Data Prefetching and Eviction Mechanisms of In-Memory Storage Systems Based on Scheduling for Big Data Processing

Chien-Hung Chen, Ting-Yuan Hsia, Yennun Huang, *Fellow, IEEE*, and Sy-Yen Kuo, *Fellow, IEEE*

Abstract—In-memory techniques keep data into faster and more expensive storage media for improving performance of big data processing. However, existing mechanisms do not consider how to expedite the data processing applications that access the input datasets only once. Another problem is how to reclaim memory without affecting other running applications. In this paper, we provide scheduling-aware data prefetching and eviction mechanisms based on Spark, Alluxio, and Hadoop. The mechanisms prefetch data and release memory resources based on the scheduling information. A mathematical method is proposed for maximizing the reduction of data access time. To make the mechanisms applicable in large-scale environments, we propose a heuristic algorithm to reduce the computational time. Furthermore, an enhanced version of the heuristic algorithm is also proposed to increase the amount of prefetched data. Finally, we perform real-testbed and simulation experiments to show the effectiveness of the proposed mechanisms.

Index Terms—Big data processing, in-memory systems, scheduling information, data prefetching, data eviction

1 Introduction

In the era of Big Data, the amount of data on the world will double in size every two years and reach at least 4.4 ZB by 2020 [1]. Cloud data management is one of the key challenges to improve performance of processing large amount of data. Nowadays, multi-tiered storage systems are used in cloud data centers where different storage media devices have a variety of capacity and performance capabilities. A cloud data management system has to decide which data should be placed on low-latency devices to meet performance requirements. If there are datasets that will not be accessed again, the management system should also evict them to high-latency devices for releasing more valuable resources to meet performance requirements of other datasets.

In-memory techniques keep datasets in random access memory to speed up processing of large amounts of datasets. The techniques are widely used in data processing and data storage systems. The in-memory data processing systems strive to analyze a large amount of data in a small amount of time. By keeping the frequently used data in memory, the execution time of jobs can be significantly improved, especially for the iterative jobs that iteratively reading the same datasets. However, non-iterative jobs are difficult to gain benefits from in-memory techniques, and the memory resources may be wasted on storing the datasets that will not be accessed again. Besides improving read throughput, write workloads are major bottleneck for data-intensive jobs. An in-memory data storage system is able to address such bottleneck. It caches output data in memory and achieves fault-tolerance by leveraging *lineage* [2].

In a large cloud data center, many data-intensive jobs may be running simultaneously. However, each computing node in the cloud has limited memory space to cache input and output data for multiple jobs. When a job j_1 is writing its output datasets, the input datasets of job j_2 may be evicted from the memory due to contention of memory resources. In such a case, if the job j_2 cannot read its input datasets from memory, its execution time will be extended. To address this problem, there is a need for management of memory usage for multiple jobs. Most in-memory data processing systems reserve memory space for storing input datasets, intermediate results, and application programs. An in-memory data processing system usually caches data in memory when the data is read or written. Nevertheless, it cannot guarantee that the cached data will be reused. Even if the data blocks will not be accessed again, the data blocks may still be kept in memory until the eviction policy throws them out. If there is not enough space to cache other data blocks, the system will evict some data blocks from memory using Least Recently Used (LRU) policy [3]. In-memory storage systems adopt the same policy to deal with the datasets. Due to that the policy is not aware of which data blocks will be accessed, it may evict the data blocks

- *C.-H. Chen, T.-Y. Hsia, and S.-Y. Kuo are with the Department of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan. E-mail: {d01921025, r03921054, sykuo}@ntu.edu.tw.*
- *Y. Huang is with the Research Center for Information Technology Innovation, Academia Sinica, Taipei 11529, Taiwan. E-mail: yennunhuang@citi.sinica.edu.tw.*

Manuscript received 14 May 2018; revised 28 Dec. 2018; accepted 5 Jan. 2019. Date of publication 14 Jan. 2019; date of current version 8 July 2019. (Corresponding author: Chien-Hung Chen). Recommended for acceptance by S. Chen. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2019.2892957

that will be accessed in near future. In the meantime, it can also keep other unnecessary data blocks in memory.

To deal with these problems, in this paper, we provide the Scheduling-Aware Data Prefetching (SADP) for data processing services in a cloud data center. The proposed SADP includes data prefetching and data eviction mechanisms. Both of the mechanisms are aware of the job scheduling in the cloud. The prefetching mechanism is able to load input data into memory before its corresponding tasks are executed. It can avoid unnecessary prefetching workloads. For the eviction mechanism, it aims to release memory space without affecting the access of running applications. When a block of datasets has already been processed by a task, the data block will be released from memory space if it will not be accessed again. The proposed mechanisms are implemented by modifying Spark [4], Alluxio [5], and Hadoop [6]. Apache Spark and Hadoop MapReduce have become popular data processing systems. Especially for Spark, it optimizes the execution of data-intensive applications with features of interactive data exploration and multi-pass analytics. Alluxio is an in-memory data storage system with ability to manage data in tiered storage. Multi-tiered storage is conducive to balance capacity and performance requirements of data storage. It provides more flexible data management in a cloud data center. Currently, the existing data managements still do not take job scheduling and prefetching deadline into account. The proposed mechanisms can use scheduling information to prefetch data in time and then evict data without affecting the running applications.

This paper is extended version of our previous work [7]. Comparing with the previous work, the paper makes the following additional contributions:

- Data prefetching mechanisms have been proposed to accelerate the progress of data processing in previous work. However, some data blocks cannot be prefetched into memory in time. It causes unnecessary workload and decrease the number of successfully prefetched data blocks. In the paper, we redesign the data prefetching mechanisms with the consideration of deadline constraint.
- The optimal solution of the data prefetching problem is obtained by integer linear programming. To avoid large computational time for obtaining the optimal solution, a heuristic algorithm is presented in Section 4.2. Additionally, an enhanced version of the heuristic algorithm is also given in Section 4.3, which is not proposed in previous work.
- Comparing with the conference version, the proposed mechanisms can prefetch data to non-local computing nodes. So that the number of prefetched data blocks can be increased.
- In addition to redesigning the data prefetching mechanisms proposed in the conference version, the journal version also extends the evaluations. More metrics in real-testbed experiments are evaluated. More existing mechanisms are compared with our proposed mechanisms. Simulations are also performed to evaluate the proposed mechanisms in large-scale cloud data centers.

Overall, this paper makes the following contributions:

- This paper proposes scheduling-aware data prefetching mechanisms with considering of deadline constraint. The proposed mechanisms are able to avoid unnecessary prefetching workloads caused by the data blocks which cannot be prefetched in time.
- A data eviction mechanism considering multiple jobs running on the cloud is presented, where each computing node in the cloud has limited memory resources.
- The data prefetching problem is optimally solved by Integer Linear Programming (ILP). Moreover, two efficient heuristic algorithms are also proposed to solve the data prefetching problem in a large-scale cloud data center.
- The mechanisms proposed in this paper are implemented on a real-testbed. The evaluations show that the proposed mechanisms can achieve about 3.77 times faster than default mechanism in heterogeneous environment.

The rest of this paper is organized as follows. The related work is given in Section 2. Section 3 gives system model. Section 4 presents our mechanisms. Section 5 shows the evaluations of the proposed mechanisms. Finally, section 6 concludes this paper.

2 Related Work

To improve the performance of data processing, in-memory techniques have been extensively used in data processing systems. Spark [8] is a popular data processing framework for data analysis. It presents a data abstraction, called resilient distributed dataset (RDD), which allows application jobs to cache intermediate results in memory with a fault-tolerance mechanism. Mammoth [9] is an implementation of in-memory techniques based on MapReduce framework. It aims to allocate and reclaim memory resources among computing nodes for enhancing overall performance of application jobs. In the system, each computing node is deployed a special engine to globally manage the memory resources among a cluster. GraphLab [10] is an efficient shared-memory implementation of parallel computing framework for machine learning. A graph-based data model is exploited for representing data and computational dependencies. However, it assumes all data can be stored in memory without the problem of resource contention. SINGA [11] is an open-source platform for distributed deep learning. It is able to support different neural net partitioning schemes and training frameworks. Shared memory resources among the systems are leveraged to store intermediate results for reducing data accessing costs. More data processing systems are proposed for real-time purposes, such as Apache Storm [12] and Yahoo! S4 [13]. Among existing in-memory data processing systems, these systems only take intermediate data of an application job into account, therefore these in-memory data processing systems can only benefit from the jobs iteratively accessing the same datasets. On the other hand, memory resource contention is not considered as well. If there are multiple jobs running in the system, some portions of the intermediate data will be evited from memory, such that

these application jobs have to read them from hard disk drives.

In general, there are two types of in-memory storage systems: file systems and database systems, respectively storing unstructured and structured data. Alluxio, formerly called Tachyon [2], is a distributed file system. It can work as a cache system to enhance performance of data accessing for other file systems and databases. Alternatively, it can work as a standalone in-memory file system managing the storage resources. Mercury [14] is a distributed in-memory database for storing structured data. A dedicated hash table is designed for small data sizes of key-value pairs to improve throughput of data analytics jobs. MICA [15] is a key-value in-memory storage system focusing on both read- and write-intensive jobs. It aims to use fewer high-performance computing nodes to reduce latency of data access. New data structure and memory management are designed for optimizing data store and cache by using the properties of the jobs. Pilaf [16] is a distributed in-memory key-value storage system with high-performance networks. It allows application jobs directly access the data stored in memory from remote computing nodes. A self-verifying data structure is also provided to address contention of read-write operations. Including above systems, most existing in-memory storage systems focus on design of memory management based on specific properties of data structures and iterative jobs. However, the job scheduling is not taken into account. In this work, we focus on the Hadoop Distributed File System and provide data prefetching mechanisms based on the schedulers provided by Hadoop. Alluxio is used to manage memory resources among the distributed system.

Job scheduling plays an important role in allocating CPU and memory resources for executing data processing jobs among a large-scale cloud. There are three popular scheduling modes to deal with multiple jobs: Standalone, Mesos, and YARN. In standalone mode, all jobs in a cluster are run in FIFO (first in, first out) order. The tasks of each job can be allocated to all computing nodes for reaching maximum usage of CPU and memory resources [3]. In Mesos mode, the system allocates resources in accordance with user-defined policy, such as fair sharing and strict priority [17]. It can also share system resources at different granularities based on latency requirements of Spark jobs. In YARN mode, one of simple FIFO, Capacity, and Fair Share schedulers can be selected depending on the user needs [18]. In addition, data locality can have significant impacts on job scheduling. A task of running jobs prefers to be allocated to where its input data stored. Therefore, the job schedulers are designed around the general principle of data locality.

Data prefetching has been widely used in data processing systems [19], [20], [21]. The authors of [19] implemented PACMan to speed up the execution of MapReduce jobs by caching the input data in memory. In the proposed caching mechanisms, the number of parallel tasks was mainly concerned. If an input data block of a running task is not cached, the job execution time can be extended. LIFE and LFU-F were two eviction mechanisms proposed for minimizing the average job execution time and maximizing system efficiency, respectively. Data prefetching was used to enhance the access of singly-accessed data blocks. The authors of [21] designed HPSO to provide data prefetching service for improving data locality of MapReduce. The proposed HPSO predicts the remaining execution time of map tasks and further estimates which computing unit will become idle. Each computing node is able to automatically complete data prefetching before a map task is launched. The authors of [20] addressed the non-local straggler problem in MapReduce by leveraging data prefetching. A speculative scheduler was proposed to predict the appearance time and the location of future non-local straggler tasks. The FlexFetch was built to generate prefetching requests based on the proposed speculative scheduler and allocate network resources for data prefetching. To allocate appropriate network resources, the authors implemented an OpenFlow-based network controller to guarantee the end-to-end network transition rate. Among the existing mechanisms, the prefetching deadline and resource contentions are not taken into account.

3 Preliminaries

This section presents the system model of in-memory systems used in this paper. The definitions regarding to the data prefetching problem are also provided.

3.1 System Description

Spark is a big data processing framework designed to be fast and general. The resilient distributed dataset is the core concept in Spark. It represents a collection of data partitions distributed across many computing nodes. A Spark job can be divided into two or more stages, where each stage consists of a set of tasks. The stages are processed in order defined by a directed acyclic graph (DAG). A central process, called the driver, is responsible for coordinating with a number of executors to run the tasks of the given job. The number of tasks in a stage is the same as that of data partitions generated in the previous stage. Each task within a job accesses its corresponding data partition, then it performs either *transformation* or *action* operations. If a task performs transformation, its output is constructed as new RDDs. If it performs action, it will return the computing results to the driver process or store the output of the job. When a Spark job is submitted, the driver process firstly asks the cloud manager for resources to launch executors. Tasks of the given job are sent to the executors to perform transformation or action operations. In the first stage of a job, the tasks usually perform transformation operation to load each data block of the input datasets from an external storage system and create RDDs. In the last stage, the tasks perform action operation to save output results to the external storage system. Our proposed mechanisms aim to prefetch and evict the data blocks of the datasets stored in the external storage system, instead of the data partitions of RDDs used in data processing layer. Therefore, the proposed mechanisms can be applied to MapReduce framework [22] as well, where the input data blocks of map tasks and the output data blocks of reduce tasks are concerned to be prefetched or evicted to/from the memory of the external storage. The intermediate output data of the map tasks is not taken into account.

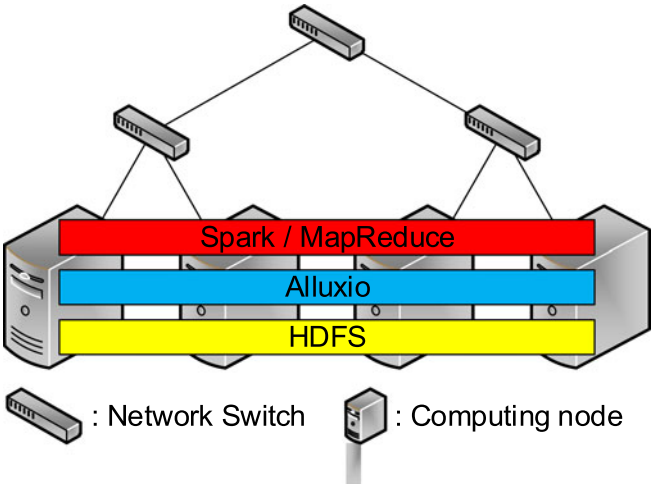


Fig. 1. The architecture of our system.

Alluxio is the external storage system used in this paper. It supports tiered storage to manage data blocks stored in memory (MEM) and hard disk drive (HDD). When a job is writing new data blocks to the external storage, the data blocks will be firstly cached to the memory. If there is no enough space to accommodate new data blocks, the system will evict the least recently used data blocks cached memory by default. For the under storages, Alluxio can integrate with various under storages, such as Apache HDFS [23], OpenStack Swift [24], Amazon S3 [25], etc. The HDFS, a popular distributed file system, is used as the under storage of Alluxio. It can be deployed to the same computing nodes with Spark and Alluxio, so that the system can take advantage of data locality to avoid network transmission delay. The system architecture we used is illustrated in Fig. 1. As shown in Fig. 1, Spark, Alluxio and Hadoop are installed among the computing nodes, where Spark and MapReduce are responsible for executing data processing jobs. Alluxio is the external storage system of the Spark. It reserves a part of memory space of each computing node for caching data blocks from its under storage system. HDFS is Alluxio's under storage system storing input and output datasets of Spark jobs in hard disk drives.

The proposed data prefetching and eviction mechanisms can be applied to another computing framework if its scheduler is able to provide explicit task assignments based on current system status. Due to the prediction of where and when a task will be launched are the important metrics, the proposed mechanisms are suggested to be used with the schedulers which are able to assign a task to the same computing node when the statuses of computing resources and data locations are the same. Otherwise, the schedulers can make inaccurate prediction of task assignments and cause lower performance.

3.2 System Model and Definitions

This paper investigates scheduling-aware data prefetching and eviction mechanisms in heterogeneous cloud environments. Before elaborating our proposed mechanisms, we first give the following definitions.

Given a large-scale cloud with a set of computing nodes N. Each computing node $n \in N$ has reserved a certain size of memory space for caching data blocks. The available memory space is firstly used for data prefetching. If the memory is full, the system will evict suitable number of data blocks from memory. A job in the cloud consists of a number of tasks, where each task t_i of the job is associated with an input split size t_i^s. If the input split size of a task is greater than the size of data blocks, the task has to receive two or more data blocks to find the corresponding data records as input. For example, a task with 128MB of input split size has to access two 64 MB data blocks. According to the running state, the tasks can be classified into three types: completed tasks, running tasks, and pending tasks. The information of both job scheduling and datasets can be obtained from the master node of the system. Each computing node in the cloud periodically reports the states of the running tasks and the stored data blocks to the master node. Therefore, the master can allocate pending tasks to the computing nodes based on a specific scheduling policy. Furthermore, it can predict which pending task will be launched after a running task is completed on a node. For the datasets, there are two sets of data blocks D_h and D_m. Each data block d_i in D_h and D_m is associated with a data block size d_i^s. To simplify the prefetching problem, we set 64 MB as the basic data block size and assume that all data block sizes are multiples of 64 MB. If a data block size is 128 MB, it can be normalized as 2 basic data block size. In the following sections, we use the basic data block size to count the number of available storage space. The D_h denotes a set of data blocks stored in the disks. D_p is a subset of D_h, where each data block $d_i \in D_p$ is not cached in memory and its corresponding task is predicted to be launched on a specific computing node. The D_m denotes a set of data blocks cached in memory. D_e is a subset of D_m. For each data block $d_i \in D_e$, its corresponding task is completed. Among the computing nodes, the available memory space is represented by a set of memory blocks M, where each memory block $m_k \in M$ can cache a data block. All data blocks are assumed to be accessed by write-once, read-many-times pattern [26].

Estimation of data access time is complicated especially in heterogeneous clouds. The queuing model $M/G/m/m+r$ is applied to evaluate the data access performance in could computing systems [27]. In our system model, all computing nodes can report the data access rates and network transmission rates for dynamical estimation. For each computing node $n \in N$, the master node maintains real-time disk access rate r_n^h and memory access rate r_n^m, respectively. With the real-time information, if a data block d_i with size d_i^s is stored on node $n \in N$, the local disk access time $T_{local}^{disk}(n, d_i)$ and memory access time $T_{local}^{mem}(n, d_i)$ can be estimated by $d_i^s \div r_n^h$ and $d_i^s \div r_n^m$, respectively. The real-time network transmission rates are maintained in an $|N|$ by $|N|$ matrix R. If a data block d_i with size d_i^s is sent from n_j to n_k, its network transmission time can be estimated by $d_i^s \div R_{jk}$.

Definition 1. *If a task is running on the node n_r and its corresponding data block $d_i \in D_h$ is stored in the disk of node n_h, the disk access time of the data block d_i is estimated by*

$$T_{access}^{disk}(n_r, n_h, d_i) = T_{local}^{disk}(n_h, d_i) + T_{trans}(n_r, n_h, d_i), \quad (1)$$

where $T_{local}^{disk}(n_h, d_i)$ is the access time taken for the local disk of node n_h storing the data block d_i, $T_{trans}(n_r, n_h)$ is the network

TABLE 1
Summary of Notations

Notation	Description
N	A set of computing nodes in the cloud.
D_h	A set of data blocks stored in the hard disk drives.
D_m	A set of data blocks cached in the memory.
D_p	A set of data blocks stored in the hard disk drives, where their corresponding tasks are the pending tasks predicted to be launched.
D_e	A the set of data blocks cached in the memory, where their corresponding tasks are completed.
M	A set of memory blocks, where each memory block $m_k \in M$ can cache a data block.
a_m	The number of available memory space in the computing node n_m.
t_r	A running task associated with a remaining execution time $T_{remain}(t_r)$. After the task t_r is completed, a task t_i is will be launched on the same node. Then, its corresponding data block d_i will be prefetched to a node n_m.
α	A coefficient that is set to $1 + \max_{\forall d_j \in D_m \wedge \forall n_m \in N} T^{disk}_{access}(n_m, d_j)$.
x_{im}	A binary variable indicating whether a data blocks d_i is prefetched to memory of node n_m.
y_{jm}	A binary variable indicating whether a data blocks d_j is evicted from memory of node n_m.

transmission time taken for node n_r to retrieve data block d_i from a remote computing node n_h. Note that, the data block d_i can be cached to the memory of another node different from n_r and n_h.

Definition 2. *If a data block $d_j \in D_m$ is cached in memory of node n_m, the memory access time of the data block d_j is estimated by*

$$T^{mem}_{access}(n_r, n_m, d_j) = T^{mem}_{local}(n_m, d_j) + T_{trans}(n_r, n_m, d_j), \quad (2)$$

where $T^{mem}_{local}(n_m, d_j)$ is the access time taken for local memory of node n_m cached the data block d_j. If the data block $d_j \in D_m$ is determined to be evicted and written to a data block $d_i \in D_h$ stored node n_h, its eviction cost can be estimated by Eq. (1).

Definition 3. *Given a task running on node n_r, a data block $d_i \in D_h$ stored in the disk of node n_h, a data block $d_j \in D_m$ representing the data block d_i cached in memory of node n_m. The benefit time earned by caching the data block d_i to memory of node n_m can be estimated as*

$$\begin{aligned} &T_{benefit}(n_r, n_h, d_i, n_m, d_j) \\ &= T^{disk}_{access}(n_r, n_h, d_i) - T^{mem}_{access}(n_r, n_m, d_j). \end{aligned} \quad (3)$$

The benefit time $T_{benefit}(n_m, d_i)$ earned by caching the data block d_i is required to be removed when the data block d_j is evicted for data prefetching.

Note that, if a data block d_i can be prefetched to the memory of node n_m, it has to satisfy the prefetching conditions: 1) The prefetching of data block d_i can be completed before the corresponding pending task is launched. 2) The benefit time is greater than 0. 3) The size of the data block has to smaller or equal to the available memory space. Based on the above definitions and conditions, the main objective of the data prefetching problem is to find an optimal dataset D_{opt} from D_p and D_e, which can maximize the total benefit time and minimize the total data eviction cost.

Estimation of task remaining time is critical to predict when a pending task will be launched. We follow our previous work [28] to estimate the data processing time of a new task by using the computing resources, the number of instructions, input data size, etc. Moreover, the data process rate of a task t on a node n, denoted by r^t_n, is also sent to the master node for monitoring the performance changes. So the data processing time of a task t running on node n, denoted by $T_process(n, t)$, can be dynamically estimated by $d^s_i \div r^t_n$, where d^s_i is the size of the input data block d_i.

The estimation of task remaining time is estimated by

$$T^{task}_{remain}(t_r) = T^{access}_{remain}(t_r) + T^{process}_{remain}(t_r), \quad (4)$$

where t_r is a task running on node n_r, $T^{access}_{remain}(t_r)$ is the remaining data access time, and $T^{process}_{remain}(t_r)$ is remaining data processing time. The remaining data access time can be estimated by Eq. (1) and Eq. (2), and replacing the input data size d^s_i by the remaining input data size d^{rs}_i. Finally, the remaining data access time can be estimated by $d^{sr}_i \div r^{tr}_{nr}$.

4 Scheduling-Aware Data Prefetching

4.1 Optimal Solution

The data prefetching problem can be optimally solved by Integer Linear Programming. ILP is a mathematical technique to the found optimal solution, where its canonical form includes a linear objective function, a number of linear constraints, and an integer solution set. In this section, the canonical form of ILP corresponding to the prefetching problem is expressed as Eq. (5) to Eq. (12). The used notations can be found in Table 1.

$$\begin{aligned} Maximize \sum_{\forall d_i \in D_p} \sum_{\forall n_m \in N} x_{im} \times \{T_{benefit}(n_m, d_i) + \alpha\} \\ - \sum_{\forall d_j \in D_e} \sum_{\forall n_m \in N} y_{jm} \times T^{disk}_{access}(n_m, d_j), \end{aligned} \quad (5)$$

$$subject\ to\ \forall d_i \in D_p, \sum_{\forall n_m \in N} x_{im} \times d^s_i \leq a_m, \quad (6)$$

$$\forall d_i \in D_p, \sum_{\forall n_m \in N} x_{im} \leq 1, \quad (7)$$

$$\forall d_j \in D_e, \sum_{\forall n_m \in N} y_{jm} \leq 1, \quad (8)$$

$$\forall d_i \in D_p, \sum_{\forall n_m \in N} x_{im} \times T_{benefit}(n_m, d_i) > 0, \quad (9)$$

$$\forall d_i \in D_p, \sum_{\forall n_m \in N} x_{im} \times T^{disk}_{access}(n_m, d_i) < T_{remain}(t_r), \quad (10)$$

$$\forall n_m \in N, \sum_{\forall d_i \in D_p} x_{im} \times d^s_i - \sum_{\forall d_j \in D_e} y_{jm} \times d^s_j = a_m, \quad (11)$$

$$\forall d_i \in D_p \wedge \forall d_j \in D_e \wedge \forall n_m \in N, x_{im}, y_{jm} \in \{0, 1\}. \quad (12)$$

For the Eq. (5), it is the objective function of the canonical form. There are two terms in the objective function. The first term is the benefit time of the data blocks prefetching to memory, and the second term is the benefit time that have to be removed due to the eviction of data blocks. The symbol D_p denotes a set of data blocks stored in hard disk drives, where their corresponding tasks are predicted to be launched. The D_e denotes a set of data blocks cached in memory, where their corresponding tasks are completed. N is a set of nodes in the cloud. In the Eq. (5), the data prefetching and eviction can be obtained based on the binary variables x_{im} and y_{jm}, respectively. If x_{im} is 1, the corresponding data block d_i is selected to be cached in memory of node n_m, where the data block d_i can earn the benefit time $T_{benefit}(n_m, d_i)$. If y_{jk} is also 1, the corresponding data block d_j is evicted from memory of node n_m. The evicted data block may have to store back to disk, where the cost of eviction is estimated Eq. (1). The coefficient is set to

$$\alpha = 1 + \max_{\forall d_j \in D_m \wedge \forall n_m \in N} T^{disk}_{access}(n_m, d_j). \tag{13}$$

By setting the coefficient α, each weight of x_{im} has larger weight than each weight of y_{jm}. It guarantees that the data prefetching works when the disk access cost of storing evicted data blocks is greater than the earned benefit time. In accordance with the constraints of Eq. (7) to Eq. (12), the canonical form will maximize the benefit time. If the memory is full and there is a data block d_i which weight is greater than the data block d_j's weight, the objective function will evict d_j and cache d_i by setting x_{im} and y_{jm} as 1. If there is a data block that is evicted from memory, the canonical form will cache another data block due to Eq. (11). The reason will be explained later. From the above description, we can know that the canonical form will firstly prefetch high-weight data blocks. Then, it replaces low-weight data blocks by high-weight data blocks. Eq. (6) expresses that the data block's size d^s_i had to small or equal to the available memory space a_m on node n_m. Eqs. (7) and (8) respectively express that a data block stored in hard disk drives or memory can only be cached or evicted once. Eq. (9) denotes that a data block can be prefetched if and only if the access time of the data block can be improved. Eq. (10) is the time constraint indicating that the selected data block d_i has to be prefetched in time. $T_{remain}(t_r)$ denotes the running task t_r's remaining execution time. In Eq. (11), the number of prefetched data blocks in a node should equal to the number of evicted data blocks plus the free memory space of the node. Due to this constraint, the high-weight data blocks will fill up the free memory space of the system, and the low-weight data blocks in memory can be replaced by high-weight data blocks. Finally, the Eq. (12) is given for setting the solution domain. The variables x_{im} and y_{jm} can only be 0 or 1, respectively. Note that, there are many dynamical parameters can also be changed in different runs depending on dynamical workloads, such as α and a_m. In our system, all dynamical parameters are reported from the computing nodes and collected on the master node, so the master node is able to use these parameters locally.

In the above canonical form, there are $|D_p| \times |N|$ and $|D_e| \times |N|$ binary variables in x_{im} and y_{jm}. In a cloud, the number of nodes $|N|$ can be up to 25,000 [29]. Solving ILP is well-known to be NP-complete [30]. If $|D_p|$, $|D_e|$ and $|N|$ are large, the above canonical form of ILP will take much computational time to obtain the optimal solution of the prefetching problem. To reduce the computational time, we propose two heuristic algorithms for the data prefetching problem.

Input: A set of computing nodes N, and a set of running jobs J.
Output: Data prefetching P and data eviction E.

/* Initial Phase */
for each job $j \in J$ **do**
  $D_p \leftarrow$ Find the corresponding data blocks of the pending tasks.
  $D_e \leftarrow$ Find the corresponding data blocks of the completed tasks.
  for each job $d_i \in D_p \cap D_e$ **do**
    Remove d_i from D_e.
    if d_i has been cached in memory **then**
      Remove d_i from D_p.
    end if
  end for
end for
/* Prefetching Phase */
Sort the running tasks T_r in ascending order of the remaining time.
for each running task $t_r \in T_r$ **do**
  $t_i \leftarrow$ Find the task that will be launched after completion of t_r.
  if the corresponding data block d_i of t_i is in D_p **then**
    $n_m \leftarrow$ Find a node with the maximum $T_{benefit}(n_m, d_i)$ from N, which satisfies the prefetching conditions.
    $P \leftarrow P \cup (n_m, d_i)$
  end if
end for
/* Eviction Phase */
for each $(n_m, d_i) \in P$ **do**
  $m_k \leftarrow$ Find an available memory space on n_m for d_i.
  if m_k is NULL **then**
    $d_j \leftarrow$ Find an un-modified data block d_j cached in n_m from D_e.
    if d_j is NULL **then**
      $d_j \leftarrow$ Find a modified data block d_j cached in n_m from D_e, where the d_j has the minimized $T^{disk}_{access}(n_m, d_j)$.
    end if
    $E \leftarrow E \cup d_j$
  end if
end for

Fig. 2. The pseudo-code of the heuristic algorithm.

4.2 Heuristic Algorithm

In this section, we present a heuristic algorithm, which also assumes that the scheduling information is available from computing layer. The algorithm consists of three phases: initial phase, prefetching phase and eviction phase. The basic idea of the heuristic algorithm is given in Fig. 2. When a job is submitted, the job is associated with an input dataset, and then it will be put in a ready queue. The input dataset is divided into multiple data blocks. Each task of the job can read or write its corresponding data block from/to the external storage. Before submitting the job, multiple data processing jobs may have been running simultaneously in the cloud. In such a case, a data processing job may be repeated several times. The characteristic of each task of a job can be known in advance. The characteristic provides the information such as the task progress and the average task execution time in accordance with the input data block. Using the information, the completion time of a task can be predicted. For example, the completion time of a task can be roughly predicted by the average task execution time minus the elapsed time of the task. The input data blocks stored in hard disk drives are collected in a set of data blocks D_h. The

D_m denotes the set of data blocks that have been cached in memory. The states and locations of the data blocks can be obtained from the external storage system. In the heuristic algorithm, the initial phase is used to find which data blocks are prefetchable and evictable. In prefetching phase, it predicts when and which pending tasks will be launched on specific computing nodes. The corresponding data blocks of the pending tasks are prefetched to the computing nodes which will run the pending tasks. If a task has processed its corresponding data, the data block can be evicted from the memory. The eviction phase releases memory resources if remaining memory space is not enough to prefetch data blocks.

4.2.1 Initial Phase

Before a new job is submitted to the cloud, its input datasets have already been stored in the external storage system. When a new job is submitted to the cloud, both of its input and output datasets have been specified. The new job is then put in a job ready queue for waiting for execution. While there may already be a number of jobs running simultaneously in the cloud, it is necessary to find the following data blocks before running the pending tasks of the jobs.

- *Prefetchable data blocks (The data blocks that can be prefetched into memory).* Considering multiple jobs running on the cloud with limited computing resources, only a portion of tasks can run simultaneously in the cloud at a time. According to job scheduling, each job running on the cloud can only launch a specific number of tasks. Some pending tasks have to wait for execution. If the input data blocks of these pending tasks are not cached in memory, these data blocks are classified as prefetchable data blocks. In other words, the prefetchable data blocks are the input data blocks of the pending tasks, which are not cached in memory.
- *Evictable data blocks (The data blocks that can be evicted from memory).* When multiple jobs running in the cloud, several data blocks may have been cached in memory. If a job reads its input dataset from the external storage system, each task of the job only reads one data block of the dataset. If the input data block of a completed task is not a prefetchable data block, it will not be accessed again. Evicting such data block will not affect accessing of other running tasks. In this phase, the proposed mechanism decides which data blocks can be evicted without affecting other running tasks.

An example is given in Fig. 3, where there are two jobs running in a cloud with the Fair scheduling. The job j_1 performs reading, and the job j_2 performs writing, respectively. Based on the job scheduling, the jobs j_1 and j_2 respectively hold two executors to perform their tasks. The $t_{1,4}$ and $t_{1,5}$ are two tasks of job j_1 respectively reading data blocks $d_{1,4}$ and $d_{1,5}$ from the external storage system. The tasks $t_{2,5}$ and $t_{2,6}$ of job j_2 read the data blocks $d_{2,5}$ and $d_{2,6}$, respectively. There are 11 data blocks $d_{1,1}$, $d_{1,2}$, $d_{1,3}$, $d_{1,4}$, $d_{1,5}$, $d_{2,1}$, $d_{2,2}$, $d_{2,3}$, $d_{2,4}$, $d_{2,5}$, and $d_{2,6}$ that have already cached in memory of the external storage system, where $d_{2,1}$, $d_{2,2}$, $d_{2,3}$, and $d_{2,4}$ are the new data blocks generated by the job j_2. In the example, we

Running Jobs

Job	Completed tasks	Running Tasks	Pending Tasks
j_1	$t_{1,1}, t_{1,2}, t_{1,3}$	$t_{1,4}, t_{1,5}$	$t_{1,6}, t_{1,7}, t_{1,8}, t_{1,9}$
j_2	$t_{2,1}, t_{2,2}, t_{2,3}, t_{2,4}$	$t_{2,5}, t_{2,6}$	$t_{2,7}, t_{2,8}, t_{2,9}, t_{2,10}$

Remaining execution time of the running tasks

Task	$t_{1,4}$	$t_{1,5}$	$t_{2,5}$	$t_{2,6}$
Time	25	27	35	38

Prefetching time of each data block (Disk)

	n_1	n_2	n_3	n_4
$d_{1,6}$	20	30	40	40
$d_{1,7}$	30	30	10	20
$d_{2,7}$	40	40	30	20
$d_{2,8}$	20	30	40	40

Benefit time of prefetching each data block

	n_1	n_2	n_3	n_4
$d_{1,6}$	15	3	-5	-7
$d_{1,7}$	15	23	5	3
$d_{2,7}$	5	3	25	13
$d_{2,8}$	15	13	25	33

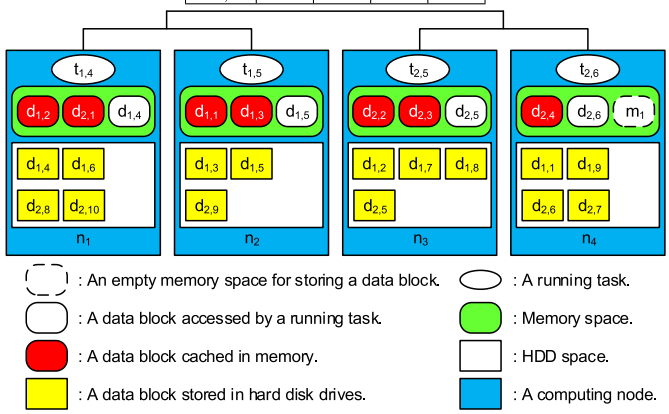


Fig. 3. An example to demonstrate the heuristic algorithm.

assume that all data blocks have the same size and the external storage system can at most cache 12 data blocks in memory. The initial phase firstly determines which data blocks are evictable. Based on the task execution log, the tasks $t_{1,1}$, $t_{1,2}$, $t_{1,3}$, $t_{2,1}$, $t_{2,2}$, $t_{2,3}$, and $t_{2,4}$ are finished. Therefore, their corresponding input data blocks $d_{1,1}$, $d_{1,2}$, $d_{1,3}$, $d_{2,1}$, $d_{2,2}$, $d_{2,3}$, and $d_{2,4}$ are decided to be evictable data blocks. After finding the evictable data blocks, the eviction phase then determines which data blocks can be prefetched into memory. In this phase, all of the pending tasks $t_{1,6}$, $t_{1,7}$, $t_{1,8}$, $t_{1,9}$, $t_{2,7}$, $t_{2,8}$, $t_{2,9}$, and $t_{2,10}$ are the tasks that will be launched in near future. Their corresponding input data blocks $d_{1,6}$, $d_{1,7}$, $d_{1,8}$, $d_{1,9}$, $d_{2,8}$, $d_{2,7}$, $d_{2,9}$, and $d_{2,10}$ are decided to be the prefetchable data blocks. Totally, eight data blocks are decided to be prefetchable data blocks. The initial phase just decides which data blocks are evictable and prefetchable. The executions of prefetching and eviction are actually performed in the following prefetching phase and eviction phase.

4.2.2 Prefetching Phase

The proposed mechanism turns into the prefetching phase. When the free memory resources are distributed across the cloud data center, a pending task may be launched on a computing node different from the location of its corresponding data block. The job scheduling allocates pending tasks to computing nodes with consideration of data locality. In prefetching phase, it firstly predicts when and which pending task will be allocated to a computing node, then the corresponding data block of the task is selected to be

prefetched to the node. By receiving the state reports from each computing node, the master node maintains the progress and the elapsed execution time of each running task. The remaining execution time of a running task t_r is estimated by Eq. (4).

Based on estimation of the remaining execution time, we can predict when and where an executor will be released for running another pending task. In the prefetching phase, it performs data prefetching on the computing nodes in order of the release sequence of the executors. As shown in the example of Fig. 3, the running tasks are completed by the sequence of $t_{1,4}$, $t_{1,5}$, $t_{2,5}$, $t_{2,6}$. Base on the sequence, the prediction will be progressed in the order of n_1, n_2, n_3, n_4. As mentioned in Section 2, the job scheduling prefers to allocate pending tasks to where their input data blocks located to achieve data locality [18]. Based on data locality and scheduling policy, we can predict which pending tasks will be launched on a specific computing node. In the Fig. 3, the task $t_{1,4}$ is running on computing node n_1. If the running task $t_{1,4}$ is completed, the node n_1 will release an executor to run another pending task. The job scheduling will allocate a task of j_1 to node n_1, because each job holds the same number of executors to run its tasks. To achieve data locality, the job scheduling prefers to find a task whose corresponding data block is closest to the computing node. Therefore, the pending task $t_{1,6}$ will be allocated to the node n_1. The corresponding data block $d_{1,6}$ of task $t_{1,6}$ is selected to be prefetched. After prediction, the data block $d_{1,6}$ is judged whether it can be prefetched. If a data block d_i can be prefetched to node n_m, it satisfies the prefetching conditions.

In the example of Fig. 3, the access time $T^{disk}_{access}(n_1, d_{1,6})$ is less than $T_{remain}(t_{1,4})$, and the $T_{benefit}(n_1, d_{1,6})$ is 15. It means that the data block $d_{1,6}$ can be prefetched to n_1 in time and can gain benefit time. Each computing node in cluster will be checked to find a placement that can obtain the maximized benefit time. In the example, the data block $d_{1,6}$ is decided to be prefetched to node n_1. The prefetching phase will check each computing node to predict and find a prefetching placement with maximized benefit time. The tasks $t_{1,7}$, $t_{2,7}$, and $t_{2,8}$ are predicted to be launched on node n_2, n_3, and n_4, respectively. Finally, only $d_{1,7}$, $d_{2,7}$, and $d_{2,8}$ are prefetched in memory of n_3, n_4, and n_2, respectively. The total benefit time is 46.

The effectiveness of data prefetching is dependent on how many data blocks can be prefetched before the corresponding tasks are launched. In a cloud computing system, many jobs may be run simultaneously. Data prefetching can cause resource contention if there are insufficient storage or network bandwidth. In such a case, the system cannot guarantee that all data blocks of pending tasks can be cached before the corresponding tasks are launched. In the example of Fig. 3, $t_{1,7}$ and $t_{2,8}$ are two tasks predicted to be launched on node n_2 and n_4, respectively. However, if we restrict that the data blocks can only be prefetched to the nodes where the pending tasks will be launched, the data blocks $d_{1,7}$ and $d_{2,8}$ cannot be prefetched to the nodes n_2 and n_4, due to the time limitation. If a data block of pending task is not prefetched, it will extend the execution time of the whole job [3], [19]. With limited available resources, the data blocks should be prefetched as many as possible for the pending tasks. Therefore, our proposed mechanisms allow prefetching data blocks to the nodes different from where the pending tasks will be launched.

4.2.3 Eviction Phase

In the initial phase, the evictable data blocks are found. However, there are two types of evictable data blocks: unmodified data blocks and modified data blocks. The unmodified data blocks are not changed when they are caching in memory. Conversely, the modified data blocks are new or updated data blocks. These data blocks are maintained by the external storage system. By default, the external storage system achieves data fault tolerance using *lineage* technique [2]. Any changes of data blocks are traced by a logical directed acyclic graph. It means that any data blocks written by jobs will be persisted in the memory first. If these data blocks are evicted from memory before writing to hard disk drives, the evicted data blocks will be lost. Therefore, if the modified data blocks are selected to be evicted, these data blocks should be written to the hard disk drives before eviction. In the eviction phase, it decides which evictable data blocks should be evicted from memory of the node if there is no available space. To avoid affecting other tasks accessing their data blocks, the empty memory space is preferred to be used. Instead of modified data blocks, the un-modified data blocks are firstly selected to be evicted from memory. After the prefetching phase, the destinations of the prefetched data blocks have been decided. The computing nodes have to reserve memory resource for the prefetched data blocks. In the example of Fig. 3, $d_{1,6}$, $d_{1,7}$, $d_{2,7}$, and $d_{2,8}$ are prefetched in memory of n_1, n_3, n_4, and n_2, respectively. The computing nodes n_1, n_3 and n_2 respectively have to evict one of the evictable data blocks. For the computing node n_1, it holds two evictable data blocks $d_{1,2}$ and $d_{2,1}$, where $d_{1,2}$ is an un-modified data block, and $d_{2,1}$ is a modified data block. According to the mechanism described above, the $d_{1,2}$ and $d_{1,1}$ are prior to be selected by n_1 and n_2, respectively. A modified data block $d_{2,2}$ is selected by node n_3 because there is no un-modified data block chaced in n_3. After selecting the un-modified and modified data blocks, the selected modified data blocks are updated to the original copies stored in the disks. If a modified data block is a completely new data block, it will be stored in the local disk of computing node caching the modified data block. Finally, the selected data blocks are erased from memory.

4.3 Enhancements to Heuristic Algorithm

As mentioned in Section 4.2, the effectiveness of data prefetching is dependent on the number of prefetched data blocks. However, the heuristic algorithm cannot achieve the maximum number of the prefetched data blocks. Additionally, the total benefit time cannot be maximized as well. In this section, we propose an enhanced version of the heuristic algorithm to maximize the total number of prefetched data blocks and the benefit time.

In the heuristic algorithm, we have predicted which tasks will be launched on specific computing nodes P, and the evictable data blocks have also been found D_e. To maximize the number of prefetched data blocks and the total benefit time, the enhanced version of the heuristic algorithm plans to prefetch the data blocks to more suitable computing

Input: N, D_p, D_e, M.
Output: A network flow graph G.
1: **/* The First Step */**
2: **for** each $d_i \in D_p$ **do**
3: **for** each node n_m in N **do**
4: **if** d_i can be prefetched to the n_m satisfying the prefetching conditions **then**
5: Connect a directed edge from d_i to n_m.
6: $a(d_i, n_m) \leftarrow 1$; $c(d_i, n_m) \leftarrow T_{benefit}(n_m, d_i) + \alpha$
7: **end if**
8: **end for**
9: **end for**
10: **/* The Second Step */**
11: **for** each node $n_m \in N$ **do**
12: **for** each data block $d_j \in D_e$ **do**
13: **if** d_j is cached in n_m **then**
14: Connect a directed edge from n_m to d_j.
15: $a(n_m, d_j) \leftarrow 1$; $c(n_m, d_j) \leftarrow 0$
16: **end if**
17: **end for**
18: **for** each memory block $m_k \in M$ **do**
19: **if** m_k is located on n_m **then**
20: Connect a directed edge from n_m to m_k.
21: $a(n_m, m_k) \leftarrow 1$; $c(n_m, m_k) \leftarrow 0$
22: **end if**
23: **end for**
24: **end for**
25: **/* The Third Step */**
26: **for** each $d_i \in D_p$ **do**
27: Connect a directed edge from s to d_i.
28: $a(s, d_i) \leftarrow 1$; $c(s, d_i) \leftarrow 0$
29: **end for**
30: **for** each $d_j \in D_e$ **do**
31: Connect a directed edge from d_j to t.
32: $a(d_j, t) \leftarrow 1$; $c(d_j, t) \leftarrow$ the negative of $T^{disk}_{access}(n_m, d_j)$.
33: **end for**
34: **for** each memory block $m_k \in M$ **do**
35: Connect a directed edge from m_k to t.
36: $a(m_k, t) \leftarrow 1$; $c(m_k, t) \leftarrow 0$
37: **end for**

Fig. 4. The pseudo-code to establish a network flow graph.

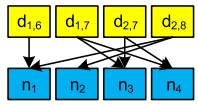
nodes. Based on the new prefetching results, the evicted data blocks are also reselected.

To achieve the goal, we transform the data prefetching problem to the maximum-cost maximum-flow problem. A three-step algorithm is proposed to establish the network graph, as shown in Fig. 4.

Definition 4. *Given a network flow graph $G = (V, E)$, where V is a set of nodes, E is a set of directed edges. Two special nodes represent a source node $s \in V$ and a sink node $t \in V$. The amount of flow sent from the source node s to the sink node t is denoted by f, where the leaving flow of the source node s must equal to the entering flow of sink node t. Other than the source node s and the sink node t, the amount of flow entering a node $v \in V$ is equal to the amount of flow leaving the node v. Each edge $(u, v) \in E$ is associated with a sub flow $f_s(u, v) \geq 0$, a capacity $a(u, v) > 0$, and a cost $c(u, v)$, where the capacity represents the maximum amount of the flow which can be sent via the edge ($a(u, v) \geq f_s(u, v)$). If a sub flow $f_s(u, v)$ is sent by the edge (u, v), the cost of passing the edge is $f_s(u, v) \times c(u, v)$. The maximum-cost maximum-flow problem is to find the maximum amount of flow f passing the network graph with maximum total cost.*

4.3.1 The First Step

In the prefetching phase, it has predicted which pending task will be launched, and the corresponding data blocks are represented by a set of D_p. However, the heuristic algorithm may not obtain optimal total benefit time. Due to the remaining execution time of the running tasks, some data blocks may not be prefetched in time as well. In such a case, the pending tasks have to find other appropriate computing nodes to prefetch their corresponding data blocks. When processing a prefetching request for a data block, it is required to confirm which computing nodes can satisfy the prefetching conditions. In accordance with these constraints, each pending task can find one or more computing nodes to prefetch its corresponding data block. In the example of Fig. 3, the tasks $t_{1,6}$, $t_{1,7}$, $t_{2,7}$, and $t_{2,8}$ are predicted to be launched on computing node n_1, n_2, n_4, and n_3, respectively. Based on the deadline constraint, if a data block d_i can be cached in memory of computing node n_m in time, then a directed edge is connected from d_i to n_m. The relationship between the data blocks and the appropriate computing nodes can be modeled as a subgraph of the network flow graph, as shown in Fig. 5.



Settings of (capacity, cost)

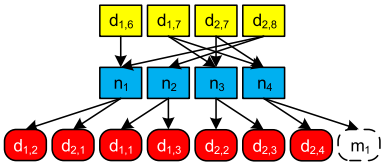
	n_1	n_2	n_3	n_4
$d_{1,6}$	(1, 15)			
$d_{1,7}$			(1, 5)	(1, 3)
$d_{2,7}$			(1, 25)	(1, 13)
$d_{2,8}$	(1, 15)	(1, 13)		

Fig. 5. A subgraph representing the relationship between the data blocks and the appropriate computing nodes.

The capacity and cost of each edge are set as follows. For each directed edge (d_i, n_m) connected from d_i to n_m, the capacity $a(d_i, n_m)$ is set to 1 and the cost $c(d_i, n_m)$ is set to $T_{benefit} + \alpha$, where $T_{benefit}(n_m, d_i)$ is the benefit time estimated by Eq. (3) and α is the maximal disk access time plus 1 estimated by Eq. (13).

4.3.2 The Second Step

The computing nodes have to reserve memory space to cache the prefeched data blocks. In the SADP, the available memory space of each computing node is represented by several memory blocks, where each memory block can cache a data block with the maximal size. The memory blocks will be used first for data prefetching. If the memory blocks are not available on a node, some data blocks cached in memory need to be replaced by the prefetched data blocks. In the initial phase of the heuristic algorithm, the evictable data blocks are found. Based on the locations of evictable data blocks and the memory blocks, we can extend the graph of Fig. 5. As shown in Fig. 6, each computing node is connected to the evictable data blocks if the node caches them. Note that, each memory block is connected with the node where it is located. The node n_4 also connects to a memory block m_1. For each new edge from n_m in N to d_j in D_e, the capacity $a(n_m, d_j)$ is set to 1 and the cost $c(n_m, d_j)$ is set to 0. For the memory blocks, each edge from n_m in N to m_k in M, the capacity $a(n_m, m_k)$ is set to 1 and the cost $c(n_m, m_k)$ is set to 0.



Settings of (capacity, cost)

	n_1	n_2	n_3	n_4
$d_{1,6}$	(1, 15)			
$d_{1,7}$			(1, 5)	(1, 3)
$d_{2,7}$			(1, 25)	(1, 13)
$d_{2,8}$	(1, 15)	(1, 13)		
$d_{1,2}$	(1, 0)			
$d_{2,1}$	(1, 0)			
$d_{1,1}$		(1, 0)		
$d_{1,3}$		(1, 0)		
$d_{2,2}$			(1, 0)	
$d_{2,3}$			(1, 0)	
$d_{2,4}$				(1, 0)
m_1				(1, 0)

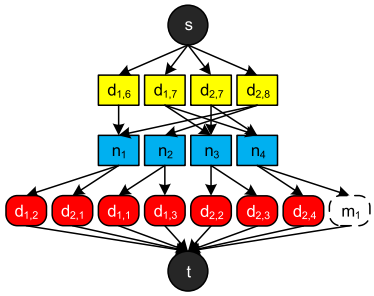
Fig. 6. A subgraph established by the second step.

4.3.3 The Third Step

In the third step, a source node s and a sink node t are added to complete the network flow graph. The source node s is connected to each prefetched data block d_i in D_p. For each edge (s, d_i) connected from s to d_i, the capacity $a(s, d_i)$ is set to 1 and the cost $c(s, d_i)$ is set to 0. For each edge (d_j, t) connected from d_j to t, the capacity $a(d_j, t)$ is set to 1, the cost $c(d_j, t)$ is set to the negative of $T^{disk}_{access}(n_o, d_o)$, where n_o is the computing node which caches the data block d_o, and the d_o is the original copy of d_j stored in hard disk drives. If the data block d_j doesn't have to store back to the hard disk drives, the cost $c(d_j, t)$ is set to 0. By adding the source node s and the sink node t, a network flow graph can be constructed as shown in Fig. 7.

4.3.4 Solving the Maximum-Cost Maximum-Flow Problem

The maximum-cost maximum-flow problem is a variation of the well-known minimum-cost maximum-flow problem. Several algorithms have been proposed to solve the problem in polynomial time [31], [32], [33]. It has been known that solution guarantees that the total cost and the amount of flow passed from the source node s to the sink node t are maximum. Therefore, the total benefit time and the number of prefetched data blocks can be maximized by transforming the optimal solution of the maximum-cost maximum-flow problem. If a unit of flow is transmitted via $d_i \in D_p$, $n_m \in N$, and $d_j \in D_e$, it represents that the data block d_i is decided to be prefetched to the memory of node n_m, and the data block d_j cached in node n_m is evicted. Otherwise, if a unit of flow is transmitted via $d_i \in D_p$, $n_m \in N$, and $m_k \in M$, the data prefetching of data block d_i will occupy the memory block m_k of node n_m. Due to the cost settings of each edge connected to the sink node t, the amount of flow prefers to pass via the memory blocks. According to the optimal solution of Fig. 7, the data blocks $d_{1,6}$, $d_{1,7}$, $d_{2,8}$, and $d_{2,7}$ are respectively prefetched to the nodes n_1, n_3, n_2, and n_4. The memory block m_1 is occupied by the $d_{2,7}$. The data blocks $d_{2,1}$, $d_{1,3}$, and $d_{2,3}$ are evicted from memory.



Settings of (capacity, cost)

	n_1	n_2	n_3	n_4	s	t
$d_{1,6}$	(1, 15)				(1, 0)	
$d_{1,7}$			(1, 5)	(1, 3)	(1, 0)	
$d_{2,7}$			(1, 25)	(1, 13)	(1, 0)	
$d_{2,8}$	(1, 15)	(1, 13)			(1, 0)	
$d_{1,2}$	(1, 0)					(1, 0)
$d_{2,1}$	(1, 0)					(1, -20)
$d_{1,1}$		(1, 0)				(1, 0)
$d_{1,3}$		(1, 0)				(1, 0)
$d_{2,2}$			(1, 0)			(1, -40)
$d_{2,3}$			(1, 0)			(1, -10)
$d_{2,4}$				(1, 0)		(1, -20)
m_1				(1, 0)		(1, 0)

Fig. 7. The network flow graph.

Comparing to the original heuristic algorithm, the enhanced version can prefetch more data blocks $d_{1,7}$ and $d_{2,8}$.

Note that, if the input split size of a Spark job is set to 4MB and the input split of its corresponding data block is 64 MB, the proposed mechanisms determine whether the whole 64 MB of data block can be prefetched to an appropriate computing node in time. If the data block satisfies the prefetching conditions, it will be accessed from the external storage and then be divided into 4MB of input splits for the Spark job. If a task will be launched, its corresponding data block will be checked whether it can be prefetched in time with considerations of the data block size and the available memory space. In the enhanced version of heuristic algorithm, different sizes of data blocks are treated as the maximum size of data blocks. For example, 64 MB and 128 MB of data blocks are treated as 128 MB of data blocks. The reason is that each data block is mapping to a unit flow in the minimum-cost maximum-flow problem. It causes some 64 MB of data blocks cannot be selected to perform prefetching due to the limitation of available memory size. Therefore, the enhanced version is recommended to be used when the data block sizes are the same.

5 Evaluation

In this section, we show our testbed experimental results. The experiments were conducted under heterogeneous environments to evaluate our Scheduling-Aware Data Prefetching mechanisms.

5.1 Experimental Environments

We establish our testbed environments by using XenServer virtualization software [34] to configure 25 virtual machines on 6 physical machines. The proposed data prefetching mechanisms are implemented by modifying source code of Spark 1.6.1, Alluxio 1.2.0, and Hadoop 2.7.2. All

TABLE 2
Types of Virtual Machines

Type	vCPU	Memory	Disk	# VMs hosted on each PM			
				PM 1	PM 2	PM 3	PM 4
1	1	2 GB	128 GB	3	5	1	3
2	2	4 GB	256 GB	0	1	5	0
3	4	8 GB	512 GB	0	0	1	5
4	4	16 GB	1024 GB	0	0	0	1

experiments run on Spark jobs using Alluxio as the external storage system and HDFS as the under storage system. Due to continuous upgrades of infrastructures [35], node heterogeneity has become inevitable in cloud environment. We use four types of physical machines to establish a heterogeneous testbed. In the testbed, three physical machines are the first type and each is with 2 cores 2.5 GHz processor, 8 GB of memory, and 500 GB of disk. The other three physical machines are the second type, third type and forth type, respectively. The second type is with 2 cores 2.50 GHz processor, 16 GB of memory, 1 TB of disk. The third type is with quad core 3.30 GHz processor, 32 GB of memory, and 2 TB of disk. The forth type with quad core 4.00 GHz processor, 64 GB of memory, and 4 TB of disk. The network connection between machines is Gigabit Ethernet. Furthermore, four types of virtual machines are configured with considerations of heterogeneity, as shown in Table 2. The virtual machines of type 1 are with the smallest computing capacities and storage, which can be fitted into more physical machines. Virtual machines of type 2 and type 3 are assumed to be the machines with common computing capacities and storage used in cloud. The virtual machine of type 4 is of the highest computing and storage capacities for special requirements. The virtual machine placements are set manually to use all resources of physical machines. In each virtual machine, 25 and 25 percent of the memory are allocated for Spark and Alluxio, respectively. For the aspect of workloads, WordCount, TeraSort, PageRank, and K-means are common benchmarks [36], [37], [38], [39] to represent different kinds of workloads in a big data processing system. To accurately estimate the data access time, data transmission time, and the task remaining time, we run these benchmarks in advance to record relevant parameters, such as data access rates, data transmission rates and data processing rates. In each experimental runs, these parameters are also reported to the master node for estimating real-time performance. To predict task assignments, a predictor is implemented to receive the status report from computing nodes and run a virtual scheduler when the system status is updated. Once the task assignments are predicted, the mechanisms determine which data blocks can be prefetched in time and witch data blocks cached in memory should be evicted.

In each experimental run, we generated 40 to 80 jobs from the benchmark jobs and submitted to the system in a random order. Each benchmark job is with five jobs by varying sizes of input datasets as 200 GB, 400 GB, 800 GB, 1600 GB, and 3200 GB. The five datasets are divided into data blocks of size 64 MB, 64 MB, 128 MB, 128 MB and 128 MB, respectively. Note that, a dataset can be accessed by multiple jobs simultaneously. After above settings, 100 experimental runs are performed. We concern the following metrics in each testbed run.

- Total job execution time: the sum of execution time of all jobs.
- Data prefetching rate: the number of prefetched data blocks divided by the number of launched tasks. If a data block has already been cached in memory when its corresponding task is launched, the data block is determined to be prefetched successfully.
- Local access rate: the number of local-access tasks divided by the number of launched tasks.
- Algorithm computational time: the time caused by performing the prefetching algorithm.

5.2 Experimental Results

In experimental results, we compare our proposed data prefetching mechanisms with four previous mechanisms: default data caching used in Alluxio (DEFAULT) [5], PACMAN [19], High Performance Scheduling Optimizer (HPSO) [21], and Taming Non-local Stragglers (TNLS) [20]. Like our proposed SADP, data prefetching is also concerned by PACMAN, TNLS, and HPSO. However, PACMAN, HPSO, and TNLS are proposed for MapReduce framework. To compare with these existing mechanisms, the Hadoop 2.7.2 is modified to implement PACMAN, HPSO, TNLS, and our proposed SADP. The modified Hadoop is able to estimate the task remaining time and data access time. Based on the scheduling information of Hadoop, the mechanisms are implemented by calling Alluxio to cache the specific data. In addition to the intermediate output data of map tasks, the proposed SADP is able to prefetch and evict the input and output data blocks of a MapReduce job to/from memory of the external storage.

We propose three mechanisms to solve the data prefetching problem. As mentioned in Section 4.1, the proposed canonical form of Integer Linear Programming is used to obtain the optimal solution of data prefetching problem. The proposed optimal solution is called O_SADP. However, the proposed O_SADP is not suitable to deal with big datasets on a large-scale cloud data center. In Section 5.3, we particularly compare O_SADP with the other proposed mechanisms in a simulation environment. A heuristic algorithm is proposed to quickly solve the data prefetching problem, called H_SADP. H_SADP is of eviction mechanism and is also an extension version of data prefetching mechanism we proposed in [7]. To enhance the H_SADP, we further transform the data prefetching problem to a maximum-cost maximum-flow problem. By solving the network flow problem, we can prefetch more data blocks in memory. To fully understand advantages and disadvantages of the enhancement, the enhanced version of H_SADP is also performed in experiments, called E_SADP. As for other comparisons, the proposed mechanisms are able to improve the execution performance of multiple jobs. We adopt FIFO and Fair schedulers to schedule the job workloads.

Fig. 8 shows the comparison of the total job execution time in testbed environments with FIFO scheduler based on average case, best case, and worst case. Here, the total job execution time of each mechanism is normalized against the average total job execution time of DEFAULT. Comparing

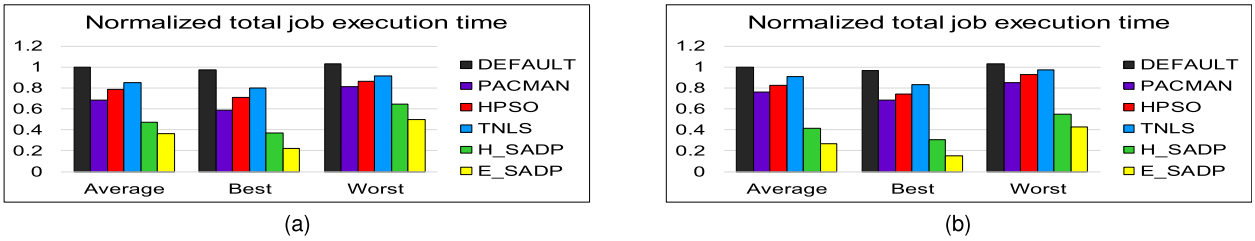


Fig. 8. Normalized total job execution time under FIFO scheduling. (a) 40 jobs. (b) 80 jobs.

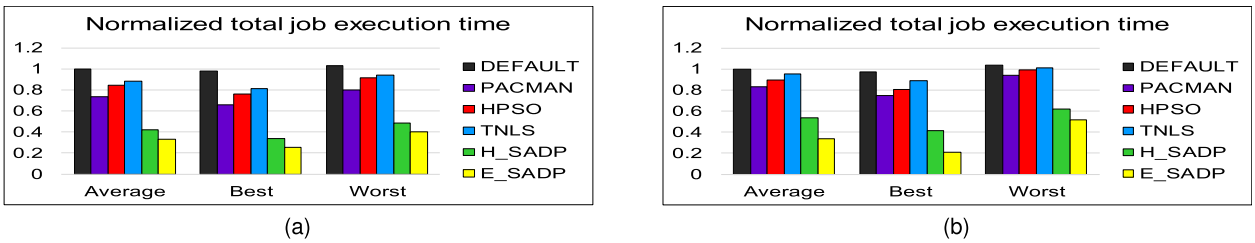


Fig. 9. Normalized total job execution time under Fair scheduling. (a) 40 jobs. (b) 80 jobs.

with DEFAULT, PACMAN, HPSO, TNLS, H_SADP, and E_SADP, can achieve the average improvement of the total job execution time about 31, 22, 13, 55, and 63 percent in 40 jobs. In 80 jobs, PACMAN, HPSO, TNLS, H_SADP, and E_SADP achieve the average improvement of the total job execution time about 22, 17, 9, 59, and 75 percent. As mentioned in Section 4.2.2, the proposed mechanisms allow prefetching data blocks to the nodes different from where the pending tasks will be launched. Comparing with previous mechanisms, H_SADP and E_SADP have lower total job execution time. DEFAULT mechanism caches data in memory only when the data is accessed. Without data prefetching, DEFAULT has the largest total job execution time. Although PACMAN, HPSO, and TNLS support data prefetching, the prefetching deadline is not taken into account. If a task is launched before completion of caching its input data block, it will access the original copy from the disk. Additional workloads of unnecessary data prefetching can even prolong the job execution time. As shown in Fig. 8, the total job execution times of PACMAN, HPSO, and TNLS are longer than H_SADP and E_SADP on average. As increasing the number of jobs, the proposed mechanism can gain more improvement. E_SADP is 76 and 64 percent of H_SADP in Fig. 8a and 8b, respectively. E_SADP transforms the data prefetching problem to maximum-cost maximum-flow problem, which can maximize the total benefit time and the number of prefetched data blocks.

Fig. 9 shows the comparison of the total job execution time with Fair scheduler in 40 jobs and 80 jobs, respectively. Comparing to DEFAULT, the other mechanisms PACMAN, HPSO, TNLS, H_SADP, and E_SADP respectively improve 26, 16, 12, 56, and 65 percent of the total job execution time in the 40 jobs, as shown in Fig. 9a. With multiple jobs running simultaneously on the system, the jobs will incur longer execution time due to resource contention. In 80 jobs, the total job execution time of PACMAN, HPSO, TNLS, H_SADP, and E_SADP achieve 85, 90, 96, 52, and 32 percent of DEFAULT on average, as shown in Fig. 9b. In SADP, the data eviction policy is aware of which data blocks cached in memory that will not be used. By releasing memory space of these unused data blocks, SADP can achieve better memory usage. As shown in Fig. 9, the total job execution times of H_SADP and E_SADP are shorter than PACMAN, HPSO, and TNLS.

Figs. 10 and 11 depict the comparisons of the data prefetching rates under FIFO and Fair schedulers, respectively. As shown in Fig. 10a, H_SADP and E_SADP have the highest data prefetching rate. H_SADP is a heuristic algorithm which cannot guarantee to achieve optimal solution. As a result, H_SADP is 82 percent of E_SADP. PACMAN is respectively 53 and 45 percent of H_SADP and E_SADP. In PACMAN, the scheduling information is not taken into account in eviction mechanism. The data blocks that will be used may be evicted from memory. Although it can still prefetch the data blocks evicted, it cannot promise that all evicted useful data block will be taken back in time. Without eviction mechanisms, HPSO and TNLS are 82 and 65 percent of PACMAN, where TNLS has lower than HPSO because it only takes the straggler tasks into account. As shown in Fig. 10a, the data prefetching rate of PACMAN, HPSO, TNLS, H_SADP, and E_SADP can achieve 2.4, 1.9, 1.6, 4.8, and 5.7 times of DEFAULT in 40 jobs. In Fig. 10b, PACMAN, HPSO, TNLS, H_SADP, and E_SADP can

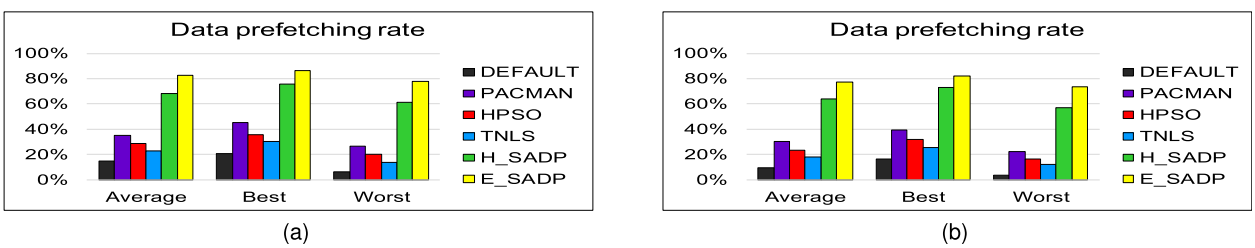


Fig. 10. Data prefetching rate under FIFO scheduling. (a) 40 jobs. (b) 80 jobs.

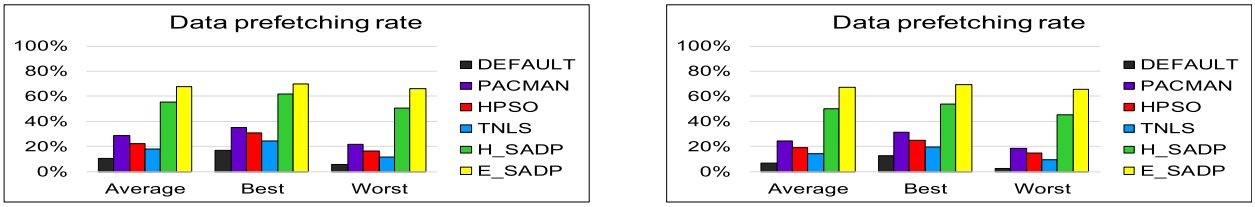


Fig. 11. Data prefetching rate under Fair scheduling. (a) 40 jobs. (b) 80 jobs.

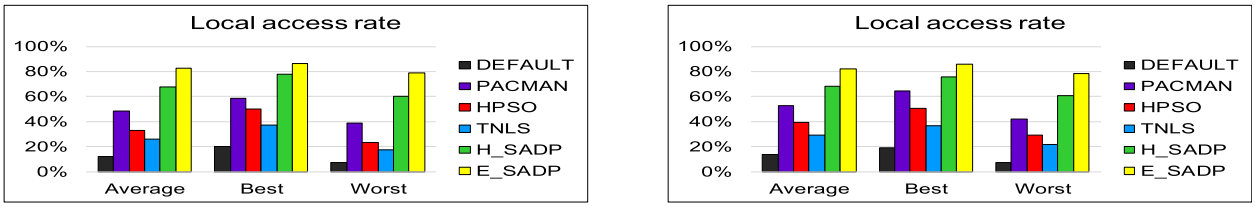


Fig. 12. Local access rate under FIFO scheduling. (a) 40 jobs. (b) 80 jobs.

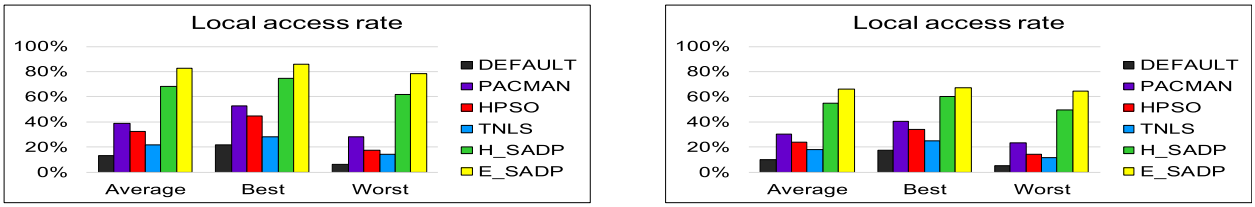


Fig. 13. Local access rate under Fair scheduling. (a) 40 jobs. (b) 80 jobs.

achieve the data prefetching rate of DEFAULT by about 3.2, 2.5, 1.9, 6.8, and 8.4 times, respectively.

The Fig. 11 illustrates the comparisons of the data prefetching rate under Fair scheduler. Compared to scenario where FIFO scheduler is used, multiple jobs running on the cluster have higher probabilities to access the same data blocks cached in memory. As shown in Fig. 11, with consideration of the scheduling and data locality information, the proposed H_SADP and E_SADP can prefetch more data blocks than other previous mechanisms. E_SADP is able to prefetch a data block to the memory of a remote computing node from where the corresponding task will be launched. Therefore, the data prefetching rate of E_SADP can be increased by 23 and 35 percent of H_SADP in 40 and 80 jobs, respectively. As shown in Fig. 11a, PACMAN, H_SAPD, and E_SADP with eviction mechanisms can achieve 1.8, 4.4, and 5.6 times of data prefetching rate of DEFAULT. Without consideration of prefetching deadline and benefit time, PACMAN, HPSO, and TNLS only achieve by 42, 33, and 26 percent of data prefetching rate of E_SADP. In 80 jobs, the data prefetching rate of PACMAN, HPSO, and TNLS achieve by 35, 26, and 20 percent of E_SADP, as shown in Fig. 11b.

Figs. 12 and 13 exhibit the local access rates. Unless DEFAULT, TNLS has lower average local access rates than other mechanisms (PACMAN, HPSO, H_SADP, E_SADP). The reason is that TNLS only concerns data locality of the straggler tasks. However, the number of straggler tasks is much less than the total number of tasks running on the cluster. HPSO applies data prefetching to improve data locality of normal tasks. As a result, comparing with TNLS, HPSO has higher local access rates. With eviction policy, PACMAN has highest local access rates among the previous mechanisms. For our proposed mechanisms, both of H_SADP and E_SADP are higher than PACMAN. When using FIFO, the local access rates of PACMAN only achieves 71 and 58 percent of H_SADP and E_SADP, respectively, as shown in Fig. 12a. Although, E_SADP has lower local access rates than H_SADP, more data blocks can be prefetched in time. In 80 jobs, the proposed H_SADP and E_SADP are about 1.3 and 1.6 times of local access rates of PACMAN, as shown in Fig. 12b. When using Fair scheduler, more tasks cannot access data blocks from local memory or disks due to resource contention. Fig. 13 shows the local access rates using Fair scheduler.

The proposed H_SADP and E_SADP still have higher local access rates. In above experiments, the average computational time of PACMAN, HPSO, TNLS, H_SADP, and E_SADP is 0.0034s, 0.0028s, 0.0018s, 0.0016s, and 0.0052s in 40 jobs, respectively. DEFAULT passively caches data blocks when data accessing, so there is no computational overhead. The computational times of PACMAN, HPSO, and TNLS are lower but close to H_SADP. E_SADP has the highest computational time. Comparing with H_SADP, it increases 68 percent. The following section

5.3 Evaluation of Algorithm Computational Time

In this section, we performed simulations to evaluate our proposed mechanisms. In the simulation experiments, we used MatLab [40] to evaluate the computational time of O_SADP, H_SADP, and E_SADP. The simulations were conducted on a physical server with 2.50 GHz processor, 16 GB of memory, 1 TB of disk. We assume that 50 and 100

jobs run on 350-node and 3500-node cloud data centers. Each job is associated with [8TB, 16 TB, 32 TB] of input dataset, and each computing node is associated with 8 executors to run the tasks in parallel. The input dataset consists of 128 MB, 128 MB, and 256 MB data blocks, respectively. The initial number of available executors is randomly ranged from 25 to 75 percent of executors in the system. The tasks in a job is associated with a processing speed t_p ranged from 0.5 to 5. The execution time of a task t is assumed to be t_p divided by the size of its data block. Totally 100 runs are performed in the simulations.

The average algorithm computational time of the proposed O_SADP, H_SADP, and E_SADP is respectively 0.0847s, 0.0056s, and 0.0144s in 50 jobs. In 100 jobs, the computational time of O_SADP, H_SADP, and E_SADP is 4.1526s, 0.0749s, and 0.2342s, respectively. O_SADP is longer than H_SADP and E_SADP. The reason is that O_SADP needs to calculate the all conditional equations for obtaining optimal solution. If O_SADP is applied in a large-scale data center, the deadline violation of data prefetching can be increased due to the long computational time. To adapt SADP in cloud, a greedy method of H_SADP is used for reducing the computational time. By sacrificing the optimal solution, the computational time of H_SADP is about 0.0056s and 0.0749s, respectively. To boost the benefit time of H_SADP, the data prefetching problem is transformed to a maximum-cost maximum-flow problem. By solving the maximum-cost maximum-flow problem, the number of prefetched data blocks can be effectively increased and then the benefit data access time can be increased. Additionally, comparing with O_SADP, E_SADP also has a relatively lower computational time. In 100 jobs, the computational time of E_SADP is about 5 percent of O_SADP. Compare to H_SADP, E_SADP only increases 3.12 times of computational time. However, the computational time is only about 0.2342s.

6 Conclusion

We have investigated the data prefetching problem in a large-scale cloud data center. Considering each computing node in the cloud has limited memory resources, we provide the Scheduling-Aware Data Prefetching to accelerate the progress of big data processing. First, we formulate a canonical form of Integer Linear Programming for obtaining optimal solution. To make the data prefetching mechanism accommodate to a large-scale cloud data center, we also propose a heuristic algorithm to prefetch and evict data to/from memory in accordance with job scheduling information. To increase the data prefetching rates, we also proposed an enhanced version of the heuristic algorithm. The SADP has been implemented in a real-testbed. The evaluation results show that the proposed mechanisms can efficiently perform the data prefetching in big data processing. Comparing with the default mechanism used in Alluxio, the proposed mechanism can achieve at least 29 percent reduction of the total job execution time in heterogeneous environment.

Acknowledgments

This research was supported by the Ministry of Science and Technology, Taiwan under Grant MOST 103-2221-E-001-028-MY3 and MOST 105-2221-E-002-119-MY3.

References

[1] M. Zwolenski and L. Weatherill, "The digital universe rich data and the increasing value of the internet of things," *Australian J. Telecommunications Digital Economy*, vol. 2, Apr. 2014, doi: 10.7790/ajtde.v2n3.47.

[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–15.

[3] H. Karau, A. Kowinski, and M. Hamstra, *Learning Spark: Lightning-fast Big Data Analysis*. Newton, MA, USA: O'Reilly Media, Inc., 2015.

[4] Apache Spark™- Lightning-Fast Cluster Computing, 2018. [Online]. Available: http://spark.apache.org/

[5] Alluxio - Open Source Memory Speed Virtual Distributed Storage, 2018. [Online]. Available: http://www.alluxio.org/

[6] Welcome to Apache™Hadoop®!, 2018. [Online]. Available: http://hadoop.apache.org/

[7] C.-H. Chen, T.-Y. Hsia, Y. Huang, and S.-Y. Kuo, "Scheduling-aware data prefetching for data processing services in cloud," in *Proc. IEEE 31st Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2017, pp. 835–842.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Networked Syst. Des. Implementation*, 2012, pp. 2–2.

[9] X. Shi, M. Chen, L. He, X. Xie, L. Lu, H. Jin, Y. Chen, and S. Wu, "Mammoth: Gearing hadoop towards memory-intensive MapReduce applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2300–2315, Jul. 2014.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[11] B. Ooi, Y. Wang, Z. Xie, M. Zhang, K. Zheng, K. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, and A. Tung, "SINGA: A distributed deep learning platform," in *Proc. 23rd ACM Int. Conf. Multimedia*, 2015, pp. 685–688.

[12] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, 2010, pp. 170–177.

[14] R. Gandhi, A. Gupta, A. Povzner, W. Belluomini, and T. Kaldewey, "Mercury: Bringing efficiency to key-value stores," in *Proc. 6th Int. Syst. Storage Conf.*, 2013, pp. 6:1–6:6.

[15] H. Lim, D. Han, D. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proc. 11th USENIX Conf. Networked Syst. Des. Implementation*, 2014, pp. 429–444.

[16] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2013, pp. 103–114.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Networked Syst. Des. Implementation*, 2011, pp. 295–308.

[18] V. Vavilapalli, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, and H. Shah, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annual Symp. Cloud Comput.*, 2013, pp. 5:1–5:16.

[19] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory caching for parallel jobs," in *Proc. 9th USENIX Conf. Networked Syst. Des. Implementation*, 2012, pp. 267–280.

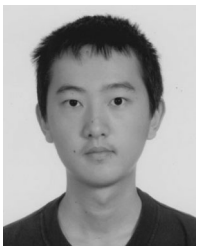
[20] Z. Yu, M. Li, X. Yang, H. Zhao, and X. Li, "Taming non-local stragglers using efficient prefetching in MapReduce," in *Proc. 2015 IEEE Int. Conf. Cluster Comput.*, 2015, pp. 52–61.

[21] M. Sun, H. Zhuang, C. Li, K. Lu, and X. Zhou, "Scheduling algorithm based on prefetching in MapReduce clusters," *Appl. Soft Comput.*, vol. 38, pp. 1109–1118, Jan. 2016.

[22] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *ACM Commun.*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[23] Apache Hadoop 2.7.2 HDFS Users Guide, 2018. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
[24] Welcome to Swift's documentation!, 2018. [Online]. Available: http://docs.openstack.org/developer/swift/
[25] Amazon Simple Storage Service (Amazon S3), 2018. [Online]. Available: https://aws.amazon.com/s3
[26] T. White, *Hadoop: The Definitive Guide*, 4th ed. Boston, MA, USA: O'Reilly, 2015.
[27] C. Jin, R. Kang, and R. Li, "VTB-RTRRP: Variable threshold based response time reliability real-time prediction," *IEEE Access*, vol. 6, pp. 60–71, 2018, doi: 10.1109/ACCESS.2017.2741666.
[28] C.-H. Chen, J.-W. Lin, and S.-Y. Kuo, "MapReduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems," *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, pp. 127–140, Jan. 2018.
[29] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[30] R. M. Karp, "Reducibility among combinatorial problems," in *Proc. Symp. Complexity Comput. Comput.*, 1972, pp. 85–103.
[31] K. Zhang, "A constrained edit distance between unordered labeled trees," *Algorithmica*, vol. 15, no. 3, pp. 205–222, Mar. 1996.
[32] C.-X. Xu, "A simple solution to maximum flow at minimum cost," in *Proc. 2nd ICIECS*, 2010, pp. 1–4.
[33] Z. Han, H. Tan, Y. Wang, and J. Zhou, "Channel selection for rendezvous with high link stability in cognitive radio network," in *Proc. 9th Int. Conf. Wireless Algorithms Syst. Appl.* - Vol. 8491, 2014, pp. 494–506.
[34] XenServer—Open Source Server Virtualization, 2018. [Online]. Available: http://xenserver.org/
[35] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou, "Improving performance of heterogeneous MapReduce clusters with adaptive task tuning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 774–786, Mar. 2017.
[36] J. A. Issa, "Performance evaluation and estimation model using regression method for hadoop wordcount," *IEEE Access*, vol. 3, pp. 2784–2793, 2015, doi: 10.1109/ACCESS.2015.2509598.
[37] R. Moussa, "Benchmarking data warehouse systems in the cloud," in *ACS Int. Conf. Comput. Syst. Appl.*, May 2013, pp. 1–8.
[38] P. Dreher, C. Byun, C. Hill, V. Gadepally, B. Kuszmaul, and J. Kepner, "PageRank pipeline benchmark Proposal for a holistic system benchmark for big-data platforms," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2016, pp. 929–937.
[39] S. E. Mendili, Y. E. B. E. Idrissi, and N. Hmina, "Benchmarking study on smart city data analytics," in *Proc. IEEE Int. CiSt*, Oct. 2016, pp. 841–846.
[40] MathWorks®, 2018. [Online]. Available: http://www.mathworks.com/

Chien-Hung Chen received the BS and the MS in computer science & information engineering from Chung Hua University and Fu Jen Catholic University, in 2008 and 2012, respectively and the PhD degree in electrical engineering from National Taiwan University, in 2018. He is a technical supervisor with AVerMedia Technology Inc. He received a fellowship as a visiting PhD student at Technical University of Darmstadt in Germany from 2014 to 2015. His research interests include deep learning, big data, cloud and edge computing, distributed systems, and internet of things. His HungYang IoT Team received the award from 2016 FITI Program launched by Ministry of Science and Technology (MOST). He took the 2nd place in the 2017 Intern Closing Competition (R&D Group) from Taiwan Semiconductor Manufacturing Company (TSMC). In 2018, he received the Best PhD Dissertation Award from the Graduate Institute of Electrical Engineering, National Taiwan University.



Ting-Yuan Hsia received the BS and MS degrees from the Department of Electrical Engineering, National Taiwan University, in 2014 and 2016. His current research interests include cloud computing, dependable distributed systems, and fault-tolerant computing.

Yennun Huang received the BS degree in EE from National Taiwan University and the PhD degree in computer science from the University of Maryland. He is a distinguished research dellow with Academia Sinica and the director of Research Center for Information Technology Innovation (CITI), Academia Sinica. He is currently the chairman of Internet of Vehicles Committee of Taiwan IoT Technology and Industry Association (TwIoTA). He Joined AT&T Bell Labs as a researcher in 1989 and became a distinguished member of Technical Staff of Bell Labs in 1996. He started the Dependable Computing Research Department in AT&T in 1999 and was the department head of the organization to ensure the high dependability of all AT&T services. He became the VP of Engineering of PreCache Inc, a Sony subsidiary, in 2001 to create a multi-media content delivery service. In late 2004, He became the AT&T Labs executive director of Dependable Distributed Computing and Communication Research Department to lead research on Digital Content Management and IPTV programs. In 2007, he returned to Taiwan and became anexecutive vice president of Institute for Information Industry (III). From 2008 to 2011, He was the president of VeeTIME Co. to create quadruple-play telecom services including cable TV, FTTx, NGN voice and 4G WiMax. He has published numerous papers in major journals and conferences, and more than 20 US patents awarded. He joined Research Center for Information Technology Innovation (CITI) of Academia Sinica in 2011 as the CEO of Security Research Center in Academia Sinica. He also served the Board of Science and Technology (BOST) of Executive Yuan as a deputy executive secretary between 2011 and 2015 to help Taiwan Government on the R&D strategy and budget allocation for Information and Communication Technology (ICT) Development. He is a fellow of the IEEE.

Sy-Yen Kuo received the BS degree in electrical engineering from National Taiwan University, in 1979, the MS degree in electrical & computer engineering from the University of California at Santa Barbara, in 1982, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign, in 1987. He is the Pegatron chair professor with the Department of Electrical Engineering, National Taiwan University (NTU), Taipei, Taiwan. He was the dean of College of Electrical Engineering and Computer Science, NTU from 2012 to 2015 and the chairman of Department of Electrical Engineering in NTU from 2001 to 2004. He also took a leave from NTU and served as a chair professor and dean of the College of Electrical Engineering and Computer Science, National Taiwan University of Science and Technology from 2006 to 2009. He spent his sabbatical years as a visiting professor with Hong Kong Polytechnic University from 2011-2012 and with the Chinese University of Hong Kong from 2004-2005, and as a visiting researcher with AT&T Labs-Research, New Jersey from 1999 to 2000, respectively. He was a faculty member with the Department of Electrical and Computer Engineering, University of Arizona from 1988 to 1991, and an engineer with Fairchild Semiconductor and Silvar-Lisco, both in California, from 1982 to 1984. In 1989, he also worked as a summer faculty fellow with Jet Propulsion Laboratory of California Institute of Technology. His current research interests include dependable and secure systems, edge and cloud computing, internet of things, quantum computing. He has published 450 papers in journals and conferences, and also holds 22 US patents, 23 Taiwan patents, and 15 patents from other countries. He received the Distinguished Research Award and the Distinguished Research Fellow award from the National Science Council, Taiwan. He was also a recipient of the Best Paper Award in the 1996 International Symposium on Software Reliability Engineering, the Best Paper Award in the simulation and test category at the 1986 IEEE/ACM Design Automation Conference (DAC), the National Science Foundation's Research Initiation Award in 1989, and the IEEE/ACM Design Automation Scholarship in 1990 and 1991. He is a fellow of the IEEE, and a member of the IEEE Fellow Committee from 2018-2020. He is also a core member and a member of the Board of Governors of IEEE Computer Society from 2017-2020.