The Journal of Systems and Software 151 (2019) 133–149

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Data prefetching and file synchronizing for performance optimization in Hadoop-based hybrid cloud

Chunlin Li[a,c,d,*], Jing Zhang[a,b], Yi Chen[c], Youlong Luo[a]

[a] School of Computer Science and Technology, Wuhan University of Technology, Wuhan 430063, China
[b] International College, Huanghuai University, Zhumadian 463000, China
[c] Beijing Key Laboratory of Big Data Technology for Food Safety, Beijing Technology and Business University, Beijing, China
[d] Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an University of Posts and Telecommunications, Xi'an, Shaanxi 710121, China

ARTICLE INFO

Article history:
Received 7 April 2018
Revised 10 October 2018
Accepted 4 February 2019
Available online 8 February 2019

Keywords:
Data prefetching
File synchronizing
Hybrid cloud

ABSTRACT

Driven by the technical factors such as system reliability, bandwidth constraints, data confidentiality and security, as well as the economic factors such as initial capital expenditure and re-occurring operating expenditure, today's cloud computing tends to adopt hybrid cloud model. However, because hybrid clouds scale both numerically and geographically, the network delay becomes the main constraint in remote file system access. To hide network latency and reduce job completion time in Hadoop-based hybrid cloud data access, a scheduling-aware data prefetching scheme to enhance non-local map task's data locality in Hadoop-based centralized hybrid cloud (*CHCDLOS-Prefetch*) and a file synchronizing method to decrease job execution delay in Hadoop-based distributed hybrid cloud (*DHCDLO-Sync*) are proposed. In the former, input data for non-local map tasks are fetched ahead of time to target compute nodes by making use of idle network bandwidth. In the latter, considered from job level scheduling, data files with high popularity are proactively synchronized beforehand among sub-clouds to strength intra sub-cloud data locality in distributed hybrid cloud. Extensive experimental results illustrate that compared to the *Capacity*, the *Fair* and the *DARE* algorithms, our proposed algorithms improve hybrid cloud performance more significantly in data locality and job completion time.

1. Introduction

The data explosion in recent years has led to a widely use of cloud computing in data-intensive applications. Driven by the technical factors such as system reliability, bandwidth constraints, data confidentiality and security, as well as the economic factors such as initial capital expenditure and re-occurring operating expenditure, cloud computing in many cases tends to adopt the hybrid cloud model (Van den Bossche et al., 2013). Hybrid clouds, which combine both public cloud offerings and private infrastructures, have been broadly accepted and used in both industry and research.

However, because hybrid clouds scale both numerically and geographically, the network delay becomes the main constraint in remote file system access (Liao et al., 2017). Tasks scheduled to off-premise resources have to spend long time on waiting for input data retrieving. To solve this problem, many prefetching and synchronizing schemes are introduced to hide the latency in distributed systems caused by network communication (Mansouri and Javidi, 2018; Chunlin Li, Jing Zhang, 2019; Chunlin Li, Tang Jianhang, 2019). Data prefetching tries to predict future access and transfers input data to target nodes in advance, which can reduce execution waiting delay and job completion time. File synchronizing is to transfer the absent files to different web sites of the hybrid cloud, which can increase file replicas and improve sub-clouds' data locality, making files be locally available, thereby avoiding remote data retrieval and improving job execution efficiency.

In this paper, we study data prefetching and file synchronizing techniques in Hadoop-based hybrid cloud. Based on Hadoop platform, we built two kinds of hybrid clouds with different scheduling modes: the centralized hybrid cloud and the distributed hybrid cloud. In centralized hybrid cloud, Hadoop is deployed in private cloud data center, off-premise resources are moved in and out of the hybrid cloud elastically on demand (Wang et al., 2013), and all hybrid cloud resources are managed by a master scheduling server. Because of its good scalability and higher flexibility, this mode of hybrid cloud is widely used in the scenarios with large workload fluctuation and cloud bursting, such as e-business, e-commerce,

* Corresponding author at: Department of Computer, Wuchang, Wuhan 430063, China.
E-mail address: chunlin74@aliyun.com' (C. Li).

https://doi.org/10.1016/j.jss.2019.02.007
0164-1212/

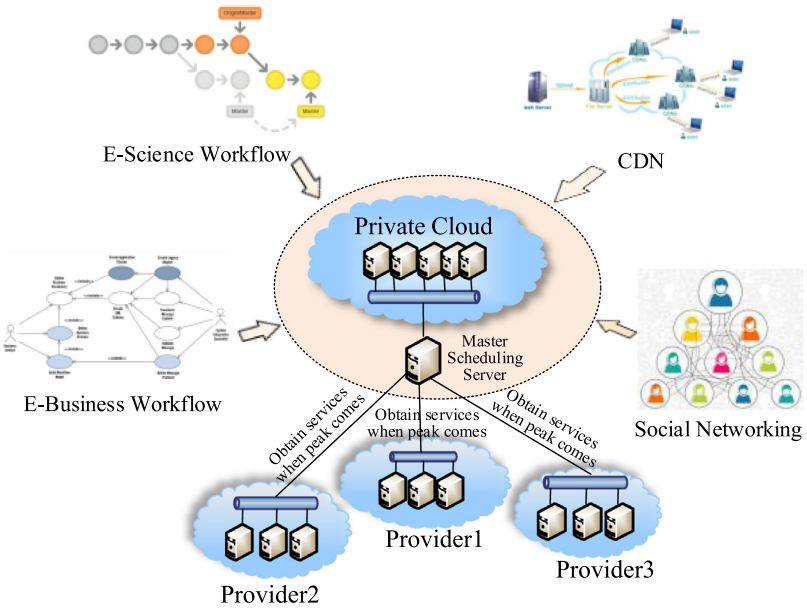


Fig. 1. Application scenarios of centralized hybrid cloud environment.

e-science, on-line gaming, CDN content dissemination and social networking (Kovachev et al., 2014), which is shown as Fig. 1. In distributed hybrid cloud, Hadoop is deployed in each sub-cloud data center of the hybrid cloud, resources of each sub-cloud are managed by an independent master scheduling server, the interaction among sub-clouds is realized through master scheduling servers, and jobs submitted to the hybrid cloud are processed cooperatively by all sub-clouds. Due to its better stability and strong disaster resistance, this mode of hybrid cloud is universally applied in the scenarios of sensor networks, fog computing, web hosting, meteorological data analysis and iterative MapReduce (Clemente-Castelló et al., 2017), which is shown as Fig. 2.

However, no matter the centralized hybrid cloud or the distributed hybrid cloud, they will suffer from network delay in remote file system access. To hide communication latency, utilize available idle bandwidth and reduce job execution waiting time, a scheduling-aware data prefetching method and a file synchronizing approach for data locality in Hadoop-based hybrid cloud are proposed. The main contributions of this work are summarized as follows.

(1) To effectively utilize idle network bandwidth and reduce job execution waiting time, we proposed a scheduling-aware data prefetching method for data locality in Hadoop-based centralized hybrid cloud. In each round scheduling, candidate compute nodes are preselected first. Then, non-node locality map tasks relative to these compute nodes are determined. Finally, according to predefined rules, the input data

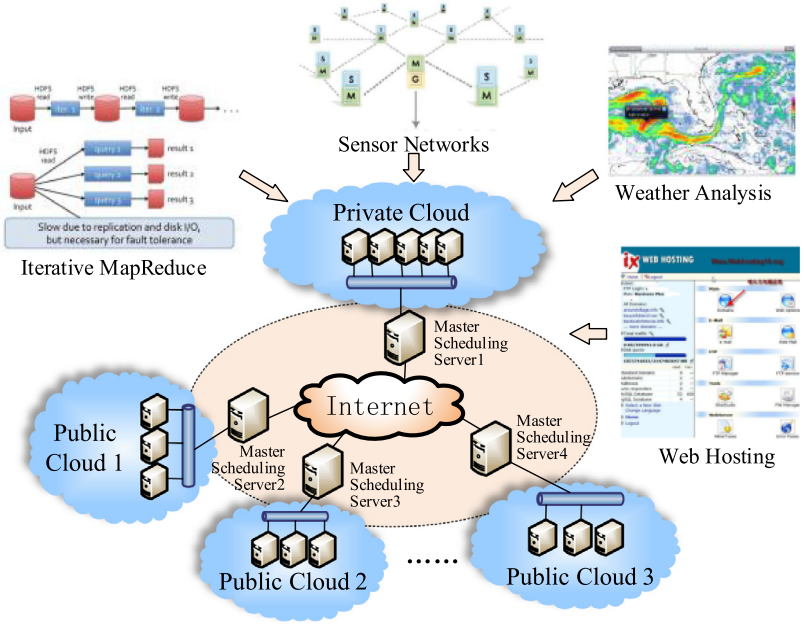


Fig. 2. Application scenarios of distributed hybrid cloud environment.

of qualified non-node locality map tasks are prefetched to target compute nodes ahead of task execution.

(2) To hide network delay caused by cross-cloud data transmission and improve job execution efficiency, a file synchronizing method for data locality in Hadoop-based distributed hybrid cloud is presented. In this method, to avoid unnecessary data transmission for reducing network overhead, only the high popular files are selected as the sync files and synchronized periodically to other sub-clouds of the distributed hybrid cloud.

(3) Through evaluation, the performance of the scheduling-aware data prefetching method and the file synchronizing algorithm in Hadoop-based hybrid cloud are verified. Experimental results demonstrate that compared to the *Capacity, Fair* and *DARE* algorithms, our proposed data prefetching and file synchronizing algorithms improve hybrid cloud performance significantly better in data locality and job completion time.

The remaining of the paper is organized as follows. Some related works are briefly reviewed in Section 2. The model and algorithm implementation of scheduling-aware data prefetching and file synchronizing in hybrid cloud are described in Section 3. Section 4 introduces an application scenario. Section 5 proceeds performance evaluation of our proposed methods. Finally, we conclude the paper in Section 6.

2. Related work

2.1. Data locality optimization methods in cloud

There has been substantial researches on data locality of cloud computing. In Chen et al. (2016), by considering data locality and global data access cost, Chen et al. proposed a topology-aware optimal data placement algorithm to improve the performance of MapReduce in cloud data center. In Chen et al. (2017), a data allocation algorithm with minimum cost for dynamic datacenter resizing was proposed. What related to our work is the prefetching idea. In Seo et al. (2009), Seo et al. presented two optimization schemes, prefetching and pre-shuffling, in the shared MapReduce environment. The former aims to improve the data locality during both map and reduce phases, while the latter focuses on reducing the shuffling overhead during the reduce phase. In Wang and Ying (2016), Wang et al. solved the data locality issues from a network perspective, and proposed a Joint Scheduler to utilize the computing resources and the communication network. There are also methods addressing the data locality problem from the side of distributed file system. Ananthanarayanan et al. (2011) presented an algorithm named Scarlett which replicates data blocks based on their popularity to address the problem of hotspot access bottlenecks. Abad et al. presented DARE (Abad et al., 2011), a distributed adaptive data replication algorithm, to aid the scheduler to achieve better data locality. DARE addresses the issues of how many replicas a file should be allocated and where to place them. These two algorithms replicate data chunks according to the variance in data popularity and access patterns, which share the same spirit as we do in distributed hybrid cloud data locality optimization that makes the popular data accessible to more sub-cloud machines.

2.2. Scheduling algorithms for data locality

Data locality has been widely studied in task scheduling in MapReduce. Delay (Zaharia et al., 2010), a scheduling algorithm was proposed to address the conflict between locality and fairness. This algorithm enhances data locality in map step by delaying the job scheduling for a small amount of time if a task cannot be scheduled to a compute node with local data. Quincy (Isard et al., 2009) introduced a resource allocation framework for scheduling concurrent distributed jobs on clusters which can get better fairness while substantially improving data locality. BAR (Jin et al., 2011) is an efficient data locality driven task scheduler for cloud computing. By heuristic task scheduling, BAR adjusts data locality dynamically according to network state and cluster workload. SLAW (Guo et al., 2010), a scalable locality-aware adaptive work stealing scheduler, aims to address the issues of fixed task scheduling policy and locality-obliviousness in work stealing schedulers. In Wang et al. (2016), Wang et al. proposed a map task scheduling in MapReduce with data locality. This algorithm focus on making the tradeoff between data locality and load balancing to simultaneously maximize throughput and minimize delay from a stochastic network perspective. In Tang et al. (2016), Tang et al. proposed a fairness concurrent distributed job scheduling framework in shared computing systems with high data locality. In Palanisamy et al. (2011), a MapReduce resource allocation system Purlieus was presented to minimize the data transfer overhead among tasks in a data locality-aware manner during both the map and reduce phases. However, if input data is large, it may consume significant waiting time before a task could be scheduled.

2.3. Hybrid cloud construction methods

For centralized hybrid cloud construction, literature (Marshall et al., 2010) affords us a prototype to establish hybrid cloud computing infrastructure which enables an organization to elastically extend its site resources from another cloud provider. In Javadi et al. (2012), a failure-aware resource provisioning algorithm is proposed, which is implemented under a centralized mode hybrid cloud. In Zhang et al. (2014), a hybrid cloud computing model is proposed to intelligently manage the proactive workload. In which, they construct their hybrid cloud by using public cloud services along with their privately-owned (legacy) data centers in a centralized manner. As for distributed hybrid cloud establishment, reference (Di Costanzo et al., 2009) provides an abstract view of the architecture and the implementation of the hybrid cloud. In which, they harnesses the cloud technologies to construct distributed execution environments that span multiple computing sites. In Malawski et al. (2013), Malawski et al. addressed the cost minimization problem for data- and compute-intensive tasks on hybrid cloud under the deadline constraint. They modeled their problems under multiple heterogeneous compute and storage clouds which organized in a distributed manner. In Lu et al. (2015), Lu et al. designed a service provisioning model to manage the resources in the hybrid cloud. In which, they organized their hybrid cloud that consists of multiple geographically distributed cloud data centers in a distributed manner.

3. System model

3.1. Scheduling-aware data prefetching for data locality in centralized hybrid cloud

Trace analysis in some production MapReduce clusters show that most jobs are map-intensive, and many of them are map-only (Kavulya et al., 2010). Therefore this work focuses on performance optimization for map tasks. In this section, the **scheduling-aware data prefetching method for data locality in centralized hybrid cloud** (*CHCDLOS-Prefetch*) is proposed. This method is realized through the following steps: map task remaining execution

time estimation, candidate compute nodes preselection, non-node locality map task preselection and input data prefetching.

3.1.1. Remaining execution time estimation

Usually, the input data of the running map tasks have been acquired, there is no need to retrieve them from remote. Thus, the nodes which the map tasks are running on have available bandwidth. To improve map task process concurrency and take full advantage of bandwidth resources, the available idle bandwidth can be exploited to prefetch input data for other map tasks. In order to ensure the input data transfer can be accomplished before a node releasing its compute resources, the remaining execution time of map tasks running on the node should be estimated before input data prefetching.

The map task execution progress is the ratio of the amount of data that the task has been processed to the total amount of data that the map task has to process. Assume the execution progress of map task j in node i is prg_{ij}, the size of the data processed by map task j is D_{ij}^{read}, the total size of the data that the map task needs to process is D_{ij}^{total}, then we have $prg_{ij} = D_{ij}^{read}/D_{ij}^{total}$. Suppose current system time is T_{ij}^{curr}, the time of task$j$ scheduled to node$i$ is T_{ij}^{schd}, then, map task$j$'s execution rate is $rate_{ij} = prg_{ij}/(T_{ij}^{curr} - T_{ij}^{schd})$. If T_{ij}^{left} denotes the remaining execution time of map task$j$ in node$i$, then,

$$T_{ij}^{left} = prg_{ij}/\left(1 - rate_{ij}\right) = (D_{ij}^{total} - D_{ij}^{read})(T_{ij}^{curr} - D_{ij}^{schd})/D_{ij}^{read} \tag{1}$$

As multi-task are ran on each node, we let the minimum one of the remaining execution time of map tasks in node i be determined as the remaining execution time of node i.

3.1.2. Compute nodes preselection

To realize data prefetching, the nodes which have available idle bandwidth should be selected out. In addition, the case that whether data prefetching can be accomplished before nodes releasing their resources should be considered if prefetching performs on these nodes. If not, bandwidth competition may occur due to tasks delivery to these nodes, this may aggravate network burden. Therefore, the nodes' remaining execution time should be compared with the data block transfer time to determine whether they are selected as the candidate compute nodes.

Due to the variety of the hybrid cloud, the data source node and the target compute node of map tasks may work in different clouds (public or private). Meanwhile, the *reduce* tasks may occupy part of the network bandwidth, thus, the data prefetching can only use the idle bandwidth.

According to cloud type (public or private), network bandwidth BW_i between compute node i and its data source nodes have the value of following cases: ① Data source node and target compute node are both in private cloud, then, BW_i is considered as the communication bandwidth of inside the private cloud, which is expressed as $width_1$. ② Data source node is in the private cloud and target compute node is in the public cloud, or vice versa, then, BW_i is deemed as the communication bandwidth between the private cloud and the public cloud, we express it as $width_2$. ③ Data source node and target compute node are both in the same public cloud, then BW_i is regarded as the communication bandwidth inside the public cloud, which is expressed as $width_3$. ④ Data source node and target compute node belong to different public clouds, then BW_i is considered as the communication bandwidth between the two public clouds, which is expressed as $width_4$. According to above description, the bandwidth between the compute node i and its source data nodes can be formulated as,

$$\begin{cases} BW_i = width_x, x \in [1,4], \ i \in [0,m) \\ width_x \geq 0 \end{cases}. \tag{2}$$

Here, m is the number of the compute nodes, x is the cases of network bandwidth.

According to formula (2), ① if $BW_i = 0$, it denotes there is no available bandwidth between node i and its data source node for data prefetching; ② if $BW_i > 0$, it indicates there is idle bandwidth for data prefetching. Next, data block transfer time should be computed.

Assume the data block size in hybrid cloud is S_{block}, the time-consuming of transmitting a data block to node i from its source data node is $T_i^{perblock}$, then, under the condition of $BW_i > 0$, the data block transfer time from data source node to the target compute node can be computed as formula (3). Here, m is the number of the compute nodes.

$$T_i^{perblock} = S_{block}/BW_i, i \in [0,m) \tag{3}$$

As each data block has multi replicas in the system, thus, the data source nodes of a map task are not unique. Therefore, the data block transfer time from all of the source data nodes to the target compute node should be computed. In addition, all data block transfer time and their corresponding source data node address are saved in a set T_i^{block} for next step data prefetching.

Finally, let the minimum remaining execution time $\min(T_{ij}^{left})$ of node i compare with its maximum data block transfer time $\max(T_i^{perblock})$, if $\max(T_i^{perblock}) < \min(T_{ij}^{left})$, node i will be selected as the candidate compute node and saved in the candidate compute node set *candidateNodes*.

Algorithm 1 depicts the pseudo-code for compute node preselection. The main steps are as follows: ① Bandwidth of each node is calculated in Algorithm 1 lines 2∼3, and the minimum bandwidth between source node and target node is selected as the available bandwidth for data prefetching. ② Per block transfer time is computed in Algorithm 1 line 8. ③ The time left of map tasks in each node is estimated in Algorithm 1 lines 9∼12. ④ Finally, the node which can complete data prefetching before releasing its compute resources is added to *candidateNodes* (Algorithm 1 lines 13∼17).

3.1.3. Map task preselection

In this part, non-node locality map tasks relative to the candidate compute nodes preselected by step of Section 3.1.2 will be screened out from the failed map tasks and the unassigned map tasks of the jobs. The unexecuted and the failed map tasks of current running jobs are regarded as the candidate map tasks. In order to re-schedule the failed map tasks to finish the executing jobs as soon as possible, the failed map tasks' data prefetching should be done in priority.

For each map task in the candidate map task set, if the map task is node locality relative to the candidate compute nodes, this map task will be deleted from the candidate map task set, as this map task does not need data prefetching. These map tasks which have the non-node locality relative to the candidate nodes in *candidateNodes* are selected as the prefetching map tasks. The process of the non-node locality map tasks preselection is as follows.

(1) For failed map tasks in set the of *failedMaps*, the map tasks which have node locality relative to the candidate nodes in *candidateNodes* are deleted from *failedMaps* and the corresponding nodes are deleted from *candidateNodes*, only the non-node locality map tasks are kept left. If there are no candidate nodes left in *candidateNodes*, the process of map task preselection will end.

Algorithm 1 Compute nodes preselection.

Input: All nodes in centralized hybrid cloud cluster {$Node_1$, $Node_2$, ..., $Node_n$}
Output: The preselected candidate nodes $candidateNodes = \{Node_1', Node_2', ..., Node_k'\}$

```
for each Node_i do        //Node_i is computing node
  getCloudType(Node_i); // Obtain Node_i's cloud type (private or public);
  BW_i ← getBandWidth_i (sourceNode_j), j = 1, ..., 4 //Acquire idle bandwidth of Node_i from source nodes
if max(BW_i) = 0 then // Node_i has no idle bandwidth available
    delete(Node_i);
    go to Node_{i+1}
end if
T_i^{perblock} = S_block/max(BW_i) //Per block transfer time to Node_i
for each m_j ∈ Node_i, j = 1..M do // m_j is the running map task j in Node_i
  T_ij^{left} = prg_ij/(1 − rate_ij) = (D_ij^{total} − D_ij^{read})(T_ij^{curr} − D_ij^{schd})/D_ij^{read}
end for
T_i^{left} ←min(T_ij^{left}) //Remaining execution time of Node_i
if T_i^{left} < T_i^{perblock} then
    delete(Node_i)
    go to Node_{i+1}
end if
candidateNodes ← Node_i // Add Node_i to the candidate node set
end for
Sort the nodes in candidateNodes according to their min remaining execution time on ASC
```

(2) For non-node locality failed map tasks, they are sorted according to the required resources, then added to the set of the candidate prefetching map tasks *prefetchMaps*.
(3) For unassigned map tasks in the set of *pendingMaps*, the map tasks which have node locality relative to the candidate nodes in *candidateNodes* are deleted from *pendingMaps* and the corresponding nodes are deleted from *candidateNodes*, only the non-node locality map tasks are left. If there are no candidate nodes left in *candidateNodes*, the process of map task preselection will end.
(4) For the non-node locality map tasks, they are sorted in accordance with their required resources, then added to the end of the candidate prefetching map tasks set *prefetchMaps*.

Algorithm 2 depicts the pseudo-code for map task preselection, which is shown as follows.

3.1.4. Input data prefetching

In order to take full advantage of the network bandwidth in centralized hybrid cloud, in each round prefetching, the data blocks should be pre-fetched as many as possible. The process of data prefetching described as follows.

(1) For each map task in prefetching map task set *prefetchMaps*, the target compute node that meets its required resources is searched from the candidate compute node set *candidateNodes*.
(2) If no compute node is found, this map task will be deleted form *prefetchMaps*, then the compute node for next map task will be found; else, according input data metadata information, the data source node of the map task is found.

As the input data of jobs are stored in distribution in HDFS, each data block has multiple copies distributed over different nodes. Thus, each map task has multiple source data nodes. In data prefetching, the nearest data source node is selected as the source data node. According to the network topology information of Hadoop cluster in the centralized hybrid cloud, distance between the source data node and the target compute node can be known, and data block transfer time can be calculated.

(3) According to data block transfer time, the map task whose data block transfer time is less than the max permitting time T of data prefetching, its scheduling tuple (including task Id, source data node, target compute node and data block transfer time) is let insert to the scheduling list *schedulerList* for input data prefetching. After that, this map task is deleted from *prefetchMaps* and the target compute node is deleted from *candidateNodes*.

In Hadoop cluster, all slaver nodes send heartbeat periodically to the master node to report their status. Then, the master node assigns tasks to the slaver nodes which have free containers. That is, currently busy nodes will not have new tasks being assigned before next heartbeat arrives even if the containers have been released. Assume the heartbeat cycle of Hadoop cluster is *Heart*, thus, the maximum permit time T for data prefetching by using the idle bandwidth should satisfy the conditions listed in formula (4).

$$\begin{cases} T > \max\left(T_i^{perblock}\right), i \in [0, p) \\ T = n * Heart,\ n \in Z \end{cases} \tag{4}$$

From formula (4), in order to ensure the slowest node can acquire the data block in each round data prefetching, the maximum

Algorithm 2 Map tasks preselection.

Input: The failed map tasks *failedMaps*; unassigned map tasks *pendingMaps*; the candidate nodes *candidateNodes*
Output: The non-node locality map tasks *prefetchMaps*; the non-node locality candidate nodes *candidateNodes*

```
for each map task i in failedMaps/pendingMaps do
  if map task i has node locality relative to node j in candidateNodes then
    Delete map task i from failedMaps/pendingMaps
    Delete node j from candidateNodes
    if candidateNodes is null then
      Break // end this algorithm
    end if
  end if
end for
Add map tasks of failedMaps/pendingMaps to prefetchMaps
```

Algorithm 3 Input data prefetching.

Input: Preselected non-node locality map tasks $prefetchMaps = \{mapTask_i\}$;
Preselected candidate compute nodes $candidateNodes = \{Node_j\}$
Output: Data blocks to be fetched $I = \{dataBlock_b\}, b = 0..B$
1. **for each** $mapTask_i \in prefetchMaps$ **do**
2. $Node_t \leftarrow$ Search compute node from $candidateNodes$ for $mapTask_i$ according to the required resources
3. $Node_s \leftarrow$ Search the nearest source data node
4. $BW_{s,t} \leftarrow$ Get communication bandwidth between $Node_t$ and $Node_s$
5. $T_i^{perBlock} = S_{block}/BW_{s,t}$// One data block transfer time
6. **if** $T_i^{perBlock} < T$ && $num(dataBlock_i) < threshold(dataBlock)$ **then**
/* $num(dataBlock_i)$is the number of replicas of $dataBlock_i$; $threshold(dataBlock)$ is the max number of replicas of each data block allowed in HDFS */
7. Add $tuple(mapTask_i, Node_t, Node_s, T_i^{perBlock})$ to $scheduleList$
8. $num(dataBlock_i) = num(dataBlock_i) + 1$
9. Set $Node_t$ the flag of being assigned
10. **end if**
11. **end for**
12. Fetch input data for map tasks in $scheduleList$ from the source nodes to the target nodes
13. Update metadata information in ResourceManager
14. Update the data block information in HDFS

permit time T for data prefetching should satisfy the condition that $T > Max(T_i^{perBlock}), i \in [0, p]$, here, p is the number of the data source nodes of map task i. Since task assignment in Hadoop cluster proceeds in accordance with the heartbeat cycle, after a node released its container, the node can keep its idle bandwidth until the next heartbeat arrives. That is, the maximum permitting time of data prefetching should be an integer multiple of the heart beat cycle, i.e. $T = n * Heart, n \in Z$, here Z is the integer set.

(4) According to scheduling list *scheduleList*, data chunks of all map tasks are fetched from their data source nodes to their target compute nodes. Then, the metadata information of ResourceManager and data blocks metadata information in HDFS are updated.

The algorithm of scheduling-aware data prefetching for data locality in centralized hybrid cloud is a cyclic process. In each loop, a new round of compute nodes preselection, map task preselection and input data prefetching is performed to achieve the real-time and efficient data locality optimization. Algorithm 3 depicts the pseudo-code for input data prefetching.

3.2. File synchronizing for data locality in distributed hybrid cloud

In order to provide long-term and stable cloud services for users, combining with the advantages (stability and high resistance to disaster) of distributed hybrid cloud, considering job level task scheduling from cloud-layer, this paper proposed **a file synchronizing method for data locality in distributed hybrid cloud** (***DHCDLO-Sync***). This method aims to transfer popular data files of jobs to the sub-clouds of hybrid cloud beforehand, so that jobs can access input data from local HDFS when execution, avoiding the waiting delay caused by cross-cloud data transmission. This shame can be divided into two steps: sync file preselection and file synchronization among sub-clouds. Firstly, according to files usage in local HDFS and confined conditions set by administrator, hot files are selected. Secondly, through information interaction among resource scheduling servers of the sub-clouds in distributed hybrid cloud, hot files are synchronized periodically to each sub-cloud, such that prior to the jobs' assignment to the sub-clouds, most or all of their input data had been synchronized to the sub-clouds, which can reduce execution waiting delay of jobs, improve execution efficiency of the sub-cloud, and hence increase the efficiency of the whole distributed hybrid cloud.

3.2.1. Sync file preselection

In this step, synchronous files are selected to synchronize among sub-clouds. First, the access frequency of each file in local HDFS is analyzed. Second, by comparing file popularity with the predefined threshold, hot files are selected as the sync files.

Assume the total number of times that file access is $cnt_{i,\,acc}$, the statistical interval is $[T_{i,\,start}, T_{i,\,now}]$, then, the access frequency of the file is,

$$frq_{i,acc} = cnt_{i,acc}/(T_{i,now} - T_{i,start}) \tag{5}$$

Suppose the influence factor of the restrictive conditions (such as access authority, data security, read only, etc.) set by administrator is *filter*, according to the affect extent of the restrictive conditions on file popularity, the value of the influence factor *filter* should be adhere to,

$$filter = \begin{cases} 0 & Limit \\ 1 & unLimit \\ +\infty & specialLimit \end{cases} \tag{6}$$

where *Limit* indicates data files dissatisfy the restrictive conditions, *unLimit* means data files have no restrictive conditions, *specialLimit* expresses data files should satisfy special conditions, such as they must be synchronized to other sub-clouds unconditionally.

The popularity reflects the quantity and intensity being accessed of a file, which can be formulated as,

$$pop_i = filter * frq_{i,acc} = filter * ct_{i,acc}/(T_{i,now} - T_{i,start}),\quad filter \in \{0, 1, +\infty\} \tag{7}$$

In which, *filter* is the influence factor of the restrictive conditions; ct_{acc} indicates file i's access times; $T_{i,\,now}$ expresses the finish time of file access statistics; $T_{i,\,start}$ shows the start time of file access statistics. From formula (7) we can learn, the more popular the file, the greater the probability of the file becoming jobs' input data.

As popular files are most likely to be the input data of jobs, in order to improve sub-clouds' data locality and reduce jobs completion time, the most popular files in each sub-cloud can be prefetch to other sub-clouds in advance. However, data files prefetching may increase the burden of network, thus only the files whose popularity is higher than the sub-cloud's popularity threshold are allowed to be prefetched to other sub-clouds. The sub-cloud's popularity threshold pop_{thred} is formulated as,

$$pop_{thred} = 2 * \frac{1}{n}\sum_{i=1}^{n} pop_i \tag{8}$$

where, $\frac{1}{n}\sum_{i=1}^{n} pop_i$ is the average popularity of the files in the sub-cloud's local HDFS, n is the number of files in local HDFS.

Finally, sync files preselection is performed according to the following regulations: If a file's popularity is larger than sub-cloud's

Algorithm 4 Sync file preselection.

Input: Files in local HDFS of sub-cloud i, $f = \{f_1, f_2, ..., f_n\}$,
Output: The sync files $syncFiles = \{f'_1, f'_2, ..., f'_m\}$
1. **for each** $f_i \in f$ **do**
2. $frq_{i,acc} = cnt_{i,acc}/(T_{i,now} - T_{i,start})$// File access frequency
3. $filter = \begin{cases} 0 & Limit \\ 1 & unLimit \\ +\infty & specialLimit \end{cases}$ // Restrictive conditions influence factor
4. $pop_i = filter * frq_{i,acc} = filter * ct_{i,acc}/(T_{i,now} - T_{i,start})$ // File popularity
5. $pop_{thred} = 2 * \frac{1}{n} \sum_{i=1}^{n} pop_i$ // File popularity threshold
6. **if** $pop_i > pop_{thred}$ **then**
7. Add f_i to $syncFiles$ // Save file f_i to $syncFiles$
8. **end if**
9. **end for**

popularity threshold, i.e.$pop_i > pop_{thred}$, the file will be chosen as the sync file and appended to the sync file set *syncFiles*; else this file will not be added to the sync file list.

Algorithm 4 depicts the pseudo-code for sync file preselection in each sub-cloud. Firstly, for each file in sub-cloud i, the popularity is calculated (Algorithm 4 lines 1∼4). Then, the popularity threshold of sub-cloud i is calculated (Algorithm 4 line5). Finally, hot files are selected as the sync files to synchronize to other sub-clouds (Algorithm 4 lines 6∼8).

3.2.2. File synchronization among sub-clouds

Files synchronization among sub-clouds is the process of copying popular data files of local HDFS to other sub-clouds in periodically during the communication of the resource scheduling servers of sub-clouds in distributed hybrid cloud. Files synchronization follows the steps of ① planning sub-clouds' synchronizing order according to their workloads and ② synchronizing popular files to other sub-clouds.

(1) Sub-clouds' synchronization planning

Before popular files' synchronization, the workload of all sub-clouds in the distributed hybrid cloud should be calculated according to formula (9). In the process of file synchronization, light loaded sub-clouds are synchronized first.

$$L_i = w_1 U_i^{cpu} + w_2 U_i^{mem} \tag{9}$$

In formula (9), L_i is the workload of sub-cloud i; U_i^{cpu} represents the average CPU usage of sub-cloud i; U_i^{mem} indicates the average memory usage of sub-cloud i; w_1 is the weight of CPU usage and w_2 is the weight of memory usage in the sub-cloud.

Each sub-cloud has a certain workload limit, if the workload of a sub-cloud exceeds its maximum workload, i.e. $L_i > L_{\max}$, then this sub-cloud is considered in its full capability, and cannot afford computing services for user requests any more. Hot files are not allowed to be synchronized to this sub-cloud and this sub-cloud will be deleted from sub-clouds sync list *syncList*. In order to let the light loaded sub-clouds be synchronized in priority, sub-clouds in *syncList* should be sorted on ascending.

(2) Data file synchronization among sub-clouds

For each sub-cloud in *syncList*, whether all popular files in file sync set *syncFiles* exist in local HDFS is checked. If not, the absent files will be synchronized to this sub-cloud, so that jobs can access them locally. Moreover, in order to avoid unnecessary data transmission and reduce the overhead caused by cross-cloud data transmission, file replication factor of each sub-cloud is set to 1, that is, only one copy of each file is allowed to be synchronized remotely to this sub-cloud.

Algorithm 5 depicts the pseudo-code of file synchronization among sub-clouds, which is shown as follows. Planning participated sync sub-clouds and sorting their synchronizing order according to workload is carried out in Algorithm 5 lines 1∼7. Synchronizing popular files among sub-clouds is implemented in Algorithm 5 lines 8∼16.

The process of remote file transmission can be realized via FTP programming interface provided by HDFS of Hadoop, which is essentially the process of cross-cloud data transmission on Internet. Different from the centralized hybrid cloud which transfers data blocks by sockets and needs to update/modify the metadata information about data blocks in NameNode of HDFS to keep the prefetched input data stay in the compute node after their use, the data locality optimization in distributed hybrid cloud is implemented through FTP transmission by using remote procedure call interface (RPC) provided by Hadoop, while does not involve the modification of MapReduce framework and HDFS metadata.

4. Application scenarios: online cloud gaming industry

In order to illustrate the validity of our algorithms for real life workloads, we design the industry motivating scenario. With the development of cloud technology, as an emerging paradigm of online entertainment industry, cloud gaming, which allows users to play high quality video games instantly without downloading or installing the original game software, has attracted significant attention (Wu et al., 2014; Cai and Leung, 2012). Similar to many cloud computing services, cloud gaming service has serval advantages in contrary to conventional video game business, such as terminal hardware constraints overcoming, salability, cost effectiveness, cross-platform support, effective antipiracy solution, and

Algorithm 5 File synchronization among sub-clouds.

Input: The resource scheduling server address list in hybrid cloud $syncList = \{addr_1, addr_2, ..., addr_n\}$; The sync files $syncFiles = \{f_1, f_2, ..., f_n\}$;
Output: The copies of the files that synchronized to different sub-clouds
1. **for each** $addr_i \in syncList$ **do**
2. $L_i \leftarrow$ Compute sub-cloud workload
3. **if** $L_i > L_{\max}$ **then** //if workload larger than the threshold, the sub-cloud is not allowed synchronizing
4. delete $addr_i$ from $syncList$
5. **end if**
6. **end for**
7. Sort $syncList$ on ASC according to sub-clouds' workload
8. **for each** $addr_j \in syncList$ **do**
9. Connect to the *j-th* sub-cloud
10. **for each** $f_i \in syncFiles$ **do**
11. **if** f_i is not in the HDFS of the *j-th* sub-cloud **then**
12. Synchronize f_i to the HDFS of the *j-th* sub-cloud
13. **end if**
14. **end for**
15. Disconnect the *j-th* sub-cloud
16. **end for**

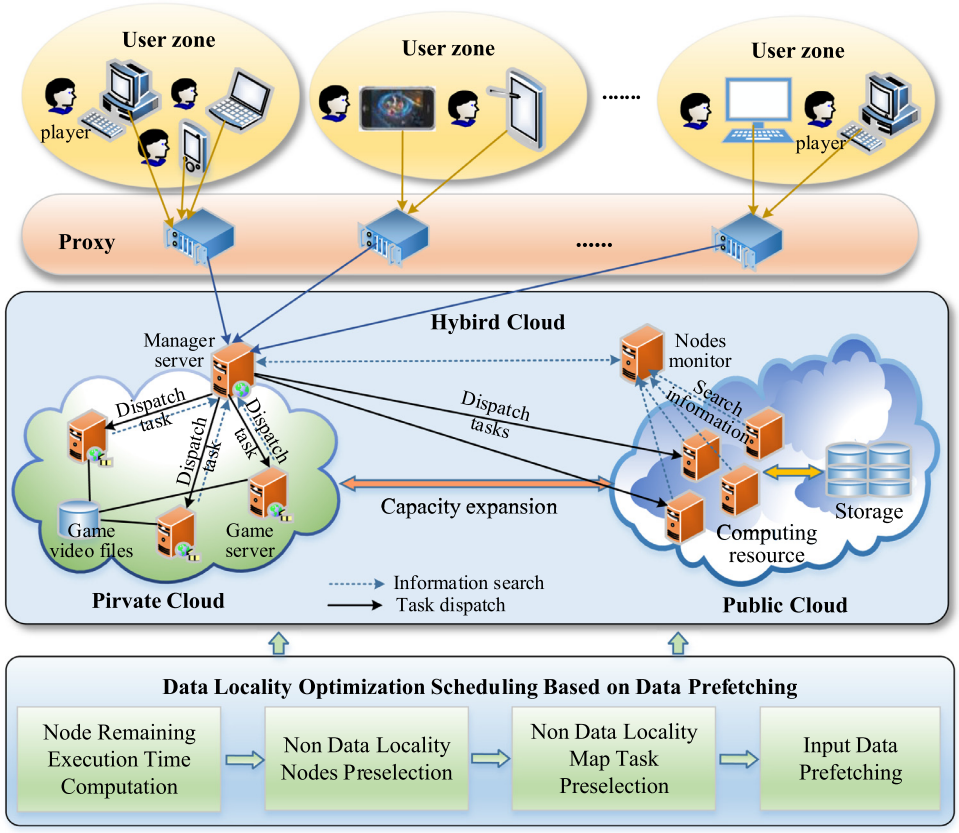


Fig. 3. Our proposed method in cloud gaming scenario.

so on (Cai et al., 2014). However, massive video rendering and encoding put forward higher requirements for computation and storage performance. Moreover, high-quality and high-frame-rate video transmissions often consume huge amount of bandwidth (Cai et al., 2014). In case of cloud bursting occasions, such as at weekend, in holidays, or on occasion of new game release, on-premise resource provision for game playing cannot afford so many players' requests, and extra resources are needed. However, due to the high initial capital expenditure and re-occurring operating expenditure, it is costly for gaming providers to procure all infrastructures to afford for cloud bursting. Therefore, hybrid cloud which uses both private and public resources becomes an alternative choice (Lu et al., 2015). Depending on game process demand, gaming providers can operate the private resources of their own, or rent resources from public clouds. Here, we take the centralized hybrid cloud as the example to illustrate the scenario of our scheduling-aware data prefetching for data locality algorithm applying in online cloud gaming, which is as Fig. 3.

As shown in Fig. 3, the structure of the online cloud gaming is mainly consists of two parts: client side and hybrid cloud gaming center. The hybrid cloud gaming provider provides game services to players located in geo-distributed zones. In client side, each zone has a service proxy, and the players' game requests are first sent to the proxies in their zones and then routed to the hybrid cloud for further processing. The hybrid cloud gaming center is composed of private cloud and public cloud. In which, all game nodes are managed by the manager server which acts as hybrid cloud master scheduling server. The manager server can be either a powerful machine or a server cluster, which in charges of requests receiving, task dispatching, resource allocation, information search, capacity expansion decision making, etc. The game nodes/servers are mainly responsible for initializing the execution environment, running game tasks, transmitting game video streaming and reporting the running state of its own. In order to manage the hybrid cloud well, there usually a monitor node set in the public cloud to monitor the state of running game servers. The monitor node communicates with the manager server periodically, and reports the state of these game servers. The algorithm of our scheduling-aware data prefetching is running on the manager server.

In hybrid cloud gaming scenario, the processing workloads are game videos, the jobs are regarded as online gaming applications, which usually associate with a sequence of video processing tasks such as rendering, capturing, encoding, transmitting, etc. Essentially, a cloud game is a software program written in some programming language. Regardless of object- or procedure-oriented design, we consider gaming as a loop procedure that enables interaction between players and game logic.

In this section, we take rendering task as the instance to illustrate the process of how our approach applying in this scenario. Rendering tasks are usually processed in parallel. For node locality rendering tasks, they are scheduled directly to corresponding nodes which have stored their game video chunks. As for non-node locality rendering tasks, in order to reduce the waiting time before tasks' execution, game video chunks are prefetched to compute nodes in advance, which can reduce rendering tasks' data access waiting delay significantly. It is in this manner, data locality is exploited. The process of our scheduling-aware data prefetching strategy for data locality applying in hybrid cloud gaming scenario is described as follows.

(1) Resource allocation. In centralized hybrid cloud, the algorithm of scheduling-aware data prefetching is running on the master scheduling server. Upon receiving a connection request from a player, if there are available on-premise resources, the master scheduling server will launch a dedicated server (either a physical machine or a virtual

Table 1
Public clouds server instances configuration.

Provider	Instance
Microsoft	Node 1∼10: CPU 2.6 Ghz Core: 1, Memory: 1 GB, Disk: 50 GB
Amazon	Node 1∼10: CPU 2.4 Ghz Core: 1, Memory: 1 GB, Disk: 80 GB
Alibaba	Node 1∼10: CPU 2.5 Ghz Core: 1, Memory: 1 GB, Disk: 70 GB

Table 2
Private cloud hardware configuration.

Host	Hardware configuration
Node 1	CPU: Intel Xeon E3, Memory: 4 GB, Disk: 1 TB, Number of NIC: 4
Node 2	CPU: Intel Xeon E3, Memory: 16 GB, Disk: 1 TB, Number of NIC: 4
Node 3	CPU: Intel Xeon E5, Memory: 2 GB, Disk: 500 GB, Number of NIC: 4

Table 3
Private cloud VM instances configuration.

Host	VM configuration
Master	CPU 3.4 Ghz Core: 4, Memory: 4 GB, Disk: 500 GB
Node 1∼10	CPU 2.5 Ghz Core: 1, Memory: 2 GB, Disk: 80 GB

machine) to run the game and stream gaming video to user client.

(2) Capacity expansion. When the capacity of the private cloud cannot meet the demand, user's requests will be held in a rendering queue, extra resources from public cloud are rented and virtual machines (VMs) are created dynamically to guarantee the quality-of-service (QoS) of game service.

(3) Data locality task scheduling. In Hadoop-based hybrid cloud, games are rendered in parallel, and processed by multiple mappers. In each round rendering task scheduling, the master scheduling server checks the state of each node in the cluster, estimates their remaining execution time, and finds those nodes that are likely to release their compute resources. In next scheduling round, the master scheduling server will assign the rendering tasks to the nodes who have stored their gaming video files.

(4) Non data locality tasks data prefetching. As for non-node locality rendering tasks, the nodes whose resource releasing time is a bit longer than the time of prefetching a game video chunk from the nearest replica node are selected as the candidate game rendering servers. The game video chunks of those non node locality rendering tasks are prefetched to corresponding game rendering servers according to the resource matching results.

5. Experiments

5.1. Experiment environment

To construct hybrid cloud, in this experiment, 10 Aliyun server instances (Aliyun ECS), 10 Microsoft Azure server instances (Microsoft Azure) and 10 Amazon EC2 server instances (Amazon EC2) are rented as the public clouds; three Inspur servers are used to structure the private cloud. Based on them an OpenStack system (Mirantis OpenStack) is deployed to virtualize more nodes for establishing Hadoop cluster. Upon on OpenStack, 11 virtual machines (VMs) are virtualized. We deploy Hadoop-2.7.1 over hybrid cloud as the cloud platform. We use VMware Workstation 11.1.2 as the virtualization solution. On each node, Ubuntu-14.04.1 is deployed as the base operating system and JDK-7u79-linux-x64 is installed as the Java development kit. The integrated development environment is MyEclipse equipped with Hadoop plug-in hadoop-eclipse-plugin-2.7.1. A 1000 Mbps router is adopted to connect the private cloud with the public clouds. The configuration of public cloud server instances is shown as Table 1; the configuration of 3 Inspur servers is shown as Table 2, upon which 11 virtualized VMs are shown as Table 3.

5.2. Evaluation metrics

(1) Data locality percentage

In our experiment, the data locality percentage of map tasks is adopted as the metric to measure the locality level of scheduling algorithms. Suppose the number of data locality map tasks is $mapCnt_{node}$, and the total number of map tasks in the system is $mapCnt_{all}$, then the data locality percentage of map tasks can be formulated as,

$$locality_{node} = mapCnt_{node}/mapCnt_{all} \tag{10}$$

(2) Job completion time

In this paper, we define the job completion time cpt_{job} as the sum of job waiting time wtt_{job} and its actual execution time ect_{job}, i.e. the turnaround time between when a job is submitted and when the job is completed (Saraswathi et al., 2015). Job completion time represents how fast a user receives the response from the system after a job is submitted, which can be formulated as,

$$cpt_{job} = wtt_{job} + ect_{job} \tag{11}$$

5.3. Hybrid cloud construction

5.3.1. Centralized hybrid cloud establishment

In centralized hybrid cloud, all servers are configured in VLAN, an Intel i5 desktop computer works as the VPN server. Hadoop configuration file of centralized hybrid cloud is set up in accordance with the Hadoop configuration process of LAN. We use VLAN IPs to configure Hadoop host address mapping table, i.e. the hosts file of configuration system, to make the communication of the nodes in the cluster being proceeded over VLAN, so that avoiding the limits of networks during communication. Hadoop hosts file is configured as Table 4.

Table 4
Hadoop hosts file configuration.

Host name	local host	master	slave1	slave2	slave3
VLAN IP address	127.0.0.1	192.168.5.11	192.168.5.21	192.168.5.22	192.168.5.23
Host name	vpn server	slaveX1	slaveX2	slaveX3	slaveX4∼X6
VLAN IP address	192.168.5.5	192.168.5.101	192.168.5.102	192.168.5.103	192.168.5.104∼106

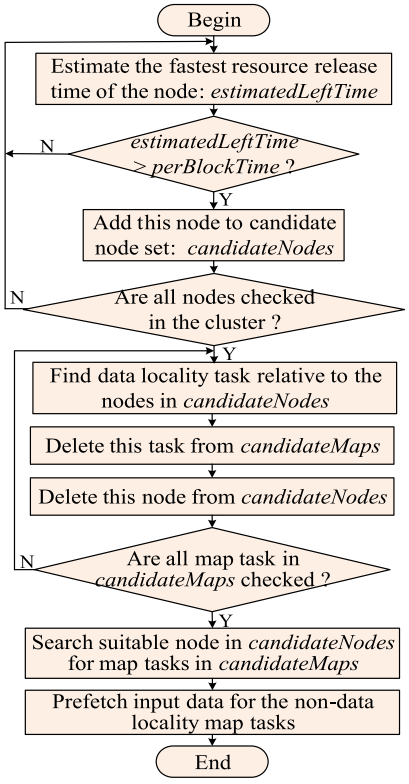


Fig. 4. The flow diagram of prefetching algorithm.

The VPN server works as the transit point for private cloud and public cloud communication, bounded with two network interface cards, one of which configures with a LAN IP to communicate with the private cloud, the other of which configures with a WAN IP to communicate with the public cloud. According to the content of hosts file, the VPN IPs address pool is configured. Through VPN client, Hadoop nodes are connected to VPN server. Thus, all of them can work in VLAN and have a fixed VLAN IP. As the gateway of VLAN, VPN server plays the role of routers and switches, routing and forwarding data for nodes communication in Hadoop-based hybrid cloud.

In centralized hybrid cloud, there is only one Hadoop platform deployed. Our data prefetching algorithm runs on the master scheduling server of the cluster, and interacts with Hadoop jobs. Jobs from client nodes are submitted to the master scheduling server. The nodes rented from public clouds are moved in and out of Hadoop cluster dynamically on demand. The flow diagram of the prefetching algorithm can be shown as Fig. 4. In each round job scheduling, the steps of the prefetching algorithm are as follows.

① Firstly, candidate nodes for prefetching are preselected. In Hadoop, all compute nodes regularly send heartbeat to the master scheduling server to report their running conditions. The master scheduling server assigns tasks to the compute nodes with idle containers. Therefore, the main basis for the selection of candidate compute nodes is the possibility of current release of busy containers on the compute node. Obviously, the nodes with the fastest task execution are most likely to release busy containers. In addition, only these nodes which can complete data prefetching before busy container release are selected as the candidate prefetching nodes. Thus, the nodes which are most likely to release busy containers and able to complete data prefetching before busy containers release are selected as the candidate compute nodes.

② Secondly, non-node locality tasks are preselected. It is known that, node locality map tasks need not data prefetching, we just consider the non-node locality map tasks for data prefetching. In each round scheduling, the un-running map tasks and the failed map tasks of currently running job are regarded as the candidate map tasks. For each candidate map task, if there is a candidate compute node had stored its input data, this map task has node locality relative to the candidate compute node. Then, this map task should be deleted from candidate map task set and the corresponding node should be deleted from the candidate compute node set. Thus, the tasks left in the candidate map task set are the non-node locality map tasks, and the nodes left in the candidate compute node set are the candidate target compute node for data prefetching.

③ Finally, according to the resource matching rules, the candidate compute nodes are traversed to find suitable resource for each non-node locality map task. After that, the scheduler lunches the prefetching process to prefetch part or all of map tasks' input data from their replica nodes to the compute nodes before these tasks are scheduled, so that data read waiting delay can be reduced during task execution, thus the response time of jobs can be decreased.

5.3.2. Distributed hybrid cloud establishment

Each sub-cloud of the distributed hybrid cloud is deployed an independent Hadoop platform, all sub-clouds status are equal. The master node which in charges of sub-cloud communication and job collaborative process is also serves as the sub-cloud's resource scheduling server. Different from the centralized hybrid cloud, information interaction among sub-clouds in distributed hybrid cloud is carried out through resource scheduling servers. The nodes in one sub-cloud cannot access the nodes that are not in the same sub-cloud directly, as their communication must be passed by the resource scheduling servers. Besides LAN IP, the master node of each sub-cloud is also configured a WAN IP, so as to interact with other sub-clouds' master nodes in the distributed hybrid cloud.

Our file synchronizing algorithm runs on each sub-cloud's master scheduling server (also the name node of HDFS). The master scheduling server interacts with other sub-clouds' master scheduling servers to process client jobs cooperatively, receives incoming jobs and the necessary input data, and feeds back the execution results up to the client during information interaction. In each sub-cloud, hot files are searched from local HDFS. Then, these hot files

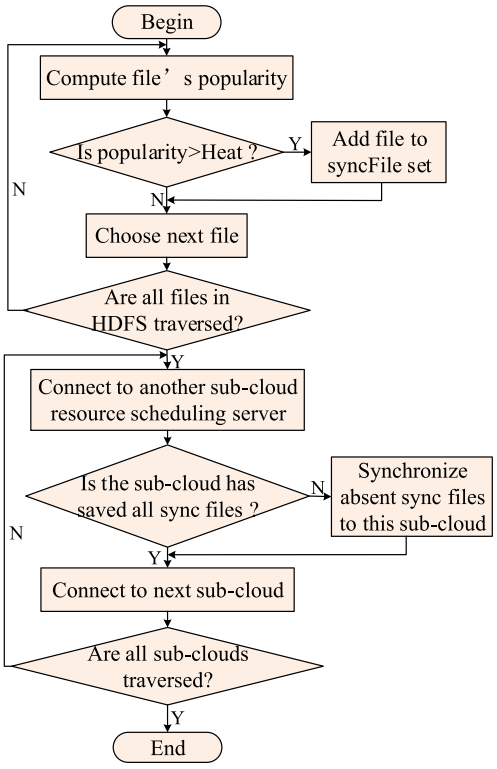


Fig. 5. The flow diagram of sync algorithm.

are transferred to other sub-clouds periodically during information interaction. The flow diagram of the sync algorithm can be shown as Fig. 5. In each round synchronizing, the process of the sync algorithm is implemented as:

① Sync files preselection. The popularity of each file in local sub-cloud HDFS is computed. The files whose popularities are larger than the threshold are selected as the sync files that to be synchronized to other sub-clouds.
② File synchronization. The master scheduling server of local sub-cloud connects with other sub-clouds master scheduling servers via querying the server address list. After connected to a sub-cloud, through information interaction the master scheduling server knows about if the connected sub-cloud had saved all of the sync files. If not, the master scheduling server will choose the absent sync files to transmit to the sub-cloud through FTP programming interface provided by HDFS.

According to formula (7) and (8), file heat and the threshold will be changed due to files synchronization before and after every test, therefore the experimental context should be restored to its initial status before each test. To obtain more files during *DHCDLO-Sync* algorithm's evaluation, parameter *filter* is set to 1, that is, there are no extra restrictions on file synchronizing filtering. In our experiment, data files in private cloud are used as the jobs' input data. The initial status of data files in private cloud is shown as Table 5, in which the number of replicas of each file is 1.

5.4. Experimental results analysis

In this section, Hadoop benchmark applications *Grep, TeraSort* and *WordCount* are adopted as the submit jobs, 10 log files (shown as Table 5) are employed as the input data. The applications of

Table 5
Initial status of files in private cloud.

File name	Size/Number of access times/Last access time
Testlog1	59.5 MB/ 190,823/ 2016–07–01
Testlog2	38.8 MB/ 110,097/ 2016–07–23
Testlog3	50.5 MB/ 122,301/ 2016–07–23
Testlog4	57.8 MB/ 110,651/ 2016–08–30
Testlog5	49.5 MB/ 133,403/ 2016–09–11
Testlog6	48.8 MB/ 34,532/ 2016–09–20
Testlog7	56.6 MB/ 48,792/ 2016–09–23
Testlog8	59.7 MB/ 56,842/ 2016–09–23
Testlog9	63.2 MB/ 31,668/ 2016–09–23
Testlog10	47.3 MB/ 52,487/ 2016–09–30

Grep, TeraSort and *WordCount* are typical MapReduce jobs for measuring performance of Hadoop. *Grep* is used for extracting the matching strings in large file; *Terasort* is adopted to sort the large collection of data; while *WordCount* counts how many times of words occur in the file. Among which, *Grep* and *TeraSort* are I/O intensive while *WordCount* is CPU intensive workload. By comparing our proposed algorithms with the *Capacity* (Capacity Scheduler), the *Fair* (Fair Scheduler), the *DARE* (Abad et al., 2011) algorithms, the performance of the scheduling-aware data prefetching algorithm and file synchronizing algorithm are evaluated in accordance with node locality percentage and job completion time.

5.4.1. Performance analysis under centralized hybrid cloud

(1) **Algorithm performance comparison under different job execution times**

Fig. 6 illustrates the performance effect of job execution times on our algorithm and the baseline algorithms. In this test, *WordCount* job is adopted as the submitted job. As shown as Fig. 6, with job execution, data locality percentage of the *CHCDLOS-Prefetch* algorithm and the *DARE* algorithm increase, while there almost no change in that of the *Capacity* and the *Fair* algorithms. That's because, the *DARE* and the *CHCDLOS-Prefetch* algorithms had considered data locality optimization in their scheduling. However, our proposed *CHCDLOS-Prefetch* algorithm outperforms the *DARE* algorithm significantly. As Fig. 6 indicates, after multi-round scheduling, the node locality percentage of our *CHCDLOS-Prefetch* algorithm is in proximity to 100%, that's to say, when job executes to a certain number of times, each node may be prefetched a replica of the input data, each map task may achieves node locality.

(2) **Performance comparison under different types of jobs**

In this part, we perform baseline algorithms and our *CHCDLOS-Prefetch* algorithm with different types of jobs, the *Grep*, the *TeraSort* and *WordCount*, and compare them in accordance with the performance metrics of node locality percentage and job completion time. The algorithms execute each job continuously for 3 times, the mean of 3 times execution is used as the measuring results of performance metrics. Figs. 7 and 8 show the average node locality percentage and average job completion time respectively of different scheduling algorithms.

As shown in Fig. 7(a), when execute the *Grep* job, our *CHCDLOS-Prefetch* algorithm has the highest average data locality percentage than the other baseline algorithms. Compared to the *Capacity*, the *Fair* and the *DARE* algorithms, the *CHCDLOS-Prefetch* algorithm can increase the average data locality by up to 21.6%, 17% and 7% respectively. The same performance trends are shown in

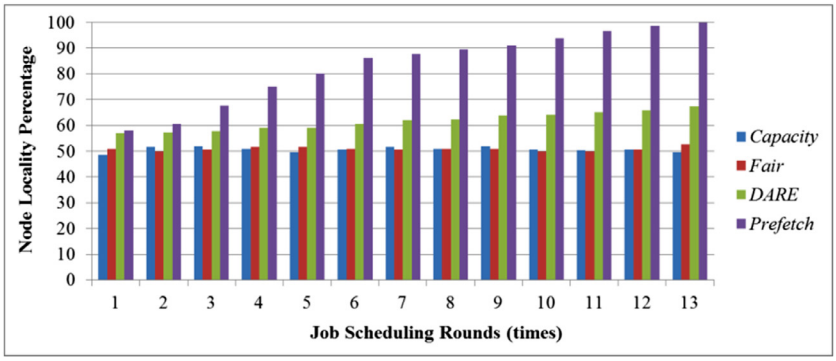
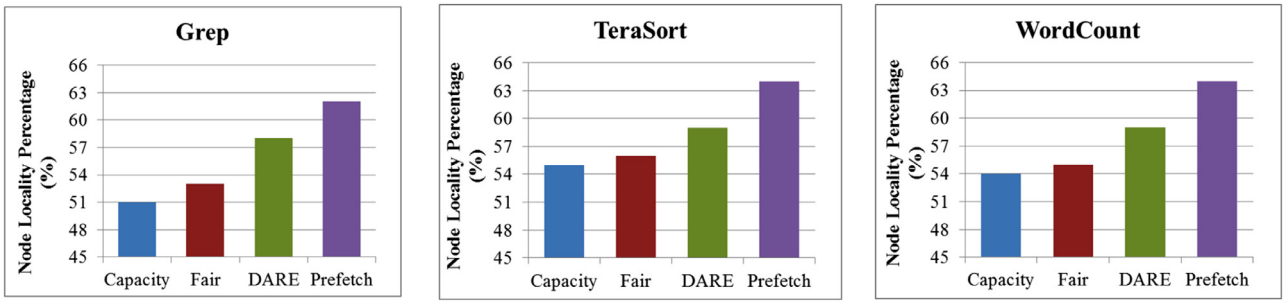
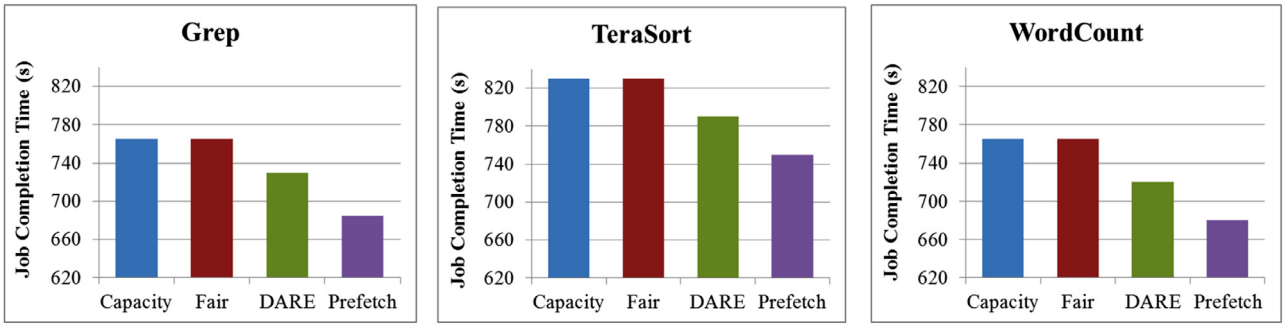


Fig. 6. Algorithm performance comparison under different job execution times.



(a) Node Locality Percentage of Grep (b) Node Locality Percentage of TeraSort (c) Node Locality Percentage of WordCount

Fig. 7. Performance comparison under different type of jobs.



(a) Job Completion Time of Grep (b) Job Completion Time of TeraSort (c) Job Completion Time of WordCount

Fig. 8. Performance comparison under different type of jobs.

Fig. 7(b) and (c) when process the *TeraSort* job and the *WordCount* job, however, our *CHCDLOS-Prefetch* algorithm always has the better data locality than the other benchmarks. That's because, in our *CHCDLOS-Prefetch* algorithm, we had considered prefetching input data to target compute nodes in advance and retaining the data in the compute nodes to enhance job's data locality. From Fig. 7(a), (b), (c) we can conclude that, our *CHCDLOS-Prefetch* method has the stable superiority in performance of data locality under different types of jobs.

As shown as Fig. 8, compare to the baseline algorithms, our *CHCDLOS-Prefetch* algorithm has the least average job completion time when process different types of jobs. As Fig. 8(a) shows, when process the *Grep* job, the *CHCDLOS-Prefetch* algorithm can reduce the job completion time by 12%, 11.7% and 6.6% respectively compare to the *Capacity*, the *Fair* and the *DARE* algorithms; when process the *TeroSort* job, shown in Fig. 8(b), the *CHCDLOS-Prefetch* algorithm can reduce job completion time by up to 10.9%, 10.6% and 5.3% respectively; and when process the *WordCount* job, illustrated as Fig. 8(c), the *CHCDLOS-Prefetch* algorithm can reduce job completion time by up to 12.5%, 12% and 5.9% respectively. Through performance comparison, the superiority of our *CHCDLOS-Prefetch* algorithm in average data locality and average job completion time under different types of jobs is verified.

(3) Comparison under different number of public cloud nodes

By resuming the experimental settings to its initial status, we measured the data locality percentage of these algorithms under different number of public cloud nodes. When *WordCount* job executed for three times, the average data locality percentage of map tasks changing with the number of public cloud nodes can be shown as Fig. 9. When the job executed for 5 times, the average

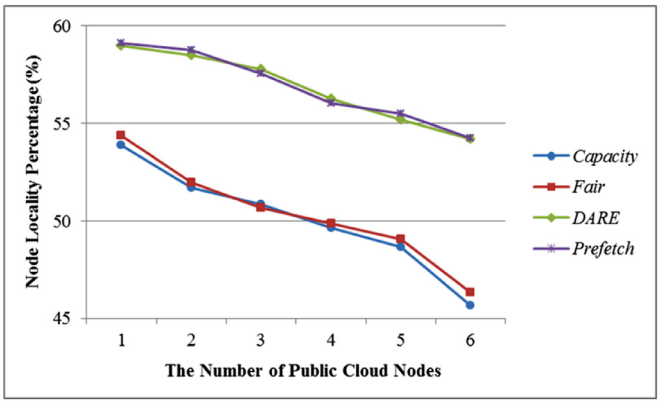


Fig. 9. Average node locality percentage of map tasks when job executed for 3 times.

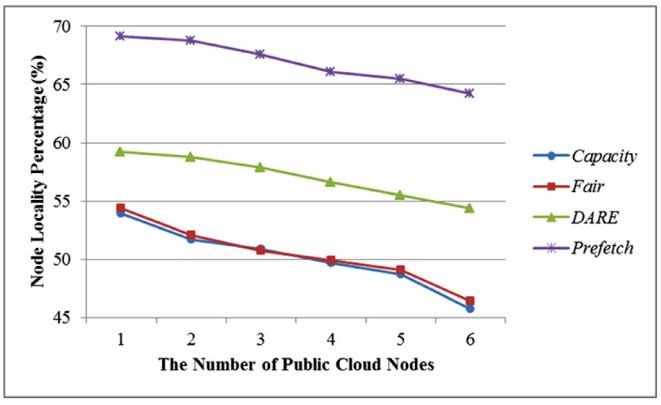


Fig. 10. Average node locality percentage of map tasks when job executed for 5 times.

data locality percentage of map tasks changing with the number of public cloud nodes is shown as Fig. 10.

From Fig. 9, as the number of public cloud nodes increases, the data locality percentage decreases. While, the data locality percentage of map tasks under the *CHCDLOS-Prefetch* algorithm and the *DARE* algorithm are higher than that of under the *Capacity* algorithm and the *Fair* algorithm. That's because, our *CHCDLOS-Prefetch* algorithm and the *DARE* algorithm have considered increasing input data replicas to aid the scheduler to achieve better data locality.

After the job executed for 5 times, as shown in Fig. 10, the average data locality percentage under our *CHCDLOS-Prefetch* algorithm is better than that of under the baseline algorithms. It indicates that the *CHCDLOS-Prefetch* algorithm has pre-synchronized more input data than that of the *DARE* algorithm at runtime, thus has better data locality.

5.4.2. Performance analysis under distributed hybrid cloud

(1) **Input data read percentage**

With the execution of *DHYDLO-Sync* algorithm, the percentage of input data read from local sub-cloud and remote sub-clouds are shown as the red histogram and blue histogram in Fig. 11 respectively. As shown in Fig. 13, with the execution of the algorithm, the input data read from local sub-cloud increases, whereas the data read from remote sub-clouds decreases. That's because, during *DHYDLO-Sync* running, hot files are constantly synchronized to the sub-clouds, which improves the sub-clouds' data locality.

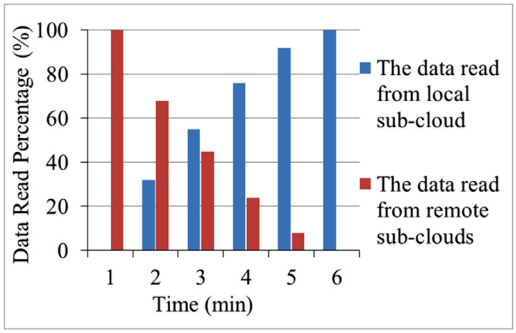


Fig. 11. The input data read percentage from local and remote sub-clouds.

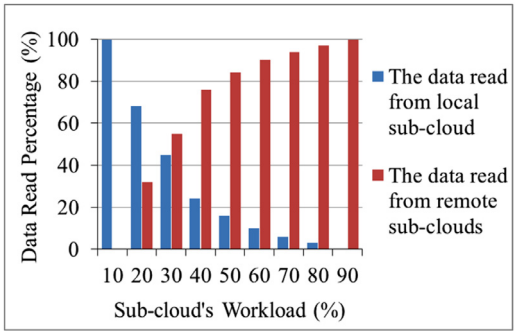


Fig. 12. The input data read percentage changes with the sub-cloud's workload.

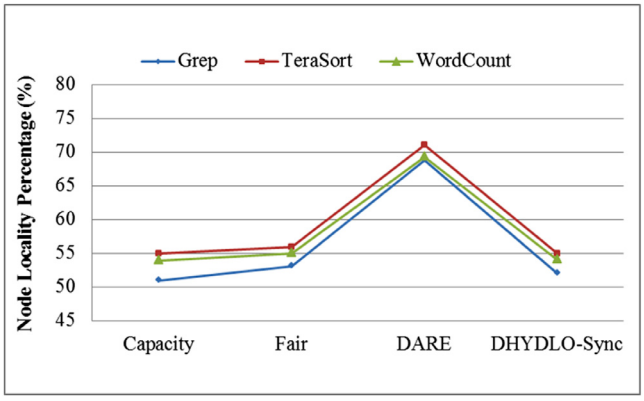


Fig. 13. Node localtiy percentage of map tasks.

The effect of sub-cloud workload on input data read percentage is shown as Fig. 12. With the increase of sub-cloud workloads, the percentage of the input data read from local sub-cloud decrease, while the input data read from remote storages increases. That's because, with the workload's inrease, there are few files can be synchronized to this sub-cloud, which reduces the data locality of the sub-cloud.

(2) **Node locality percentage**

Fig. 13 shows us the node locality percentage of map tasks under the baseline algorithms and our proposed *DHYDLO-Sync* algorithm when process the *Grep, TeraSort* and *WordCount* Jobs. As Fig. 13 shows, the *DARE* algorithm has the highest map task data locality. It is higher than our *DHYDLO-Sync* algorithm which has the similar data locality percentage with the *Capacity* and the *Fair* algorithms. That's because, the *DARE* algorithm is

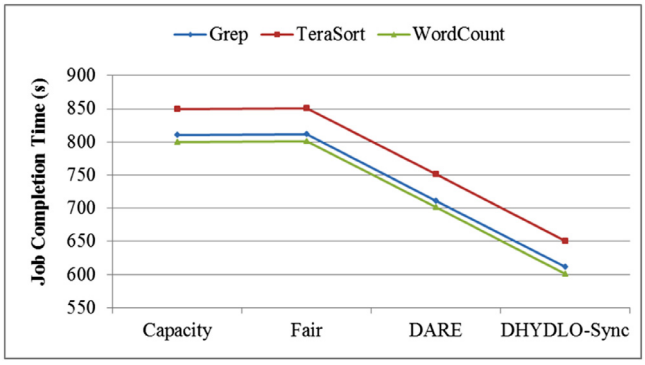


Fig. 14. Average completion time of jobs.

focus on the node locality optimization for map tasks, while our *DHYDLO-Sync* algorithm aims at data locality optimization from job level resolutions and pursues cloud-level data locality for jobs.

(3) **Jobs completion time**

We perform each job for three times, and take the average value as the job completion time. Fig. 14 shows the completion time of the *Grep, TeraSort* and *WordCount* jobs under the baseline algorithms and our *DHYDLO-Sync* algorithm. As shown as Fig. 14, our *DHYDLO-Sync* algorithm has the least job completion time compare to the baseline algorithms. During process the *Grep* job, the *DHYDLO-Sync* algorithm can save completion time by up to 32.5% and 32.7% respectively compare with the *Capacity* and the *Fair* algorithms, and about 16.4% time reduction compare with the *DARE* algorithm. When perform *WordCount* job, they have similar performance trends as performing the *Grep* job. Compare with the *Capacity*, the *Fair* and the *DARE* algorithms, our *DHYDLO-Sync* algorithm can save about 30.5%, 30.7% and 15.4% job completion time respectively when process the *TeraSort* job.

From Figs. 13 and 14 we can see, despite our *DHYDLO-Sync* algorithm has not higher node locality percentage, it has the optimum job completion time than the other baseline algorithms. That's because, in our *DHYDLO-Sync* algorithm, most popular files have been synchronized to the sub-clouds of the distributed hybrid cloud through file synchronization. When execute, jobs can read part or all of the input data from local sub-cloud's HDFS, which can reduce job execution waiting delay, thereby has the least average job completion time than other algorithms.

6. Conclusion

In this paper, to hide the latency caused by cross-cloud data transmission in remote file access, a scheduling-aware data prefetching scheme and a file synchronizing method for data locality in Hadoop-based hybrid cloud are proposed. In scheduling-aware data prefetching algorithm, to enhance centralized hybrid cloud data locality and decrease job completion time, input data for non-local map tasks are retrieved ahead of time by making use of idle network bandwidth. Firstly, the nodes which have idle bandwidth are selected as the candidate compute nodes. Secondly, map tasks which have non-node locality relative to the candidate compute nodes are selected as the non-local map tasks. Finally, input data of non-local map tasks are prefetched to target compute nodes in advance to reduce tasks execution waiting delay. In file synchronizing scheme, to decrease job execution waiting delay brought by cross-cloud data transmission in distributed hybrid cloud, considering from job level scheduling, data files with higher popularity are synchronized beforehand among sub-clouds to strengthen intra sub-cloud data locality. In order to avoid unnecessary data transmission, in each sub-cloud HDFS, the files only when their popularity is higher than the popularity threshold are selected as the sync files. Then, according to predetermined rules, the sync files from each sub-cloud are proactively sent to other sub-clouds' HDFS periodically. These two algorithms are evaluated in Hadoop-based centralized hybrid cloud and distributed hybrid cloud respectively. Extensive experimental results show that the scheduling-aware data prefetching algorithm outperforms the *Capacity*, the *Fair* and the *DARE* algorithms in data locality and job completion time. While for file synchronizing algorithm, despite has not so high node-locality percentage, it has the shortest average job completion time compared to the baseline algorithms.

In the future, we will design more accurate prediction schemes to speculate map tasks' remaining execution time. In our scheduling method, we consider data locality not only for map tasks but also for reduce tasks.

Acknowledgments

The work was supported by the National Natural Science Foundation (NSF) under grants (No. 61672397, No. 61873341, No. 61771354), Application Foundation Frontier Project of WuHan (No. 2018010401011290). Open Research Fund of Beijing Key Laboratory of Big Data Technology for Food Safety, Open Research Fund of Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the above agencies.

Appendix A. File synchronizing for data locality in distributed hybrid cloud

The source code of file synchronizing for data locality in distributed hybrid cloud is shown as following two java files. In Appendix A.1 we report on the algorithms for file heat statistics. In Appendix A.2 we report on the algorithms for file synchronization.

A.1. File heat statistics

```
package test;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
import util.MysqlConnection;
public class testcheck
{
 static String cutstring(String Stence) // Parsing audit logs, return file name
 {
 List<String> stringlist=new ArrayList<String>();//used to store the parsed elements
 for(int i=0;i<Stence.length();i++)
 {
  if(Stence.charAt(i)=='\t')
  {
   String temp=""; //Store words
   String s="";
   int wordlength=i;
   while(wordlength<Stence.length()-1&&Stence.charAt(++wordlength)!='\t')
   {
    temp+=Stence.charAt(wordlength);
   }
   s=temp.substring(0,3);
   if(s.equals("src"))
   {
    return temp.substring(4);
   }
  }
 }
 return null;
}

// Query whether the parsed elements has been recorded in the database
public static String search(String url,String ip,String port,String password) throws Exception
{
 String con_s="jdbc:mysql://"+ip+":"+port+"/d_map";
 Connection conn=null;  // Establish a connection
 Class.forName("com.mysql.jdbc.Driver");
 conn=DriverManager.getConnection(con_s,"root",password);
 System.out.println("Successfully  connected. ");
 String sql="select * from t_map where URL=?";
 PreparedStatement pstmt=conn.prepareStatement(sql);
 pstmt.setString(1,url);
 ResultSet rs=pstmt.executeQuery();
 String physical=null;
 while(rs.next())
 {
  physical=rs.getString(1);
  System.out.println(rs.getString(1)+"\n"+rs.getString(2)+"\n  ");
 }
 //long st1=System.currentTimeMillis();
 //System.out.println("Query  time:  "+(st1-st)+"ms");
 return physical;
}
// Statistics of the number of file access
static int filecheck()
{
 int index=0;
 BufferedReader br;
 try
 {
 br=new BufferedReader( new FileReader("/usr/hadoop/hadoop-2.7.1/logs/hdfs-audit.log"));
 StringBuffer sb = new StringBuffer();
 String x="";
 String line;
 while ((line = br.readLine()) != null) {
  // Parsing the rows of the log file with the characters of "open"
 if(line.indexOf("open", index)!=-1)
  {
   x=cutstring(line); //Parse which file is accessed
 //Query whether there are records inserted, if not then update new records into database
   if(search(x,"10.138.126.229","3306","root")==null)
   {
   String con_s="jdbc:mysql://10.138.126.229:3306/d_map";
   Connection conn=null;  // Establish the connection
   Class.forName("com.mysql.jdbc.Driver");
   conn=DriverManager.getConnection(con_s,"root","root");
   System.out.println("Successfully  connected. ");
   String sql="insert into t_map values (?,?,?,?)";
   try
   {
    PreparedStatement pstmt=conn.prepareStatement(sql);
    pstmt.setObject(1,x);
    pstmt.setObject(2,x);
    pstmt.setObject(3,1);
    pstmt.setObject(4,1);
    pstmt.executeUpdate();
    System.out.println("insert  success!");
   }
   catch(Exception e)
   {
    e.printStackTrace();
    System.out.println("insert  default");
    return -1;
   }
  }
  else
  {
   String con_s="jdbc:mysql://10.138.126.229:3306/d_map";
   Connection conn=null;  // Establish the connection
   Class.forName("com.mysql.jdbc.Driver");

    conn=DriverManager.getConnection(con_s,"root","root");
    System.out.println("Successfully  connected ");
    String sql="update t_map set touchtime = touchtime +1 where URL = ?";
    try
    {
        PreparedStatement pstmt=conn.prepareStatement(sql);
        pstmt.setObject(1,x);
     pstmt.executeUpdate();
     System.out.println("Update success!");
    }
    catch(Exception e)
    {
     e.printStackTrace();
     System.out.println("Update default");
     return -1;
    }
   }
  }
  br.close();
 }
 catch(Exception e)
 {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
 return 0;
}

public static void main(String[] args)
{
  filecheck(); //statistics of file access times
}
}
```

A.2. File synchronization algorithm

```
1.  package test;
2.  import java.io.InputStream;
3.  import java.io.OutputStream;
4.  import java.sql.Connection;
5.  import java.sql.DriverManager;
6.  import java.sql.PreparedStatement;
7.  import java.sql.ResultSet;
8.  import java.util.ArrayList;
9.  import java.util.List;
10. import org.apache.log4j.Logger;
11. import hdfs.ftp.client.HdfsFtpClient;
12. import hdfs.ftp.sync.SyncPolicy;
13. import hdfs.ftp.sync.SyncThread;
14. import hdfs.ftp.util.PathGetter;
15. import hdfs.ftp.util.PropertiesLoader;
16.
17. public class filesync
18. {
19.    static List<String> excludePaths=new ArrayList<String>();
20.    static HdfsFtpClient client=new HdfsFtpClient();
21.    static SyncPolicy syncPolicy=new SyncPolicy();
22.
23.    static  boolean isInExclude(String filePath)
24.    {
25.       for(String path:excludePaths)
26.       {
27.          if(filePath.startsWith(path))
28.             return true;
29.       }
30.       return false;
31.    }
32.    static  void transmitFileFromLocalHdfsToRemoteHdfs(List<String> filePaths)
33.    {
34.       client.setUsername("root");
35.       client.setPassword("root");
36.       client.setHost("10.141.68.126");
37.       System.out.println("Synchronize start, the target address is:"+client.getHost());
38.       if(client.connect())
39.       {
40.          for(String filePath:filePaths)
41.          {
42.             if(client.isExistedOnRemoteHdfs(filePath))
43.             {
44.                System.out.println("Remote HDFS already exists the file:"+filePath);
45.                continue;
46.             }
47.             else if(!client.isExistedOnLocalHdfs(filePath))
48.             {
49.                System.out.println("Local HDFS does not exist the file:"+filePath);
50.                continue;
51.             }
52.             else
53.             {
54.                InputStream is=client.getInputStreamFromLocalHdfs(filePath);
55.                if(is==null)
56.                {
57.                   System.out.println("Cannot get the input stream of the file in local HDFS. ");
58.                   continue;
59.                }
60.                OutputStream os=
61.                      client.getOutputStreamFromRemoteHdfs(filePath);
62.                if(os==null)
63.                {
64.                   System.out.println("Cannot get the output stream of the file from remote HDFS ");
65.                   try
66.                   {
67.                      is.close();
68.                   //  os.close();
69.                   }
70.                   catch(Exception e)
71.                   {
72.                      System.out.println("exception occurs when close local HDFS file's input stream:"+e);
73.                   }
74.                   continue;
75.                }
76.                if(client.copyStream(is,os))
77.                   System.out.println(" Synchronize successful:"+filePath);
78.                else
79.                   System.out.println(" Synchronize failed:"+filePath);
80.                if(!client.reconnect())
81.                {
82.                   System.out.println("Connection interrupted:"+client.getHost());
83.                   break;
84.                }
85.             }
86.          }
87.       }
88.       if(client.isConnected())
89.          client.disconnect();
90.    }
91.
92.    public static void main(String[] args)
93.    {
94.       HdfsFtpClient client=new HdfsFtpClient();
95.       SyncPolicy syncPolicy=new SyncPolicy();
96.       List<String> filePaths=new ArrayList<String>();
97.       String homeDir=null;// Synchronize the root directory
98.       //Check file
99.       String test1="/test/file01";
100.      String con_s="jdbc:mysql://10.138.126.229:3306/d_map";
101.      // Establish a connection
102.      Connection conn=null;
103.      try
104.      {
105.         Class.forName("com.mysql.jdbc.Driver");
106.         conn=DriverManager.getConnection(con_s,"root","root");
107.         System.out.println("Connection success ");
108.         // Get the objects that sql to query
109.         // Obtain files' access times
110.         String sql="select touchtime from t_map where URL=?";
111.         PreparedStatement pstmt=conn.prepareStatement(sql);
112.         pstmt.setString(1,test1);
113.         ResultSet rs=pstmt.executeQuery();
114.         int heat=0;
115.         while(rs.next())
116.         {
117.            heat=rs.getInt(1);
118.         }
119.         // Get file heat threshold (double of the average file heat of local HDFS)
120.         String sql1="select avg(touchtime)*2 from t_map ";
121.         PreparedStatement pstmt1=conn.prepareStatement(sql1);
122.         ResultSet rs1=pstmt1.executeQuery();
123.         int heatThreshold=0;
124.         while(rs1.next())
125.         {
126.            heatThreshold=rs1.getInt(1);
127.         }
128.         //Judge whether is the hot file,  if yes then added to the synchronize list
129.         if(heat>heatThreshold)
130.         {
131.            filePaths.add(test1);
132.         }
133.      }
134.      catch(Exception e)
135.      {
136.         // TODO Auto-generated catch block
137.         e.printStackTrace();
138.      }
139.      // Synchronize to remote sub-cloud's HDFS
140.      transmitFileFromLocalHdfsToRemoteHdfs(filePaths);
141.      // After synchronization, empty the table of t_map
142.      //  delete t_map for next cycle statistics
143.      try
144.      {
145.         Class.forName("com.mysql.jdbc.Driver");
146.
147.         conn=DriverManager.getConnection(con_s,"root","root");
148.         System.out.println("Connection success.");
149.         String sql2="delete  from t_map where 1=1";
150.         PreparedStatement pstmt2=conn.prepareStatement(sql2);
151.         ResultSet rs2=pstmt2.executeQuery();
152.      }
153.      catch(Exception e)
154.      {
155.         // TODO Auto-generated catch block
156.         e.printStackTrace();
157.      }
158.   }
159. }
```

References

Abad, C.L., Lu, Y., Campbell, R.H., 2011. DARE: adaptive data replication for efficient cluster scheduling. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp. 159–168.

Aliyun ECS, https://www.alibabacloud.com/zh/product/ecs.

Amazon EC2, http://aws.amazon.com/ec2/.

Ananthanarayanan, G., Agarwal, S., Kandula, S., et al., 2011. Scarlett: coping with skewed content popularity in mapreduce clusters. In: Proceedings of the sixth conference on Computer systems. ACM, pp. 287–300.

Cai, W., Chen, M., Leung, V.C.M., 2014. Toward gaming as a service. IEEE Internet Comput. 18 (3), 12–18.

Cai, W., Leung, V.C.M., 2012. Multiplayer cloud gaming system with cooperative video sharing. In: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp. 640–645.

Capacity Scheduler. https://issues.apache.org/jira/browse/HADOOP-3445.

Chen, W., Paik, I., Li, Z., 2016. Tology-aware optimal data placement algorithm for network traffic optimization. IEEE Trans. Comput 65 (8), 2603–2617.

Chen, W., Paik, I., Li, Z., et al., 2017. A cost minimization data allocation algorithm for dynamic datacenter resizing. J. Parallel Distrib. Comput.

Clemente-Castelló, F.J., Mayo, R., Fernández, J.C., 2017. Cost model and analysis of iterative mapreduce applications for hybrid cloud bursting. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster. Cloud and Grid Computing. IEEE Press, pp. 858–864.

Di Costanzo, A., De Assuncao, M.D., Buyya, R., 2009. Harnessing cloud technologies for a virtualized distributed computing infrastructure. IEEE Internet Comput. 13 (5).

Fair Scheduler. https://issues.apache.org/jira/browse/HADOOP-3746.

Guo, Y., Zhao, J., Cave, V., et al., 2010. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE, pp. 1–12.

Isard, M., Prabhakaran, V., Currey, J., et al., 2009. Quincy: fair scheduling for distributed computing clusters. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, pp. 261–276.

Javadi, B., Abawajy, J., Buyya, R., 2012. Failure-aware resource provisioning for hybrid cloud infrastructure. J. Parallel Distrib. Comput. 72 (10), 1318–1331.

Jin, J., Luo, J., Song, A., et al., 2011. Bar: an efficient data locality driven task scheduling algorithm for cloud computing. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). IEEE, pp. 295–304.

Kavulya, S., Tan, J., Gandhi, R., et al., 2010. An analysis of traces from a production mapreduce cluster. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid). IEEE, pp. 94–103.

Kovachev, D., Cao, Y., Klamma, R., 2014. Building mobile multimedia services: a hybrid cloud computing approach. Multimed. Tool. Appl. 70 (2), 977–1005.

Liao, J., Trahay, F., Xiao, G., et al., 2017. Performing initiative data prefetching in distributed file systems for cloud computing. IEEE Trans. Cloud Comp. (3) 550–562.

Li, C., Tang, J., Hengliang, T., Luo, Y., 2019. Collaborative Cache Allocation and Task Scheduling for Data-Intensive Applications in Edge Computing. Future Gener. Comput. Syst. 95, 249–264.

Li, C., Zhang, J., Tao, M., Tang, H., Lei, Z., Luo, Y., 2019. Data Locality Optimization Based on Data Migration and Hotspots Prediction in Geo-Distributed Cloud Environment. Knowl. Based Syst. 165, 321–334.

Lu, P., et al., 2015a. Distributed online hybrid cloud management for profit-driven multimedia cloud computing. IEEE Trans. Multimed. 17 (8), 1297–1308.

Lu, P., Sun, Q., Wu, K., et al., 2015b. Distributed online hybrid cloud management for profit-driven multimedia cloud computing. IEEE Trans. Multimed. 17 (8), 1297–1308.

Malawski, M., Figiela, K., Nabrzyski, J., 2013. Cost minimization for computational applications on hybrid cloud infrastructures. Future Gen. Comp. Syst. 29 (7), 1786–1794.

Mansouri, N., Javidi, M.M., 2018. A new prefetching-aware data replication to decrease access latency in cloud environment. J. Syst. Softw.

Marshall, P., Keahey, K., Freeman, T., 2010. Elastic site: using clouds to elastically extend site resources. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid). IEEE, pp. 43–52.

Microsoft Azure, http://msdn.microsoft.com/windowsazure.

Mirantis OpenStack, https://launchpad.net/mos/5.1.x.

Palanisamy, B., Singh, A., Liu, L., et al., 2011. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In: Proceedings of 2011 International Conference for High Performance Computing, Networking. Storage and Analysis. ACM, p. 58.

Saraswathi, A.T., Kalaashri, Y.R.A., Padmavathi, S., 2015. Dynamic resource allocation scheme in cloud computing. Proced. Comp. Sci. 47, 30–36.

Seo, S., Jang, I., Woo, K., et al., 2009. HPMR: prefetching and pre-shuffling in shared MapReduce computation environment. IEEE Int. Conf. Clust. Comp. Workshop. IEEE 1–8.

Tang, S., Lee, B.S., He, B., 2016. Fair resource allocation for data-intensive computing in the cloud. IEEE Trans. Serv. Comput. 1–14.

Van den Bossche, R., Vanmechelen, K., Broeckhove, J., 2013. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. Fut. Gen. Comp. Syst. 29 (4), 973–985.

Wang, W., Ying, L., 2016. Data locality in MapReduce: a network perspective. Perform. Evaluat. 96, 1–11.

Wang, W., Zhu, K., Ying, L., et al., 2016. Map task scheduling in mapreduce with data locality: throughput and heavy-traffic optimality. IEEE/ACM Trans. Netw. 24 (1), 190–203.

Wang, W.J., Chang, Y.S., Lo, W.T., et al., 2013. Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments. J. Supercomp. 66 (2), 783–811.

Wu, D., Xue, Z., He, J., 2014. iCloudAccess: cost-effective streaming of video games from the cloud with low latency. IEEE Trans. Circuits Syst. Video Technol. 24 (8), 1405–1416.

Zaharia, M., Borthakur, D., Sen Sarma, J., et al., 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European conference on Computer systems. ACM, pp. 265–278.

Zhang, H., et al., 2014. Proactive workload management in hybrid cloud computing. IEEE Trans. Netw. Serv. Manage. 11 (1), 90–100.

Li Chunlin is a Professor of Computer Science in Wuhan University of Technology. She received the M.E. in Computer Science from Wuhan Transportation University in 2000, and Ph.D. in Computer Software and Theory from Huazhong University of Science and Technology in 2003. Her research interests include cloud computing and distributed computing.

Zhang Jing received her BE degree in Computer Science and M.E. degree in Software Engineering from Xidian University in 2005 and 2008 respectively. She now is a Ph.D. student in Wuhan University of Technology. Her research interests include distributed computing, cloud computing and big data.

Chen Yi received his Ph.D. degree in Beijing Institute of Technology. She is now an Professor of school of computer and information engineering in Beijing Technology and Business University. Her research interests include Information Visualization and Visual Analytics, Intelligent Data Processing, Big Data Technology for Food Safety, Data Mining and Machine Learning. She has published more than 90 papers.

Luo Youlong He is a vice Professor of Management at Wuhan University of Technology. He received his M.S. in Telecommunication and System from Wuhan University of Technology in 2003 and his Ph.D. in Finance from Wuhan University of Technology in 2012. His research interests include cloud computing and electronic commerce.