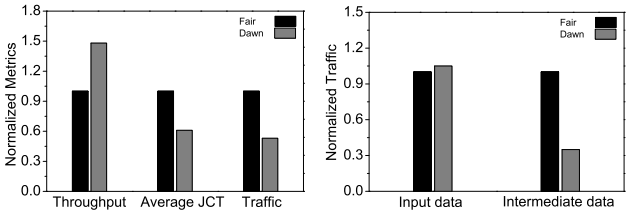
IV. EVALUATION

We evaluate Dawn on a 36-node testbed in our university cloud. We divide our cluster into 6 sub-clusters, each with 6 nodes, and identify the sub-clusters by IP addresses. As in ShuffleWatcher [7], we limit the bandwidth from one sub-cluster to another to 500 Mbps, without limiting the bandwidth within each sub-cluster.



(a) Input data read. (b) Intermediate data transfer.

Fig. 4. Cross-rack traffic comparison on jobs with dependencies.



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

Fig. 5. Apache Tez job experiments.

We use workloads consisting of three ML benchmarks as shown in Table I. Each benchmark contains several MapReduce jobs with dependencies. Benchmark submission follows an exponential distribution [7]. We also use 22 queries with 50GB input size from TPC-H benchmark [13], using Hive 2.0.0. We execute these queries via MapReduce, in which each query contains several MapReduce jobs with dependencies, and Apache Tez [2], in which each Tez job contains multiple stages. The queries are submitted over a period of 25 minutes, with arrival times chosen uniformly at random [12].

We compare Dawn with two online schedulers: Apache Yarn Fair Scheduler (Fair) and ShuffleWatcher. The primary metrics include jobs throughput, average job completion time (average JCT) and cross-rack traffic [7], [12]. Each experiment lasts for a steady-state 4 hours [7].

A. Performance on MapReduce Jobs with Dependencies

We evaluate the benefits of using Dawn to schedule MapReduce jobs with dependencies. Figure 3 compares Dawn against Fair and ShuffleWatcher in three metrics. With the ML workload, Dawn achieves significant improvements over Fair, with 51% higher throughput, 41% lower average JCT and 49% lower cross-rack traffic. The experiment on queries workload shows the similar results (e.g., 42% throughput improvement). Fair only employs delay scheduling technique to optimize map tasks locality. Dawn's improvements result from better locality for tasks in both subsequent stages and future 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, Dawn results in higher throughput (28% and 21%) and lower average JCT (23% and 19%) compared to ShuffleWatcher for two workloads as shown in Figures 3(a) and 3(b) respectively. For cross-rack traffic, although the scheduling in Dawn may sacrifice part intra-job locality, the overall cross-rack traffic is still 26% and 22% lower than ShuffleWatcher respectively.

Figure 4 shows the measured cross-rack traffic for input data read and intermediate data transfer. For intermediate data transfer in ML workload, Dawn and ShufflerWatcher achieve 26% and 29% traffic reduction than Fair respectively since both of them optimize the reduce task schedule. Note Figure 4(b) shows that ShuffleWatcher results in less traffic than Dawn does due to the "sacrifice" in Dawn. However, the input data read traffic in Dawn is much less than both Fair and ShuffleWatcher as shown in Figure 4(a). Thus, Dawn outperforms Fair and ShuffleWatcher.

The performance enhancement on ML workload is more significant than on queries workload. Because, on average, each benchmark in ML workload contains 11 jobs which is more than 6 jobs in queries workload. This leads to more data dependencies and gives Dawn more opportunities to excel.

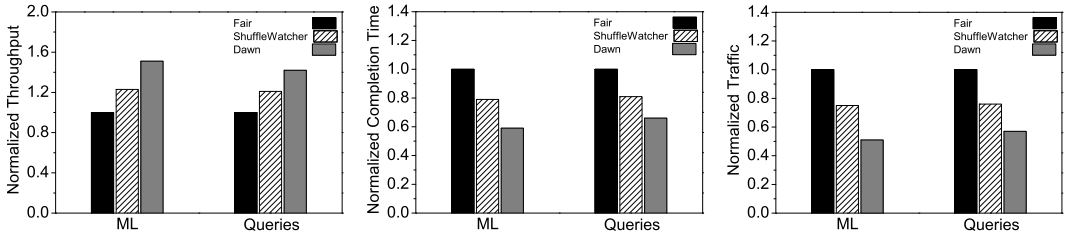
B. Performance on General Jobs with Multiple Stages

To show the performance improvement of Dawn on general jobs with multiple stages, we also ran the jobs on Apache Tez engine with TPC-H queries. Note that ShuffleWatcher only supports MapReduce framework, so we compare Dawn with Fair in this experiment. Figure 5(a) shows the 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 JCT and cross-rack traffic respectively. Moreover, compared to the same TPC-H queries workload 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 5(b) compares the cross-rack traffic in different stages. The result shows that Dawn achieves more intermediate data transfer reduction on Apache Tez compared to MapReduce framework. This is because Apache Tez contains multi-stage tasks, so more intermediate data transfer in subsequent stages can benefit from data aggregation in previous stage.

V. 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 researches.

Task placement techniques: Techniques like delay scheduling [6] and Quincy [14] try to improve the locality of individual tasks (e.g., maps) by scheduling them close to their input. ShuffleWatcher [7] attempts to localize the shuffle phase of a MapReduce job by scheduling reducers on the same set of racks with most mappers. These techniques can achieve better map or reduce locality within a single MapReduce job. However, it does not consider data dependency between jobs and schedules them independently from each other. iShuffle [15]



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

Fig. 3. Performance comparison on jobs with dependencies.

reduces the time cost of remote data transfer through proactively pushing map output data to nodes via a novel shuffle-onwrite operation. ChEsS [16] takes into account the jobs data locality constraints, the clusters processing capabilities and the clusters intra-job scheduling policies to assign jobs.

Input data placement techniques: CoHadoop [17] aims to co-locate different datasets processed by a job on the same set of nodes, but does not guarantee locality for subsequent stages. FlexMap [18] changes the input block sizes to create elastic map tasks for MapReduce. Corral [12] tries to coordinate input data placement with job task placement, so that the most jobs can be run entirely within 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, Corral utilizes extra network to offline aggregates input data for each job. In contrast, Dawn can dynamically aggregate dependent input data for subsequent jobs and co-locate their tasks with data without offline operation.

Data replication techniques: Scarlett [19] is a proactive replication design that periodically replicates files based on predicted popularity. The main objective of Scarlett is to alleviate hotspots caused by massive concurrent accesses to the popular files. DARE [20] dynamically detects the popularity changes at smaller time scales and adopts caching approach to achieve better locality. DALM [21] is a dependency-aware replication strategy which preserves the data dependency in replica placement to minimize cross-server traffic. Unlike Dawn which are online scheduler, the offline extra replications in these techniques, however, requires more disc to store data and network to transfer data.

VI. Conclusion and Discussion

In this paper, we proposed and developed Dawn, a 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.

VII. Acknowledgment

This research was supported in part by U.S. NSF awards CNS-1217979 and CNS-1422119.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 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," *Proc. of ACM SIGMOD*, 2015.
[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *Proc. of USENIX HotCloud*, 2010.
[4] "Hadoop capacity scheduler," http://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.
[5] "Hadoop fair scheduler," http://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.
[6] 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," *Proc. of ACM EuroSys*, 2010.
[7] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," *Proc. of USENIX ATC*, 2014.
[8] "Mahout," https://mahout.apache.org.
[9] 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," *Proc. of VLDB*, 2009.
[10] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, "Scale-out networking in the data center," *IEEE Micro*, 2010.
[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," *Proc. of ACM SIGCOMM*, 2009.
[12] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: plan when you can," *Proc. of ACM SIGCOMM*, 2015.
[13] "Tpc-h benchmark," http://www.tpc.org/tpch/.
[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," *Proc. of ACM SIGOPS*, 2009.
[15] Y. Guo, J. Rao, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," *Proc. of USENIX ICAC*, 2013.
[16] N. Zacheilas and V. Kalogeraki, "Chess: Cost-effective scheduling across multiple heterogeneous mapreduce clusters," *Proc. of IEEE ICAC*, 2016.
[17] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," *Proc. of VLDB*, 2011.
[18] C. Wei, R. Jia, and Z. Xiaobo, "Addressing performance heterogeneity in mapreduce clusters with elastic tasks." in *Proc. of IEEE IPDPS*, 2017.
[19] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," *Proc. of ACM EuroSys*, 2011.
[20] C. L. Abad, Y. Lu, and R. H. Campbell, "Dare: Adaptive data replication for efficient cluster scheduling," *Proc. of IEEE Cluster*, 2011.
[21] X. Fan, X. Ma, J. Liu, and D. Li, "Dependency-aware data locality for mapreduce," *Proc. of IEEE Cluster*, 2014.