Big Data Research ••• (••••) •••–•••

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

A Dynamic Data Placement Strategy for Hadoop in Heterogeneous Environments ☆,☆☆

Chia-Wei Lee [a], Kuang-Yu Hsieh [a], Sun-Yuan Hsieh [a,b,*], Hung-Chang Hsiao [a]

[a] *Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 70101, Taiwan*
[b] *Institute of Manufacturing Information and Systems, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan*

A R T I C L E   I N F O

Article history:
Available online xxxx

Keywords:
MapReduce
Hadoop
Heterogeneous
Data placement

A B S T R A C T

Cloud computing is a type of parallel distributed computing system that has become a frequently used computer application. MapReduce is an effective programming model used in cloud computing and large-scale data-parallel applications. Hadoop is an open-source implementation of the MapReduce model, and is usually used for data-intensive applications such as data mining and web indexing. The current Hadoop implementation assumes that every node in a cluster has the same computing capacity and that the tasks are data-local, which may increase extra overhead and reduce MapReduce performance. This paper proposes a data placement algorithm to resolve the unbalanced node workload problem. The proposed method can dynamically adapt and balance data stored in each node based on the computing capacity of each node in a heterogeneous Hadoop cluster. The proposed method can reduce data transfer time to achieve improved Hadoop performance. The experimental results show that the dynamic data placement policy can decrease the time of execution and improve Hadoop performance in a heterogeneous cluster.

© 2014 Published by Elsevier Inc.

1. Introduction

In recent years, with the rapid development of the Internet, network service has become one of the most frequently used computer applications. Search engine, webmail, and social network services are currently indispensable data-intensive applications. Because increasingly more people use web services, processing a large amount of data efficiently can be a substantial problem. Currently, the method for processing a large amount of data involves adopting parallel computing. In 2004, Google proposed MapReduce [10]. Since then the Google File System [11] and Bigtable [8] have used MapReduce to construct a data center that can process at least 20 petabytes a day. Because of the scalability, simplicity, and fault tolerance of the MapReduce model, it is frequently used in parallel data processing in large-scale clusters. Yahoo! [6] is the main developer of Hadoop, which is the most famous open source that implements the Google MapReduce model [2,3]. Hadoop is used to process hundreds of terabytes of data on Linux with 10,000 cores. In addition, Facebook [7] and Amazon [1] have also adopted Hadoop to manage and process large amounts of data.

MapReduce exhibits several advantages that differ from those of traditional parallel computing systems. First, regarding scalability, even when new machines are added to a cluster, the system still works well without reconstruction or much modification. Second, regarding fault tolerance, the MapReduce model can automatically manage failures and mitigate complexity of fault tolerance mechanisms. When a machine fails, MapReduce moves the task that was run on the failed machine to be rerun on another machine. Third, regarding simplicity, programmers can use the MapReduce model without needing to understand thoroughly the details of parallel distributed programming. A program executed using the MapReduce model partitions jobs into a numerous tasks to be assigned and run on multiple nodes in the cluster, and the program collects the processing results of each node to be return.

In the Hadoop architecture, data locality is one of the crucial factors affecting Hadoop performance. However, in a heterogeneous environment, the data required for performing a task is often nonlocal, which affects the performance of Hadoop. In a Hadoop default environment, each node has the same ability of execution and hard disk capacity in a homogeneous cluster. When data are written into the Hadoop distributed file systems (short HDFS), the data are partitioned into numerous blocks of the same size, and Hadoop balance the load by distributing the blocks to

☆ This article belongs to Scalable Computing for Big Data.
☆☆ This work was supported in part by National Science Council under Grant NSC 100-2221-E-006-116-MY3.

* Corresponding author at: Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 70101, Taiwan.
E-mail addresses: cwlee@csie.ncku.edu.tw (C.-W. Lee), p76001027@mail.ncku.edu.tw (K.-Y. Hsieh), hsiehsy@mail.ncku.edu.tw (S.-Y. Hsieh), hchsiao@csie.ncku.edu.tw (H.-C. Hsiao).

http://dx.doi.org/10.1016/j.bdr.2014.07.002
2214-5796/© 2014 Published by Elsevier Inc.

each node equally. Such a data placement strategy can achieve load balance, which is highly efficient and practical in a homogeneous cluster. In a homogeneous environment, each node is assigned to the same workload, implying that moving a large amount of data from one node to another is unnecessary. However, in a heterogeneous environment, the nodes of the execution and hard disk capacity may not be the same. If these nodes are executed using the Hadoop default strategy, a higher execution capacity node completes tasks with local data blocks faster than the lower execution capacity nodes can. After executing tasks with local data blocks, the faster nodes process the tasks with nonlocal data blocks. Because the unprocessed data blocks may be located in slower nodes, additional overhead is required to move these data blocks from a fast node to a slow node. The nodes demonstrating different abilities create the need for more data blocks to be moved; thus, the extra overhead becomes higher, resulting in decreased overall performance of Hadoop. Therefore, we develop a dynamic data placement policy in a heterogeneous environment to reduce the transmission time required to move data blocks from fast nodes to slow nodes, thereby improving the performance of Hadoop.

In this study, we develop a data placement strategy that is different from the Hadoop default strategy. The original strategy in Hadoop assumes that each node has the same capacity in a homogeneous environment, and each node is assigned to the same workload. However, in a heterogeneous environment, this strategy may reduce the efficiency of Hadoop. In the proposed strategy, the data are assigned to nodes according to their execution abilities. Although the data or resources used by a node can be decreased or increased after the data are written into HDFS, the proposed strategy can dynamically adjust and reallocate data. In an experiment, we compare the proposed strategy with the Hadoop default strategy. We adopt two jobs, WordCount and Grep, to test the strategies. For the WordCount job, we improve the execution time by nearly 14.5%, and grep could be improved the execution time of nearly 23.5%.

The remainder of the paper is organized as follows. Section 2 describe the Hadoop system architecture, MapReduce model, HDFS, and motivation for this study. Section 3 describes additional details about the proposed strategy that involves selecting the node to allocate data blocks and why they are reallocated. Section 4 presents the experimental results. Finally, Section 5 presents the conclusion of this study.

2. Related works and motivation

This section introduces the Hadoop system and motivation for this study. Section 2.1 describes the features of Hadoop. Section 2.2 describes the MapReduce model, Section 2.3 presents an overview of HDFS. Section 2.4 describes the details of the problem to be solved.

2.1. Hadoop

Hadoop is a well-known implementation of the MapReduce model platform, which is an open-source supported by the Apache Software Foundation. Hadoop is user-friendly for distributed application developers because it mitigates complicated managements. Hadoop consists of two main parts: MapReduce [5] and Hadoop Distributed File System (HDFS) [4], in which MapReduce is responsible of parallel computing and the HDFS is responsible for data management. In the Hadoop system, MapReduce and HDFS management parallel process jobs and data, respectively. Hadoop partitions a job and data into tasks and blocks, and assigns them to nodes in a cluster. Hadoop adopts master/slave architecture, in which a master node manages other slave nodes in the cluster. In the MapReduce model, the master is called *JobTracker*, and each slave is called *TaskTracker*. In the HDFS, the master is called *NameNode*, and each slave is called *DataNode*. Job and data distributions are managed by the master to assign nodes for computing and storing. In the default setting of Hadoop, node computing capacity and storage capacity are the same in the Hadoop cluster. In such a homogeneous environment, the data placement strategy of Hadoop can enhance the efficiency of the MapReduce model.

Hadoop uses the distributed architecture that can greatly improve the efficiency of reading and computing, and also uses numerous general PCs that can build a high-performance computing platform. Spending large amounts of money to buy high-end servers is unnecessary. For example, assume that the price of a high-end server can buy 10 or more PCs, but the performance of a high-end server is lower than 10 sets of the overall performance of the PCs. This can further reduce the cost for the data center. This is also one of the reasons why Hadoop is frequently used.

2.2. MapReduce

MapReduce is a programming model used in clusters that have numerous nodes and use considerable computing resources to manage large amounts of data in parallel. MapReduce is proposed by Google in 2004. In the MapReduce model, an application to be executed is called a "job". A job can be divided into two parts: "map tasks" and "reduce tasks", in which the map-tasks run the map function and the reduce-tasks run the reduce function. Map function processes input data assigned by the master and produce many intermediate $\langle \text{key}, \text{value} \rangle$ pairs. Based on $\langle \text{key}, \text{value} \rangle$ pairs that are generated by map function processes, the reduce function then merges, sorts, and finally returns the result.

The MapReduce model mainly entails applying the idea of divide and conquer. It distributes a large amount of data to many nodes to perform parallel processing, which reduces the execution time and improves the performance. At runtime, input data are divided into many of the same sized data blocks; these blocks are then assigned to nodes that perform the same map function in parallel. After the map function is performed, the generated output is an intermediate datum composed of several $\langle \text{key}, \text{value} \rangle$ pairs. The nodes that perform the reduce function obtain these intermediate data, and finally generate the output data. Fig. 1 shows the MapReduce flow chart.

When executing jobs, the JobTracker manages all jobs scheduling and task distributions, and each TaskTracker is responsible for performing tasks and returns the results to the JobTracker. JobTracker and TaskTrackers use heartbeat message to communicate and transfer data. Each TaskTtracker has a different number of task slots, which are usually based on the performance of nodes, to set the number of task slots. Each task slot can perform a task at once. When a TaskTracker has empty task slots, it uses the regular heartbeat message to inform the JobTracker. If some tasks have yet to be performed, the JobTracker uses a heartbeat response for assigning tasks to TaskTrackers and sends the information required to perform the tasks. When the task is completed, a TaskTracker uses a heartbeat for sending data and informing the JobTracker.

The advantage of MapReduce is that it is easy to use. By using this model, many parallel computing details are hidden. The system automatically assigns nodes that differ from the mapper and reducer for computing. When programming, a programmer does not need to spend extra time on data and program division. Therefore, a programmer does not need to have a deep understanding of parallel computing. A programmer must simply focus on the normal function processing rather than the parallel processing. This can simplify the application development process substantially and shorten the development time.

In recent years, MapReduce has been used increasingly more in large-scale distributed systems. For example, Google uses MapReduce to manage large amounts of data daily. Yahoo! is developed

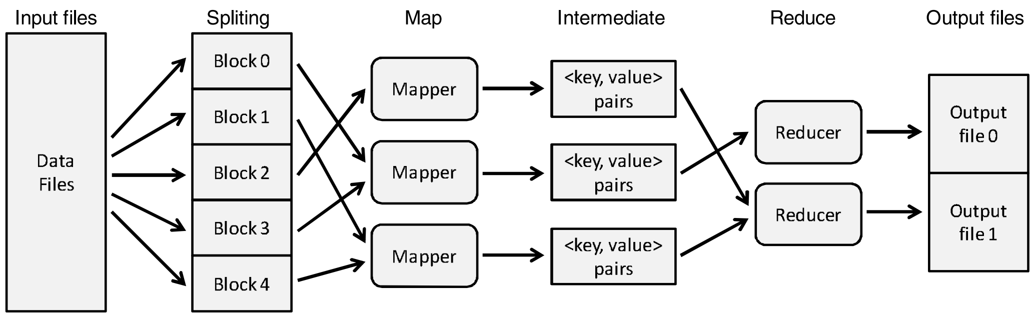


Fig. 1. The overview of the MapReduce model.

Hadoop, which is a platform that involves using the MapReduce model, and Mars [12] also entails using the MapReduce model developed as a graphics processing unit (GPU) hardware platform.

2.3. HDFS

The HDFS is implemented by Yahoo! based on the Google File System, which is used with the MapReduce model. It consists of a NameNode and many DataNodes. The NameNode is responsible for the management of the entire file system, file information (such as namespace and metadata), and storage and management. It also partitions the files that are written in HDFS into many same sized blocks, and then allocates these blocks to different DataNodes. By contrast, DataNodes are responsible for storing data blocks. The initial default block size of the HDFS is 64 MB. When a file is less than 64 MB and does not take up an entire block, it does not waste the extra space. When an HDFS client reads data from the HDFS, it asks the NameNode to find DataNodes that have data blocks that must be read, and then data from those DataNodes are read simultaneously and finally combined into a complete file. When writing data, an HDFS client first requests the NameNode for creating a file. After the NameNode accepted, the HDFS client directly writes the file to the assigned DataNodes.

Using this distributed file system, the read rate is much faster than that of a single node. When a larger file must be read, the file can be read from many nodes simultaneously, which is more efficient than a single node read. This reduces the time of access data and is similar to the average depletion of a hard disk, which further reduces the cost of hardware maintenance. Regarding large data storage, even if the size of the files that must be written to the HDFS is larger than one physical hard disk capacity, the file can be divided into several blocks to be stored. For example, each node in the cluster is only 500 GB of hard disk capacity, but this cluster could still be used to store files of over 1 TB even 1 PB, as long as the total hard disk capacity contributed by all nodes in the cluster is adequately large. Fig. 2 shows a simplified HDFS architecture diagram.

The Hadoop default data placement strategy assumes that the storage capacity of each node is the same, and Hadoop balances loads by distributing the blocks to each node averagely. Under this assumption, the NameNode assigns map tasks to DataNodes with data blocks needing to be processed. This can not only reduce unnecessary data transfer time but achieve the load balancing effect and further enhance the effectiveness of Hadoop.

2.4. Motivation

Hadoop assumes that the computing capacity of each node in a cluster is the same. In such a homogeneous environment, each node is assigned to the same load, and thus it can fully use the resources in the cluster. There would not be many nodes that are idle or overloaded. However, in real-world applications, clusters are often worked in a heterogeneous environment [9,13–15]. In such a cluster, there is likely to be various specifications of PCs or servers, which causes the abilities of the nodes to differ. If such a heterogeneous environment still uses the original Hadoop strategy that distributes data blocks into each node equally and the load is also evenly distributed to each node, then the overall performance of Hadoop may be reduced. The major reason is that different computing capacities between nodes cause the task execution time to differ so that the faster execution-rate nodes finish processing local data blocks faster than other slower nodes do. At this point, the master still assigns non-performed tasks to the idle faster node, but this node does not own the data needed for processing. The required data should be transferred from another node through the network. Because waiting for the data transmission time increases the task execution time, it causes the entire job execution time to become prolonged. A large number of moved data affects the overall Hadoop performance.

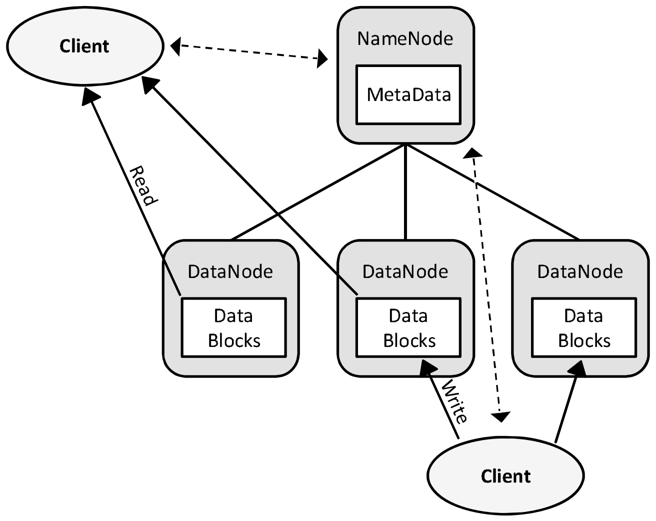


Fig. 2. The overview of HDFS read and write.

For example, in Fig. 3, three nodes are in the cluster, and the computing capacities of the three nodes are different. The computing capacity of Node A is fastest, followed by Node B, and Node C is the slowest. Suppose that the computing capacity of Node A is two times faster than Node B, and three times faster than Node C. As shown in Fig. 3(a), the processing job requiring data blocks are near-equally distributed in each node: Node A has three data blocks, Node B has four, and Node C has four. After the job begins execution, Node A will be the fastest finished node that processes the data blocks stored in Node A. As shown in Fig. 3(b), at this point, Node B and Node C are finished processing one block respectively; they then process the second block that has not yet been processed. Because Node A has an empty task slot, the NameNode assigns the unprocessed task to Node A; Node A must then wait

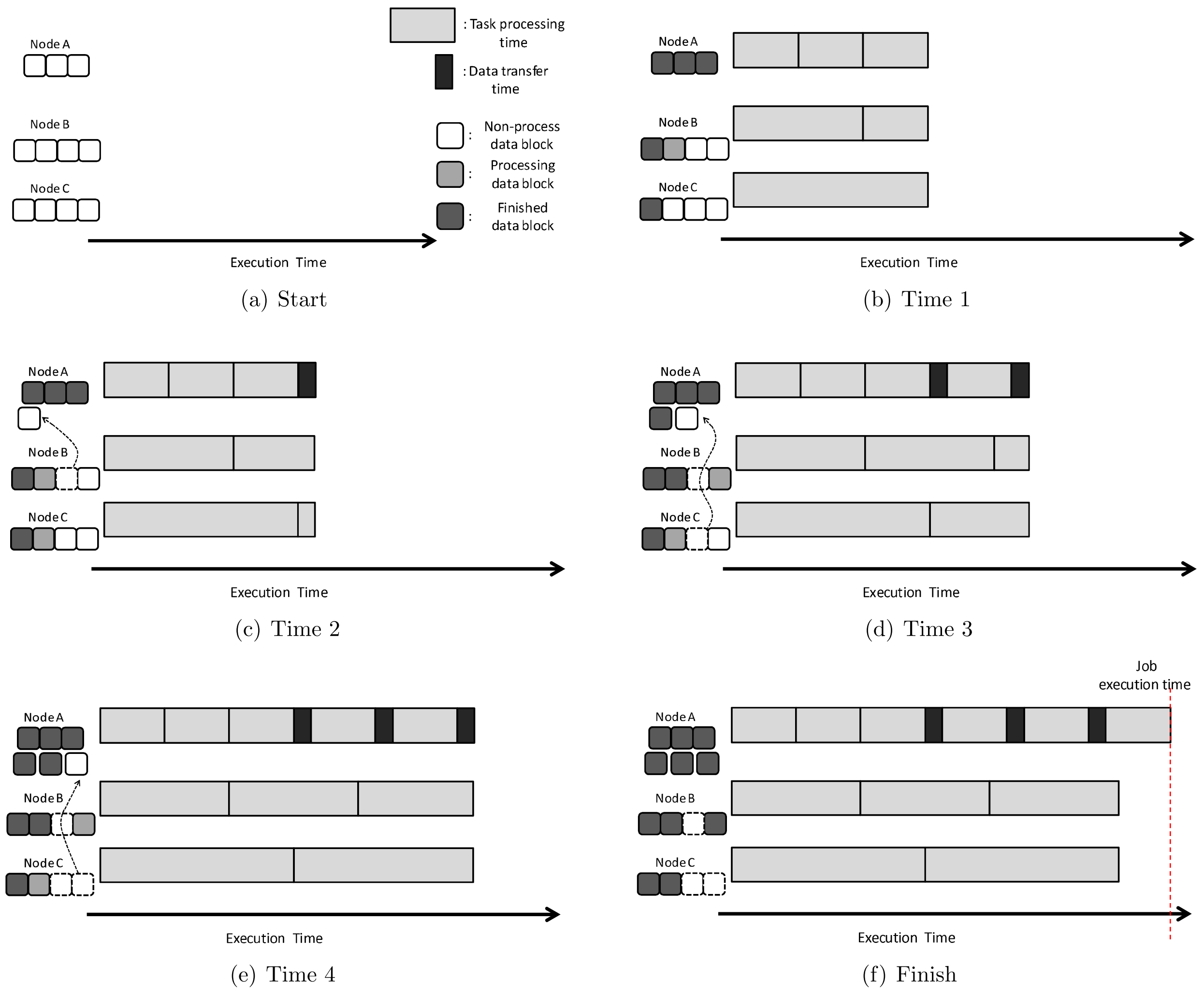


Fig. 3. The default data allocation strategy of Hadoop.

for the processing task that requires the data transfer from Node B or C (from B to A is assumed in Fig. 3(c)). As shown in Fig. 3(c), Node A should wait until the data transfer is finished before continuing. Finally, as shown in Fig. 3(f), Node A should transfer three data blocks from the other two nodes. This extends the job execution time. A job that has a large amount of data blocks needing to be transferred, affects Hadoop performance.

If the amount of transferred data could be reduced, it could effectively reduce the idle time of the node waiting for data transfer, then reduce the task execution time, and further improve the performance of Hadoop. As mentioned, if the average distribution in three nodes data blocks in accordance with three node computing capacity reallocation, the total execution time can be further reduced. Suppose that Node A has six data blocks, Node B has three, while Node C has two. As shown in Fig. 4, when performing job, Node A performs fastest, but Node A has the data blocks more than Node B and Node C, no need to transfer data blocks, nodes could use local data to perform tasks. Therefore, the job execution time and usage rate of the network can be reduced.

3. Dynamic data placement strategy

In a heterogeneous cluster, the computing capacity for each node is not the same. Moreover, for different types of job, the computing capacity ratio of nodes are also not the same. Therefore, a Dynamic Data Placement (DDP) strategy is presented according to the types of jobs for adjusting the distribution of data blocks. The proposed algorithm, namely DDP, consists of two main phases: the first phase is performed when the input data are written into the HDFS, and the second phase is performed when a job is processed. Section 3.1 presents the core of the proposed algorithm: how to build a *RatioTable*. Section 3.2 presents Phase 1: initial allocation of data. Section 3.3 presents Phase 2: capacity decision and data reallocation.

3.1. RatioTable

When Hadoop starts, a RatioTable is created in the NameNode, which is used to determine the allocation ratio of data blocks in nodes when the data is written into the HDFS, and is used to determine whether the data blocks must be reallocated when the job is executed. The RatioTable records the types of jobs and the computing capacity ratio of each node. The computing capacity of each node is based on the average execution time of a single task in that node. The NameNode calculates the computing capacity ratio for each node according to the task execution time return by the heartbeat message of each DataNode.

Table 1 shows an example of the RatioTable. There are three nodes in the cluster in which the computing capacity of each node is not the same: Node A is the fastest, followed by Node B, and Node C is the slowest. The cluster performs both jobs, WordCount and Grep. Hence, there are two jobs recorded in the RatioTable. For the WordCount job, the computing capacity ratio between nodes is 3:1.5:1, in which Node A is three times faster than Node C, and

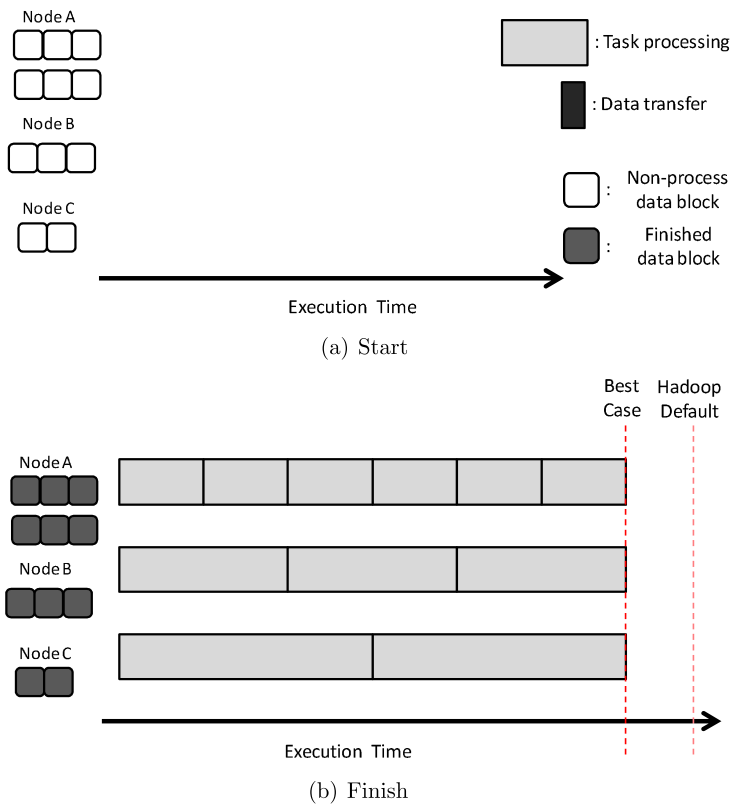


Fig. 4. The best case of data allocation.

Table 1
RatioTable example.

	Node A	Node B	Node C
WordCount	3	1.5	1
Grep	2.5	1.5	1

Node B is one and a half times faster than Node C. For the Grep job, the ratio is 2.5:1.5:1, in which Node A is two and a half times faster than Node C, and Node B is one and a half times faster than Node C.

3.2. Phase 1

When data are written into the HDFS, NameNode first checks the RatioTable. These data are used to determine whether this type of job has been performed. If the RatioTable has a record of this job, the newly written data will be allocated to each node in accordance with the computing capacity that records in the RatioTable. If the RatioTable has no record of this job, the data will be equally distributed to the nodes in the cluster, and the NameNode will add a new record of this type of job in the RatioTable. Each node's computing capacity will be set at 1 for this type of job.

As shown in Table 1, if there are data to be written into the HDFS, assume that these data can be partitioned into 11 data blocks. If the data are performed for WordCount, then the data will be allocated according to the ratio recorded in the RatioTable. Therefore, Node A is assigned six ($11 * [\frac{3}{3+1.5+1}] = 6$) data blocks, Node B is assigned three ($11 * [\frac{1.5}{3+1.5+1}] = 3$) data blocks, and Node C is assigned two ($11 * [\frac{1}{3+1.5+1}] = 2$) data blocks. If the performed job is TeraSort, the NameNode will check the RatioTable and find no record of TeraSort. In this case, the data will be equally distributed to the three nodes, a record for TeraSort is then created for the RatioTable, and the computing capacity among Node A, Node B and Node C for TeraSort to be set at 1:1:1. Fig. 5 shows the flow chart of Algorithm 1 (Phase 1).

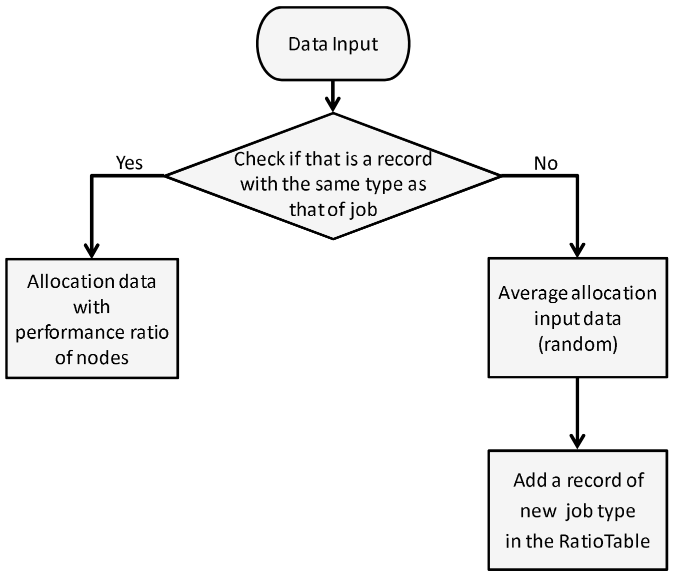

Data Input
Check if that is a record with the same type as that of job
Yes
No
Allocation data with performance ratio of nodes
Average allocation input data (random)
Add a record of new job type in the RatioTable

Fig. 5. The flow chart of Phase 1.

3.3. Phase 2

Phase 2 starts when the job begins execution; as the job starts executing, each node will first receive the first batch of tasks. When the task finishes executing in each DataNode, all DataNodes will return the task execution time to the NameNode. The NameNode calculates the computing capacity ratio of this job for each node according to those execution time. However, each node has a different number of task slots. In a DataNode, the tasks in task

Algorithm 1: Initial_Data_Allocation.

```
1  When a data is written into HDFS:
2  JobType ← job type of the data will be performed;
3  DataSize ← obtain from data information;
4  BlockSize ← set by user;
5  TotalBlockNumber = $\lceil \frac{DataSize}{BlockSize} \rceil$;
6  set Same = 0;
7  for each record in the RatioTable do
8      if compare JobType with record are the same then
9          Same = 1;
10         ComputingCapacityRatio ← obtain from record;
11         for each DataNode in the cluster do
12             NodeCapacity ← obtain from ComputingCapacityRatio;
13             BlockNumber = TotalBlockNumber * $[\frac{NodeCapacity}{\sum each\ node\ capacity}]$;
14             Allocate BlockNumber data blocks to the DataNode;
15 if Same = 0 then
16     ComputingCapacityRatio ← set 1 for each node;
17     Add JobType with ComputingCapacityRatio to RatioTable;
18     for each DataNode in the cluster do
19         NodeCapacity = 1;
20         BlockNumber = TotalBlockNumber * $[\frac{NodeCapacity}{\sum each\ node\ capacity}]$;
21         Allocate BlockNumber data blocks to the DataNode.
```

slots can be processed in parallel. This causes the computing capacity ratio that is calculated based on the task execution time to be inaccurate. Therefore, the task execution time requires calculating the ratio of computing capacity for the nodes, and the number of task slots must be considered. Therefore, the computing capacity adopts the average time required to complete one task.

For example, there are two node: Node A and Node B in which Node A is two times faster than Node B. Moreover, assume that the map task slot number of Node A is four, and the number of Node B is 2. Assume that the times required by Node A to execute four tasks are 45, 43, 43, and 46 seconds, respectively. The four tasks are performed simultaneously, which on average, requires 44.25 seconds to complete one task. The times required by Node B to execute two tasks are 39 and 40 seconds, which takes an average execution time of 39.5 seconds. If it is just in accordance with the average execution time to compare the efficiency of nodes, Node B is efficiency than that of Node A, but actually Node A is two times faster than Node B. Although the Node B average execution time is shorter than that of Node A, but the number of task slots set for Node A and Node B are not the same; Node A can execute four tasks simultaneously, and Node B can perform only two tasks simultaneously. Therefore, it is reasonable to compute the average time required to complete a task obtained by letting the execution time be divided by the number of task slots. For mathematical formulas, let $T_{avg}(X)$ denote the average execution time to complete a batch of tasks in node X, let $S(X)$ denote the number of task slots set for X, and $T_t(X)$ denote the average time required to complete one task for X. Then, $T_t(X) = [\frac{T_{avg}(X)}{S(X)}]$. As in the mentioned example, $T_t(A) = 11$ and $T_t(B) = 19$. Therefore, the efficiency of Node A is close two times faster than Node B.

The NameNode will use the $T_t(X)$ of each node X to calculate the computing capacity ratio of each node. After the NameNode calculates the computing capacity ratio, it will compare the record of the RatioTable. If the computing capacity ratio and record are the same, then it will not be transferred to any data blocks. However, if they are not same, then data blocks will be transferred according to the new ratio calculated by the NameNode, and the NameNode will modify the record at the RatioTable. The transferred data blocks are processed in the background, and the Hadoop job does not need to wait until the data transfer is completed to be executed. Fig. 6 is the flow chart of Algorithm 2 (Phase 2).

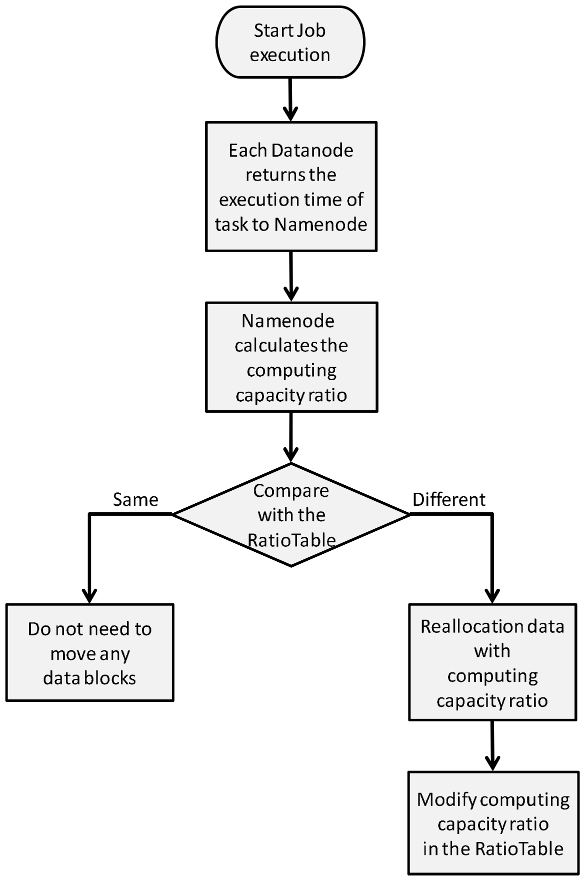

Start Job execution
Each Datanode returns the execution time of task to Namenode
Namenode calculates the computing capacity ratio
Compare with the RatioTable
Same
Different
Do not need to move any data blocks
Reallocation data with computing capacity ratio
Modify computing capacity ratio in the RatioTable

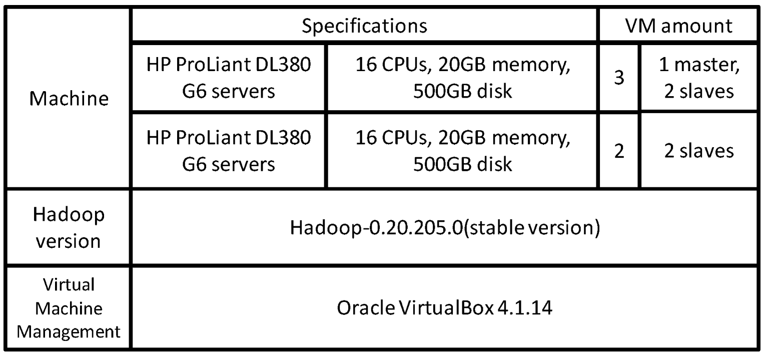
Fig. 6. The flow chart of Phase 2.

Algorithm 2: Capacity_Decision_and_Data_Reallocation.

```
1  When a job start:
2  NodeNumber ← obtain from NameNode;
3  CurrentNumber[NodeNumber] ← set 0 for all entries; // record the
       number of task execution time received from each
       node.
4  TotalExecutionTime[NodeNumber] ← set 0 for all entries;
5  while receive the task execution time from DataNode[i] do
6      SlotNumber ← obtain from DataNode[i];
7      ExecutionTime ← task execution time;
8      TotalExecutionTime[i] = TotalExecutionTime[i] + ExecutionTime;
9      CurrentNumber[i] = CurrentNumber[i] + 1;
10     if CurrentNumber[i] = SlotNumber then
11         $T_{avg} = \frac{TotalExecutionTime[i]}{SlotNumber}$;
12         $T_t = \frac{T_{avg}}{SlotNumber}$;
13         CurrentNumber[i] = 0;
14         TotalExecutionTime[i] = 0;
15     if obtain $T_t$ for each node then
16         PerformanceRatio ← PerformanceRatio and $T_t$ are inversely
           proportional;
17         for record in the RatioTable do
18             if compare PerformanceRatio with record are different then
19                 Reallocation data blocks according to PerformanceRatio;
20                 Modify the record according to PerformanceRatio.
```

4. Experimental results

This section presents the experimental environment and the experimental results for the proposed algorithm.



	Specifications		VM amount	
Machine	HP ProLiant DL380 G6 servers	16 CPUs, 20GB memory, 500GB disk	3	1 master, 2 slaves
	HP ProLiant DL380 G6 servers	16 CPUs, 20GB memory, 500GB disk	2	2 slaves
Hadoop version	Hadoop-0.20.205.0(stable version)			
Virtual Machine Management	Oracle VirtualBox 4.1.14			

Fig. 7. Experimental environment.

Table 2
Each node specification.

Node	CPU	Memory	Disk
Node A (Master)	2	2 GB	50 GB
Node B (Slave)	4	4 GB	50 GB
Node C (Slave)	2	2 GB	50 GB
Node D (Slave)	1	1 GB	50 GB
Node E (Slave)	1	1 GB	50 GB

4.1. Environment

The experimental environment is shown in Fig. 7. We use two HP ProLiant DL380 G6 servers and each one has 16 CPUs, 20 GB memory, and 500 GB disk. We use VirtualBox 4.1.14 to create our computing node. In order to achieve the effect of a heterogeneous environment, the capacity of the nodes are not the same. We set different amounts of CPU and memory on each node. In total, we create the five virtual machines: one master and four slaves. One virtual machine as the master has 2 CPUs, 2 GB of memory, and 50 GB disk; one virtual machine as a slave has 4 CPUs, 4 GB of memory, and a 50 GB disk; one virtual machine as a slave has 2 CPUs, 2 GB of memory, and a 50 GB disk; two virtual machines as slaves both have 1 CPU, 1 GB of memory, and a 50 GB disk. Table 2 presents the specifications of each node. All of the virtual machines adopt the operating system as Ubuntu 12.04 LTS, and the used Hadoop version is Hadoop-0.20.205.0.

4.2. Result

WordCount and Grep are two types of jobs run to evaluate the performance of the proposed algorithm in a Hadoop heterogeneous cluster. WordCount and Grep are MapReduce applications running on a Hadoop cluster. WordCount is an application used for counting the words in the input file, and Grep is used to search for regular expressions. In each round, 10 jobs are run simultaneously, and each job processes different input data respectively, in which the size of all input data is 1 GB. The experimental data are run in 10 rounds in which each round runs 10 jobs, to average the execution time.

First, the execution time of WordCount and Grep are measured for each node to perform different sizes of data.

Fig. 8 shows that the average execution time of 1 GB and 2 GB WordCount job respectively. Fig. 9 shows the average execution time of 1 GB and 2 GB Grep job respectively. As shown in Fig. 8 and Fig. 9, the performed data size does not affect the computing capacity ratio between nodes. The execution time of each node is proportional to the data size.

The data shown in Fig. 8 and Fig. 9 are adopted to calculate the computing ratio of each node for two types of jobs, as listed in Table 3. According to the experimental results, regardless of the size of data to be processed, each node's computing capacity ratio is maintained. For a WordCount job, based on the average time of one job execution, Slave 1 is four times faster than Slave 3, and Slave 2 is two times faster than Slave 3.

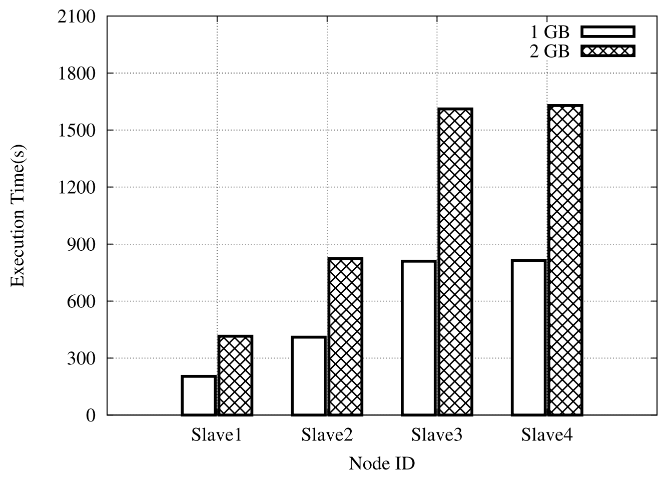


Fig. 8. Execution time of WordCount job on each node.

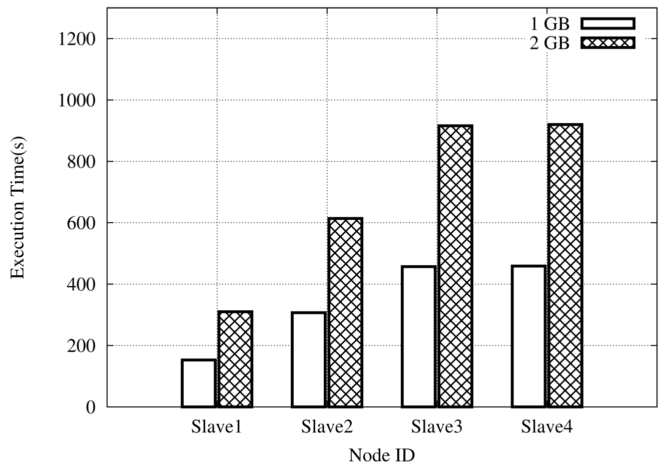


Fig. 9. Execution time of Grep job on each node.

Table 3
The job computing capacity ratio of each node.

Node	Job type	
	WordCount	Grep
Slave 1	4	3
Slave 2	2	1.5
Slave 3	1	1
Slave 4	1	1

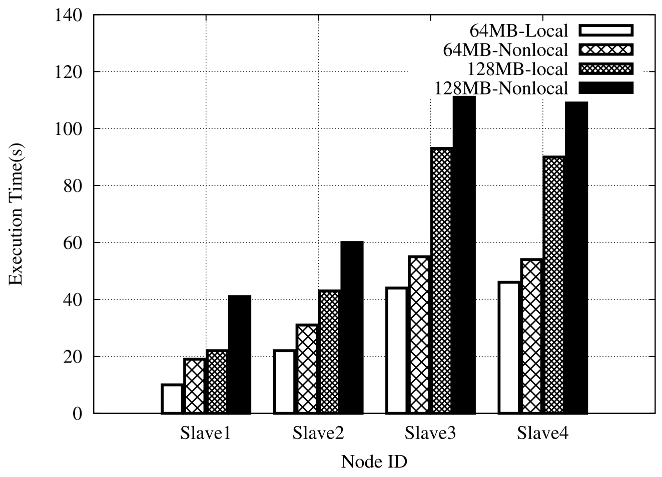


Fig. 10. Average time required to complete a WordCount task on each node.

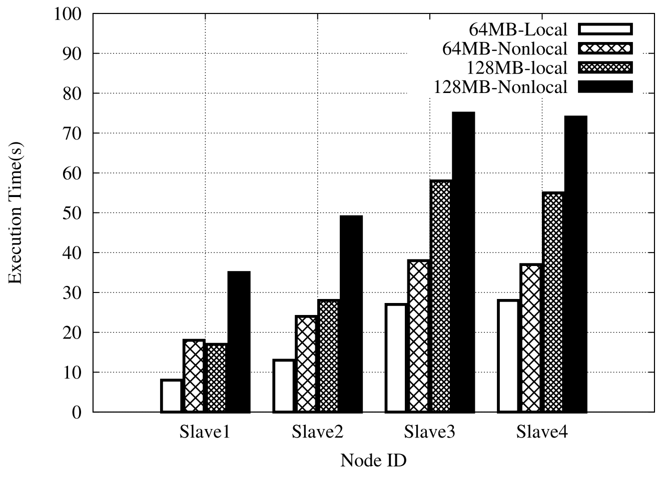


Fig. 11. Average time required to complete a Grep task on each node.

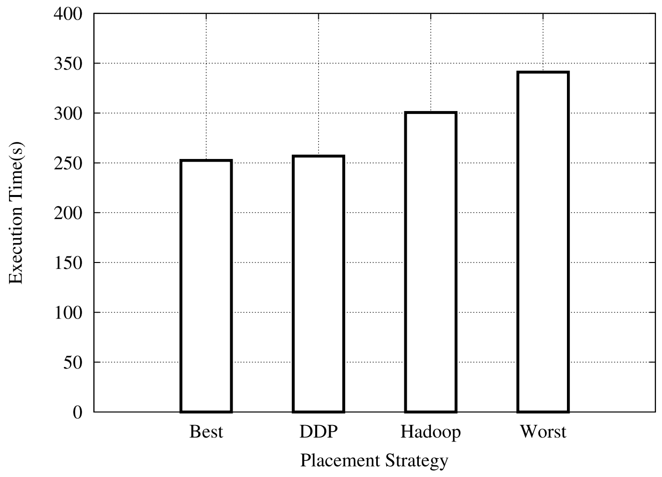


Fig. 12. Impact of data placement strategy on average execution time of WordCount.

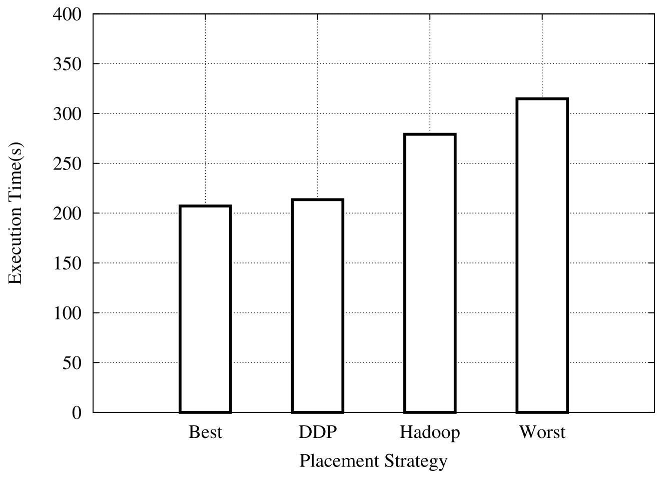


Fig. 13. Impact of data placement strategy on average execution time of Grep.

However, when Hadoop perform jobs, it spends extra time to start up and clean up, and the tasks are not only map tasks but also include reducing tasks. The proposed algorithm mainly involves the part of the map tasks to optimize. However, according to the job execution time, calculating the computing ratio may cause slight differences. Therefore, different jobs are considered to measure the average time required to complete a single task.

Fig. 10 shows the average time required to complete a single WordCount local task and nonlocal task. Fig. 11 shows the average time required to complete a single Grep local task and nonlocal task. As shown in Fig. 10 and Fig. 11, the executing time offset of each node for executing the local and nonlocal tasks is the same. Regardless of computing capacity, the time offset of executing local tasks and nonlocal tasks depends on the size of a data block.

When the size of a data block is set to be larger, the time offset of executing a local task and nonlocal task is much larger. According to the experimental results, the local task execution time is proportional with the data block size. Therefore, when calculating the computing capacity ratio of each node, the local task execution time is adopted. The setting of a data block size does not affect the computing ratio of each node. Therefore, the experimental data block size involves using the Hadoop default 64 MB.

Table 4 shows the computing capacity ratio of the average time required to complete a single task. This ratio is calculated according to the experimental results shown in Fig. 10 and Fig. 11. For a WordCount job, the average time required to complete a single task, Slave 1 is four and a half times faster than Slave 3, Slave 2 is 2 times faster than Slave 3. For a Grep job, Slave 1 is three and a half times faster than Slave 3, Slave 2 is 2 times faster than Slave 3. This ratio are slightly different from the ratio calculated with job execution time, because the job is not only map task also includes other actions. Therefore, we adopt the ratio of Table 4 records to treat as the computing capacity ratio.

Figs. 12 and 13 represent the average execution time of a WordCount and a Grep job, compared with the proposed algorithm and another three data allocation strategies. In this experiment, we used the size of each data as 2 GB, and the data block size is set at 64 MB. Each round executes 10 jobs, and perform a total of five rounds in which each job processes different data files. Finally, the average execution time of one job is used for evaluation.

As shown in Fig. 12 and Fig. 13, the proposed DDP algorithm is compared with the Hadoop default strategy, the best case data distribution, and the worst case data distribution. The best case of distribution represents all of the task processing local data in which no data are required to be transferred. If the data are locally processed, the data blocks must be allocated according to the computing capacity ratio of each node. Therefore, data blocks are allocated in accordance with the ratio recorded in Table 4. For example, suppose that a WordCount job type of data must be written into the HDFS. These data can be partitioned into 17 data blocks, and the ratio recorded in Table 4 is 4.5, 2, 1, 1. Therefore, if data blocks are allocated according to the ratio, then Slave 1 is assigned nine data blocks, Slave 2 is assigned to four blocks, Slave 3 and Slave 4 are each assigned two blocks. The worst case indicates

Table 4
The task computing capacity ratio of each node.

Node	Task type	
	WordCount	Grep
Slave 1	4.5	3.5
Slave 2	2	2
Slave 3	1	1
Slave 4	1	1

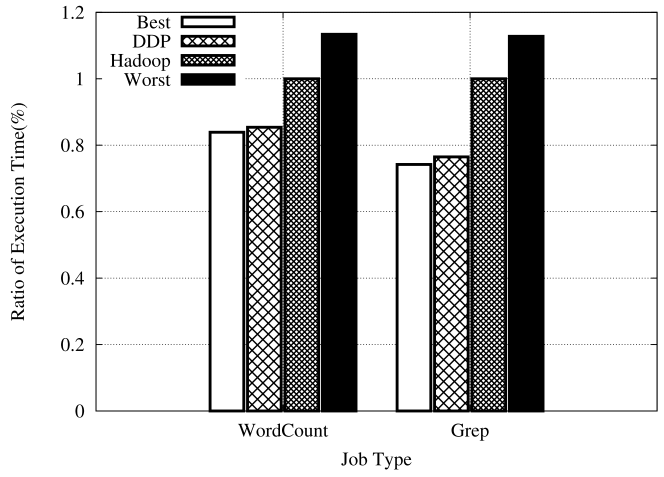


Fig. 14. The percent of average execution time of a job compare with Hadoop default strategy.

that all of the data blocks are concentrated on the same node so that other nodes performing tasks must wait until data blocks are transferred from the node storing all data blocks, and the node computing capacity must be the slowest. Therefore, the worst case is the data blocks being placed in Slave 3 or Slave 4.

Fig. 12 shows the different types of data placement policy impacts on WordCount job execution time. The average execution time of DDP is closed to the optimal case. Even if using our approach the first time to perform this type of job does not follow computing capabilities to allocate data blocks, it may result in a slight delay at the beginning of execution, but the proposed algorithm will adjust the location of data blocks when job performing, and after this, the data of this job type to be written into the HDFS will be allocated according to the computing ratio. Therefore, the average execution time of the proposed algorithm and the best case are nearly identical. As shown in Fig. 14, the comparison with Hadoop default strategy, DDP can reduce execution time approximately 14.5%. As shown in Fig. 15, DDP compared with the worst case can improve by approximately 24.7%.

Fig. 13 shows the comparison of different data placement policies on Grep job execution time. For the Grep job, after using the proposed algorithm, the average execution time is also extremely close to the best case. As shown in Fig. 14, compared with the Hadoop, the DDP can reduce the execution time by approximately 23.5%. As shown in Fig. 15, compared with the worst case, the DDP can improve by approximately 32.1%.

5. Conclusion

This paper proposes a data placement policy (DDP) for map tasks of data locality to allocate data blocks. The Hadoop default data placement strategy is assumed to be applied in a homogeneous environment. In a homogeneous cluster, the Hadoop strategy

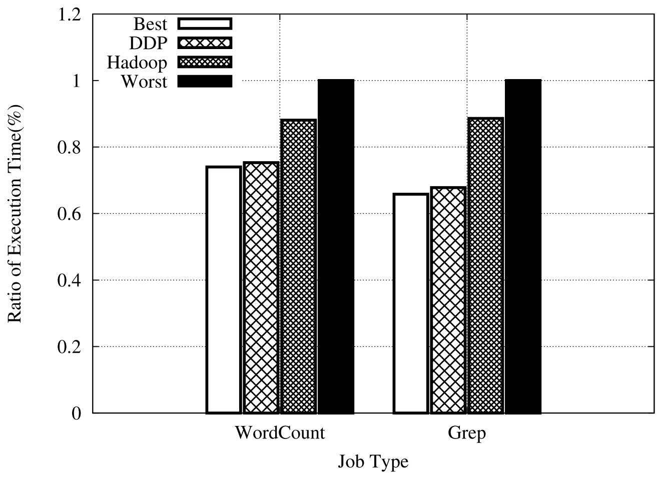


Fig. 15. The percent of average execution time of a job compare with worst case.

can make full use of the resources of each node. However, in a heterogeneous environment, a produces load imbalance creates the necessity to spend additional overhead. The proposed DDP algorithm is based on the different computing capacities of nodes to allocate data blocks, thereby improving data locality and reducing the additional overhead to enhance Hadoop performance. Finally in the experiment, for two types of applications, WordCount and Grep, the execution time of the DDP compared with the Hadoop default policy was improved. Regarding WordCount, the DDP can improve by up to 24.7%, with an average improvement of 14.5%. Regarding Grep, the DDP can improve by up to 32.1%, with an average improvement of 23.5%. In the future, we will focus on other types of jobs to improve Hadoop performance.

References

[1] Amazon Elastic MapReduce, http://aws.amazon.com/elasticmapreduce/.
[2] Apache, http://httpd.apache.org/.
[3] Hadoop, http://hadoop.apache.org/.
[4] Hadoop Distributed File System, http://hadoop.apache.org/docs/stable/hdfs_design.html.
[5] Hadoop MapReduce, http://hadoop.apache.org/docs/stable/mapred_tutorial.html.
[6] Hadoop Yahoo, http://www.ithome.com.tw/itadm/article.php?c=49410&s=4.
[7] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash, J.-S. Sarma, N. Spiegelberg, D. Molkov, R. Schmidt, J. Gray, H. Kuang, A. Menon, A. Aiyer, Apache Hadoop goes realtime at Facebook, in: SIGMOD '11, Athens, Greece, June 12–16, 2011.
[8] F. Chang, J. Dean, S. Ghemawat, W.-C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fiker, R.E. Gruber, BigTable: a distributed storage system for structured data, in: 7th USENIX Symposium on Operating Systems Design and Implementation, OSDI'06, 2006, pp. 205–218.
[9] Q. Chen, D. Zhang, M. Guo, Q. Deng, S. Guo, SAMR: a self-adaptive MapReduce scheduling algorithm in heterogeneous environment, in: 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), IEEE, 2010, pp. 2736–2743.
[10] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: OSDI '04, Dec. 2004, pp. 137–150.
[11] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: Proc. SOSP 2003, 2003, pp. 29–43.
[12] B. He, W. Fang, Q. Luo, N. Govindaraju, T. Wang, Mars: a MapReduce framework on graphics processors, in: ACM 2008, 2008, pp. 260–269.
[13] G. Lee, B.G. Chun, R.H. Katz, Heterogeneity-aware resource allocation and scheduling in the cloud, in: Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud, vol. 11, 2011.
[14] C. Tian, H. Zhou, Y. He, L. Zha, A dynamic MapReduce scheduler for heterogeneous workloads, in: Eighth International Conference on Grid and Cooperative Computing, GCC'09, IEEE, 2009.
[15] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: Proc. OSDI, San Diego, CA, December 2008, pp. 29–42.