

# An Uncoupled Data Process and Transfer Model for MapReduce

Li Zha<sup>1</sup>, Jie Zhang<sup>1,2</sup>, Wei Liu<sup>1,2</sup>(✉), and Jian Lin<sup>1,2</sup>

<sup>1</sup> Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China  
char@ict.ac.cn, {zhangjie,liuwei,linjian}@software.ict.ac.cn

<sup>2</sup> University of the Chinese Academy of Sciences, Beijing, China

**Abstract.** In the original MapReduce model, reduce tasks need to fetch output data of map tasks in the manner of “pull”. However, reduce tasks which are occupying reduce slots cannot start executing until all the corresponding map tasks are completed. It forms the dependence between map and reduce tasks, which is called the coupled relationship in this paper. The coupled relationship leads to two problems: reduce slot hoarding and underutilized network bandwidth. Meanwhile, storing the result data is costly especially when the system has replications, which leads to the inefficient storage problem. We propose an uncoupled data process and transfer model in order to address these problems. Four core techniques, including weighted mapping, data pushing, partial data backup, and data compression are introduced and applied in Apache Hadoop, the mainstream open-source implementation of MapReduce model. This work has been practiced in Baidu, the biggest search engine company in China. A real-world application for web data processing shows that our model can improve the system throughput by 29.5 %, reduce the total wall time by 22.8 %, provide a weighted wall time acceleration of 26.3 %, and reduce the result data stored in disk by 70 %. What’s more, the implementation of this model is transparent to users and compatible with the original Hadoop.

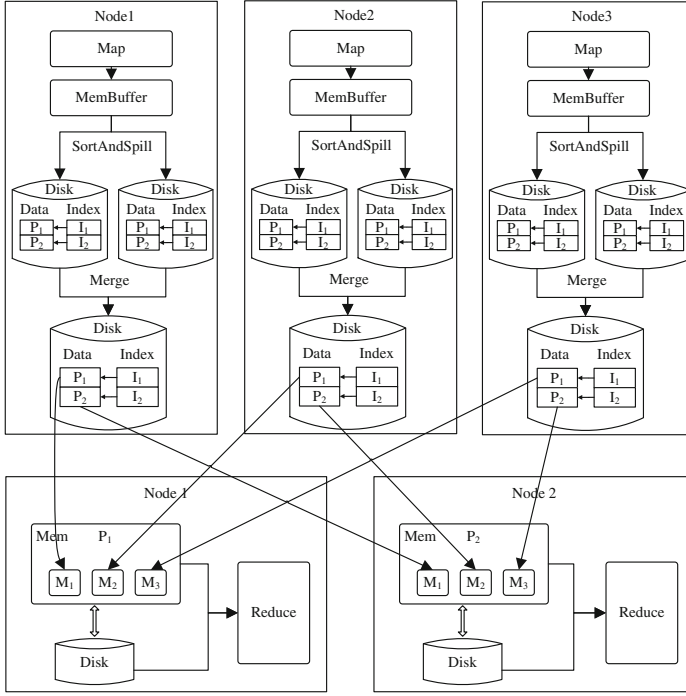
**Keywords:** MapReduce · Data transfer · Uncoupled model · Compression

## 1 Introduction

With the arrival of “big data” era, the original computing and storing systems face great challenges. Platforms which can process and store large data are receiving more and more attention, such as MapReduce [12], Dryad [16], Sector/Sphere [15], and BigTable [8]. The MapReduce programming model proposed by Google has become the mainstream data-centric platform for large data processing because of its scalability and simplicity. Apache Hadoop [1], an open-source implementation of MapReduce, is widely used.

The MapReduce model is a software architecture for parallel computing on large data sets with commercial hardware. A job is divided into map tasks and reduce tasks. Map tasks are responsible for reading the source data, resolving

the data into key-values, and writing the intermediate result into local disk. Reduce tasks read the intermediate result written by corresponding map tasks, resolve the key-values of the same key and write the final result into file system. In the MapReduce architecture, one node works as the master where a JobTracker runs. The JobTracker is responsible for monitoring and managing map and reduce tasks. The other nodes work as slaves where TaskTrackers run. The TaskTrackers are responsible for executing map and reduce tasks. When a job is submitted, the related input data is divided into several splits. The JobTracker will pick up idle TaskTrackers to perform map tasks on the splits, and then perform reduce tasks on the intermediate output of map tasks. The final result will usually be a set of key-value pairs stored in a distributed file system like HDFS [22] and GFS [13].



**Fig. 1.** The data flow in original MapReduce

In the original MapReduce model, data is transferred between map tasks and reduce tasks in the way that can be described as “pull”. The reduce task is divided into three phases: shuffle, sort and reduce. When some of the map tasks are completed, the corresponding reduce tasks can fetch data in the shuffle phase. However, the reduce phase of the reduce task will not start until all the map tasks finish. Thus, the reduce tasks will occupy the assigned slots all the time

to wait for the completion of all map tasks, which is called the coupled relationship. Due to the coupled relationship between map and reduce tasks, it results in two problems: reduce slot hoarding [26] and underutilized network bandwidth. Meanwhile, as a fault-tolerant computing model, MapReduce introduces replicated distributed storage, which leads to inefficient storage in the data pulling scene (Fig. 1).

### 1.1 Reduce Slot Hoarding Problem

In the MapReduce model, it usually begins to schedule reduce tasks when a certain amount of map tasks are completed. If a big job is submitted, reduce tasks of the job will occupy all the assigned slots until all the map tasks finish. Consequently, when another job is submitted at this moment, it will not get corresponding reduce slots even after all the map tasks finish. Therefore, the later one will starve until the big job is completed. This is called the reduce slot hoarding problem, which will seriously reduce the execution efficiency of jobs, especially for small ones.

Figure 2 shows an example of reduce slot hoarding problem. Job1 and job2 are submitted at the same time while job3 is submitted a little later. The reduce task of job1 start when a certain amount of the map task finished, but no reduce tasks can be finished before all map tasks finished, so the reduce slots will be occupied by job1 for a long time. In this case, the reduce task of job3 can't execute when its map tasks are finished since job1 occupies all the reduce slots. Job3 will starve until job1 releases the reduce slots.

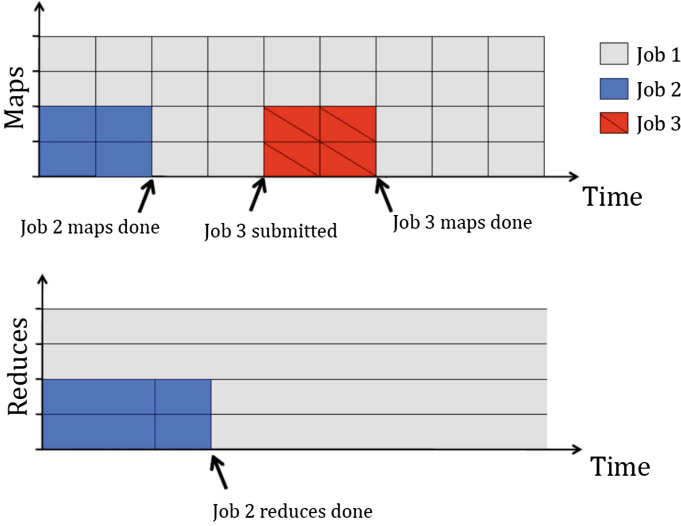


Fig. 2. Reduce slot hoarding problem

One solution to this problem is to delay the reduce tasks. In [26], the authors put forward a solution that starts reduce tasks after the completion of map tasks. However, their tests show that it will decrease the whole throughput. We consider that if reduce tasks start early, partially overlapped with map tasks, they will get a part of the data from map tasks, which can save the time of data transfer and the total completion time.

## 1.2 Underutilized Network Bandwidth Problem

For a MapReduce job, the network load will mainly concentrate in reduce tasks. In the reduce task, both shuffle and reduce phase need large network bandwidth since shuffle phase should pull data from other nodes, and reduce phase should write the result data into distributed file system. However, map tasks almost don't need large network bandwidth, because MapReduce has done some data locality optimization [7, 20, 21] when scheduling to make sure map tasks execute on the node where the needed data is stored on. So a map task can mostly read data from local disks.

There are some capabilities to balance network load and optimize network bandwidth in the original version of MapReduce, such as scheduling reduce tasks in advance. When the completion of map tasks reaches a certain ratio (default 5%), the reduce task will start so that they can run in parallel with maps. However, it can only alleviate this problem rather than resolve it. When some of the early-scheduled reduce tasks, whose desired intermediate data is ready, are getting partitions through shuffle, they cannot use much bandwidth in the map task. Conversely, they will occupy the reduce slots all the time. Besides, limited by the total slots, not all the reduce tasks can be scheduled. Therefore, these reduce tasks cannot work with map tasks simultaneously, and the network bandwidth will still be underutilized.

## 1.3 Inefficient Storage Problem

The MapReduce model is designed to run on big data sets with commercial hardware. In original version of MapReduce, the result is written into distributed file system directly. It will need many disks to store the data when the result is huge, which is costly especially when the system has one or more replications. This is the inefficient storage problem.

Compressing the result of MapReduce is a good way to reduce the data that is needed to be stored on the disk. Many compression modules have been proposed to address the inefficient storage problem. Apache Hadoop, an open-source implementation of MapReduce, can also support compressing the data now. Hadoop allows users to choose one compression algorithm. Once chosen, all the data will be compressed using this algorithm. BlobSeer [18, 19], a typical compression module that has achieved a great success on distributed file system, is a transparent compression module using prediction to determine whether to compress the data or not. First, BlobSeer samples part of the data to predict the compression ratio of total data block, then they judge if compressing the

data is beneficial to the system. This module only compresses the data when they think that compression is beneficial to the system. BlobSeer can save about 40 % space comparing with storing the data directly. However, these two modules don't take CPU rate and memory usage into consideration. As we all know that compression can cost too much CPU and memory resource, these modules may have a bad effect on other jobs if CPU is overload or memory is exhausted.

In this paper, we propose an uncoupled MapReduce model to address the above three problems, which can improve the system throughput and overall resource utilization. The rest of this paper is organized as follows. In Sect. 2, we present the uncoupled MapReduce model. Section 3 describes the architecture and implementation of this model. Section 4 offers the evaluation about our model and its application effects. Section 5 lists some related work. At last, Sect. 6 concludes the paper.

## 2 The Uncoupled MapReduce Model

An uncoupled MapReduce model with intermediate data transfer is designed to address the three problems, meanwhile improving the job execution efficiency and system throughput. Figure 3 shows the data flow in this model. The data transfer is completed during the map task in the uncoupled version of MapReduce, instead of during the reduce task as the original version does. It needs to meet three conditions as follows:

- To address the reduce slot hoarding problem thoroughly, reduce tasks should be scheduled after all map tasks are completed. Reduce tasks will not occupy the slots, and they can read the data to process locally once launched, which improves job execution efficiency and system throughput.
- To address the underutilized network bandwidth problem, the data transfer process of maps' result should be completed in the map task. When reduce tasks start, they will read data directly from local disks. Therefore, the network load will not concentrate in the reduce task and the network bandwidth in the map task can be used fully.
- To address the inefficient storage problem, the result data of MapReduce jobs should be compressed before stored in file system. Compression should be done transparently and automatically. Considering compression jobs may cause overload of CPU, a specific hardware is needed to help reducing the workload of CPU.

However, there is a conflict between the first two conditions. Map tasks need to transfer data to reduce tasks, but reduce tasks do not run until all the map tasks are completed. Therefore, some trade-offs are necessary to find out when and where reduce tasks run. The inefficient storage problem is more independent relatively to the first two conditions. In order to resolve these problems, we introduce the following four techniques.

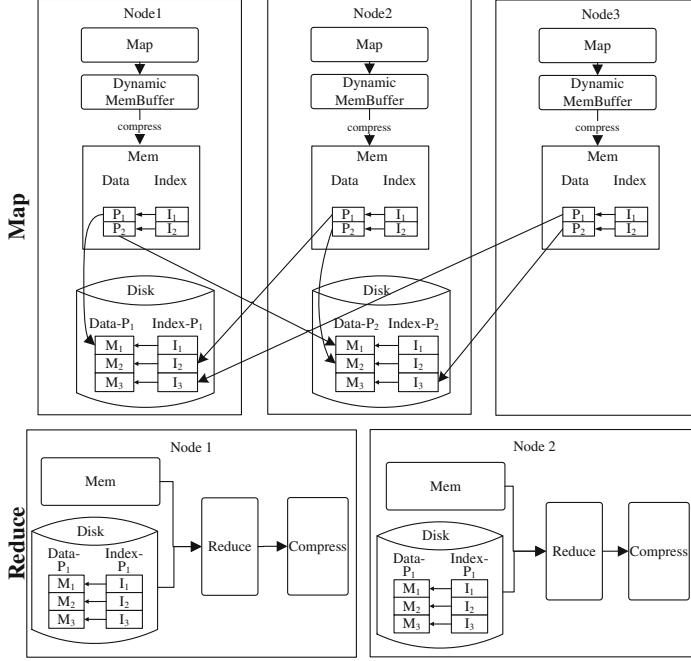


Fig. 3. The data flow in uncoupled MapReduce

- **Weighted mapping** [23]. This technique creates a mapping relationship between reduce tasks and nodes. Through the mapping relationship, the map task can find out the node where the corresponding reduce tasks will run, and they can transfer data to these nodes. In a heterogeneous cluster, the computing capability of each node is not identical. In consideration of this fact, each node has its own weight. The node with higher weight will get more tasks. This technique can guarantee balance and consistency of task assignment.
- **Balance.** The balance means that the node with greater weight will be assigned more reduce tasks. A linear relationship exists between the number of assigned tasks and the node's weight. We assume that there are  $n$  nodes with weight  $w_i (1 \leq i \leq n)$  in a cluster. Normalize the weight and get the normalization value  $w'_i$  from  $w_i$ , as shown in Eq. 1.

$$w'_i = \frac{w_i}{\sum_{i=1}^n w_i} \quad (1)$$

The variable  $M$  stands for the total number of reduce tasks, and  $M_i$  stands for the number of reduce tasks assigned to node  $i$ . Equation 2 shows the result of task assignment.

$$M_i = w'_i M = \frac{w_i}{\sum_{i=1}^n w_i} M \quad (2)$$

The two equations above can ensure the balance of task assignment.

- **Consistency.** The consistency means that the fixed mapping relationship between reduce tasks and nodes should be guaranteed and cannot be changed once decided. We assign reduce task  $R_j (1 \leq j \leq M)$  with weight  $w_{R_j}$ . The relationship is expressed in Eq. 3.

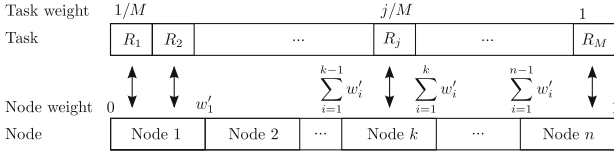
$$w_{R_j} = \frac{j}{M} \quad (3)$$

Reduce task  $R_j$  with weight  $w_{R_j}$  will be mapped to node  $k (1 \leq k \leq n)$  if they follow the relationship in Eq. 4.

$$\sum_{i=1}^{k-1} w'_i < w_{R_j} \leq \sum_{i=1}^k w'_i \quad (4)$$

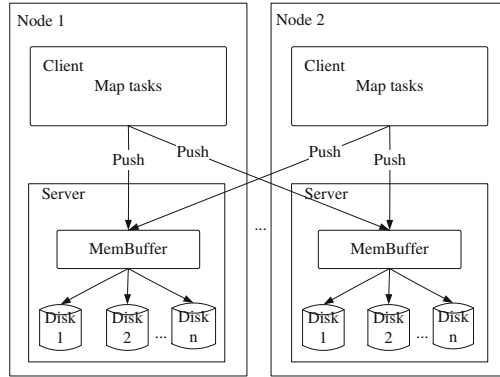
The mapping relationship is shown in Fig. 4. The two kinds of weights will be normalized into the same range, and then we can establish the mapping relationship between reduce tasks and nodes if they have the same weights after normalizing. Any module of a MapReduce application can inquire the relationship through Eq. 4.

Weighted mapping technique has no conflict with the scheduler in original MapReduce module. The original MapReduce module focuses on scheduling map tasks and reduce tasks between different MapReduce jobs while weighted mapping technique focus on the nodes and reduce tasks in one MapReduce job.



**Fig. 4.** The mapping relationship between reduce tasks and nodes

- **Data pushing.** In the uncoupled version of MapReduce, map tasks implement data transfer using data pushing. As shown in Fig. 3, map tasks will put intermediate data into dynamic buffers and partition them. Then they push data to the reduce tasks in corresponding mapped node from partition 1 to  $n$  in order, which is infeasible in the original version because where reduce tasks are responsible for getting data from map tasks. This idea is partially inspired by the pipelined MapReduce [11]. In the uncoupled version, a server is setup in each node which is responsible for receiving data from map tasks as shown in Fig. 5. When map tasks are generating intermediate output data, they will work as clients and push data to the servers. So there is no need to start reduce tasks before map tasks finish.
- **Partial data backup.** Each node has a server for receiving data from map tasks. If some servers go wrong, the data pushed by map tasks will be lost. It's costly to re-execute map tasks to get the lost data. The original version of



**Fig. 5.** The diagram of data pushing

MapReduce has the fault-tolerance module to avoid re-executing map tasks. However, uncoupled MapReduce changes the data transfer module leading to the useless of the original fault-tolerance module. The partial data backup technique in uncoupled MapReduce can resolve this problem. When the map tasks are pushing data, the data will also be backed up on local disks. When some servers go wrong, there is no need to re-execute the completed map tasks, because the data can be recovered by reduce tasks. A backup server is setup on each node which is responsible for managing the backup data. Reduce tasks will pull their own backup data by requesting each backup server. “Partial” here means if a partition is pushed from a map task to its own node, it will not be backed up. What’s more, it is compatible with the original fault tolerance mechanism in MapReduce.

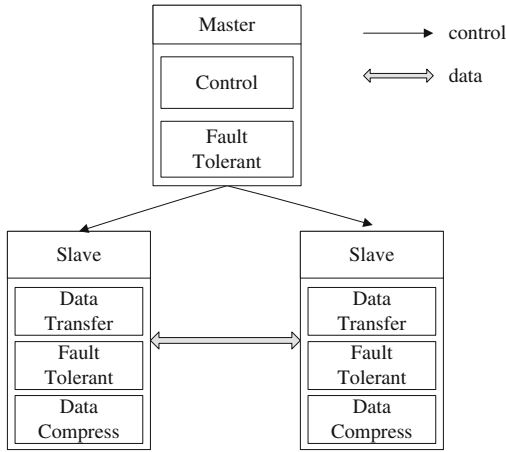
- **Data compressing.** In the uncoupled version of MapReduce, the result data of MapReduce jobs is stored into file system after compression which can reduce the data amount dramatically. CPU may be overload since the compression jobs need a lot of computing. In this case, other executing jobs may be seriously affected due to CPU resources exhausted. In data compressing technique, we choose the most appropriate compression algorithm to do compression jobs according to the CPU workload, the data property and so on. The technique will transfer compression jobs to a specific hardware if we think the workload of CPU is too heavy. All the job concerning compression include algorithm selecting and compressing are done transparently, and automatically to the user.

This model uncouples the dependency relationship between maps and reduces in MapReduce, replaces the shuffle phase in the original version and compresses the result data of MapReduce jobs. Through the four techniques mentioned above, it can make sure that all the reduce tasks can read data from their local disks when the map tasks are completed and all result data is stored after compressed. Therefore, reduce tasks will not occupy the reduce slots to wait for the completion of map tasks. It can also satisfy the needs of slots from other jobs.

This model can make full use of the network bandwidth in map and reduce tasks, balance the network load and improve the efficiency of storage.

### 3 Architecture and Implementation

Considering the original architecture of MapReduce and the requirements of four techniques mentioned above, the architecture of uncoupled MapReduce is designed. It includes four kinds of modules: master control module, data transfer module, fault tolerance module, and data compress module. We have implemented the architecture in Hadoop, and integrated different modules in the master and slave nodes. The uncoupled MapReduce architecture and its implementation in Hadoop are presented in Fig. 6.



**Fig. 6.** The uncoupled MapReduce architecture and its implementation in Hadoop

#### 3.1 Master Control Module

The master control module lies in the master node, responsible for monitoring and scheduling tasks, and coordinating other modules. This module should run all through the MapReduce job. Its functionalities are as follows.

- Create the mapping relationship between reduce tasks and nodes to make sure that reduce tasks will be executed on the node where data stored.
- Convey the mapping relationship information to the data transfer module and the fault tolerance module.
- Schedule tasks and make sure that reduce tasks will not start until all the map tasks are completed. When scheduling reduce tasks, make sure all the data is transferred to the specific node.

- Ensure balance and consistency of the mapping relationship, configure the computing capability of every node and make each node getting proper tasks according to its computing capability.
- Coordinate the data transfer module and the fault tolerance module. Normally, this module controls the data transfer module to complete the task of data pushing. If there is something wrong with data pushing, this module will notify the fault tolerance module to recover the missing data.

### 3.2 Data Transfer Module

The data transfer module lies in all the slave nodes, responsible for processing, storing, and pushing data. The module creates a server responsible for receiving and managing the intermediate data for every slave nodes since the data should be pushed to the node where the corresponding reduce tasks run according to the mapping relationship before reduce tasks begin in uncoupled version of MapReduce. The functionalities of this module are as follows.

- Create a data transfer server in each slave node.
- Get the output data from the map tasks.
- Do some preprocessing on the data.
- Get the mapping relationship.
- Work as a client to push the data of each partition to their corresponding reduce tasks.
- The data transfer server in this module is responsible for receiving and managing data.

### 3.3 Fault Tolerance Module

The fault tolerance module lies in all the nodes, responsible for processing exceptions caused by system crash, power outage and so on. As our model changes the intermediate data transfer mode, we must make some supplements to the original fault tolerance mechanism. In this module, the partial data backup technique is introduced following specific rules. The functionalities of this module are as follows.

- Backup each partition that map tasks will push to other nodes in the local disk.
- Assign the mapped reduce tasks to other nodes when a node failed.
- Offer the backup data to reduce tasks through backup servers, if the backup data has been made successfully. A reduce task checks whether the node is the mapped one through weighted mapping mechanism. If it is not the mapped one, the reduce task will pull its own backup data by requesting to other backup servers.

The fault tolerance module backups the partitions and the data transfer module pushes the partition to the node where reduce tasks will run. If one node

failed, the module will assign the mapped reduce tasks to other nodes and the transfer module can push the intermediate data to the new node without executing map tasks again. The fault tolerance module can not handle any exceptions, it's helpless if the system run into a catastrophic failure.

### 3.4 Data Compress Module

The data compress module lies in all the slave nodes, responsible for predicting, deciding, and compressing. This module will predict the CPU workload to decide whether compress the data in CPU or the specific hardware, and also predict the property of output data to decide which algorithm is the most appropriate compression algorithm. After predicting and deciding, the module will do the data compression using the decided algorithm. The functionalities of this module are as follows.

- Predict the CPU workload if the compressing job will be done in CPU.
- Predict the property of the output data.
- Decide whether the data compression job should be done in CPU or the specific hardware.
- Decide which is the most appropriate compression algorithm under this condition.
- Compress the data according to the decided strategy.

We find that different compression algorithm has its advantages and disadvantages after research. The data compress module implements five kinds of compression algorithms including quicklz [5], elzs, exar, snappy [6], and zlib, for different kinds of data under different conditions. First, the module predicts the CPU workload and the property of the output data. Then the module chooses the best compression algorithm. At last, do the compressing job according to the decided strategy.

The four modules remove the coupled relationship of map tasks and reduce tasks in original MapReduce, and compress the result of MapReduce jobs. It guarantees that the uncoupled MapReduce can resolve the reduce slot hoarding, underutilized network bandwidth, and inefficient storage problems.

## 4 Evaluations

We evaluate the model and its application effects using a micro-benchmark and a real-world example. In the micro-benchmark, we use a cluster to compare the job execution time in the uncoupled version of Hadoop with that in the original version. Our work has also been applied in a production environment of Baidu, which gives a comprehensive evaluation on the uncoupled MapReduce model.

- **Definition 1:** Wall time is the total time span from the moment a job is submitted to the moment it is completed.

- **Definition 2:** Throughput  $T$  is the number of jobs finished in a unit time interval. Suppose that  $N$  jobs are completed in a time interval  $t$ , we will get:

$$T = \frac{N}{t} \quad (5)$$

Our tests use the same workload in both the original version and the uncoupled version, so:

$$N_{original} = N_{uncoupled} \quad (6)$$

Suppose all the jobs are submitted nearly simultaneously.  $t_{1i}$  is the wall time of job  $i$  in the original version, and  $t_{2i}$  is that in the uncoupled version. Use  $t_{original} = \max(\{t_{1i}\})$  and  $t_{uncoupled} = \max(\{t_{2i}\})$  ( $1 \leq i \leq N$ ) to represent the total wall time in the original version and the uncoupled version. Then we define the throughput increment rate as  $I$ :

$$I = \frac{T_{uncoupled} - T_{original}}{T_{original}} = \frac{\frac{N_{uncoupled}}{t_{uncoupled}} - \frac{N_{original}}{t_{original}}}{\frac{N_{original}}{t_{original}}} = \frac{t_{original}}{t_{uncoupled}} - 1 \quad (7)$$

Then we define the rate of total wall time reduction as  $r$ , and the rate of job  $i$ 's wall time reduction as  $r_i$ :

$$r = \frac{t_{original} - t_{uncoupled}}{t_{original}} \quad (8)$$

$$r_i = \frac{t_{1i} - t_{2i}}{t_{1i}} \quad (9)$$

The impact factor of job  $i$ ,  $\lambda_i$ , represents the proportion of job  $i$ 's wall time in all the jobs. We can get  $\lambda_i$  from Eq. 10:

$$\lambda_i = \frac{t_{1i}}{\sum_{i=1}^N t_{1i}} \quad (10)$$

The weighted wall time acceleration  $P$  represents the sum of wall time reduction rate with impact factors. It can be deduced from Eq. 11:

$$P = \sum_{i=1}^N \lambda_i r_i = \sum_{i=1}^N \frac{t_{1i}}{\sum_{i=1}^N t_{1i}} \frac{t_{1i} - t_{2i}}{t_{1i}} = \frac{\sum_{i=1}^N (t_{1i} - t_{2i})}{\sum_{i=1}^N t_{1i}} = 1 - \frac{\sum_{i=1}^N t_{2i}}{\sum_{i=1}^N t_{1i}} \quad (11)$$

The throughput increment rate and total wall time reduction rate are the metrics reflecting the overall performance. The weighted wall time acceleration is the metric reflecting the cumulative performance of each job.

#### 4.1 Micro-Benchmark

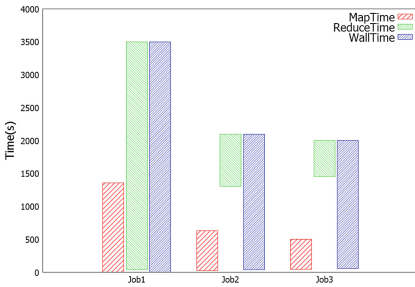
The micro-benchmark is performed in a cluster with 6 nodes. The operating system is CentOS 6.1  $\times$  86\_64, and the Hadoop version is 0.19. We use gridmix [3] applications for our test. Gridmix is a set of benchmark programs for Hadoop which contains several kinds of jobs. The micro-benchmark includes 3 jobs as shown in Table 1. job1 was submitted first, then job2, and job3 at last. The interval between two adjacent jobs is 30 seconds. We divide the micro-benchmark into three parts as follows.

**Table 1.** The workload of the micro-benchmark

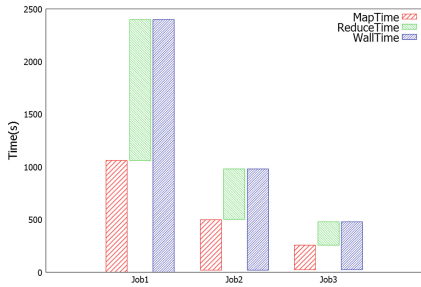
Job name	Input size	Maps	Reduces
job1	50 GB	400	200
job2	12.5 GB	100	50
job3	6.25 GB	50	25

**Slot Allocation.** In original MapReduce, reduce tasks of job1 occupy all the reduce slot since only job1 in the system at that time in Fig. 7. Reduce tasks of job2 and job3 can not start until one or more reduce tasks of job1 finished. As job2 and job3 are small jobs comparing to job1, when map tasks of job2 and job3 finished, no reduce tasks of job1 finished and all reduce slots were stilled occupied by job1. So it caused the reduce slot hoarding problem. As we can see in Fig. 8, the uncoupled MapReduce resolves the reduce slot hoarding problem by taking the strategy that all reduce tasks must be executed after all map tasks finished.

**Network Bandwidth.** Although the coupled MapReduce starts reduce tasks before all map tasks finished, most of reduce tasks have to wait due to the limitation of reduce slots. So the network load is low since only reduce tasks need large network bandwidth. As we can see in Fig. 9, network bandwidth is

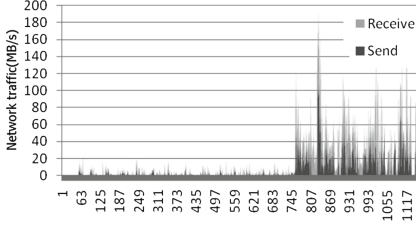


**Fig. 7.** Reduce slot in original MapReduce

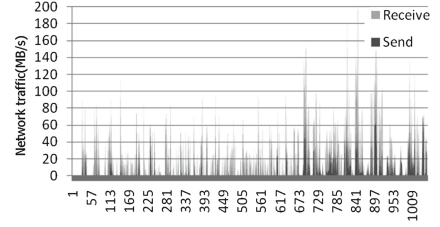


**Fig. 8.** Reduce slot in uncoupled MapReduce

underutilized for that the network load in reduce tasks is much higher than that in map tasks. In map tasks, the uncoupled MapReduce transfers the intermediate data to the node where the mapping reduce tasks will run on. So the uncoupled MapReduce makes full use of network bandwidth as the map tasks transfer the intermediate data and the reduce tasks write the result and replications into file system. Figure 10 shows the network traffic in uncoupled MapReduce.



**Fig. 9.** Network traffic in original MapReduce



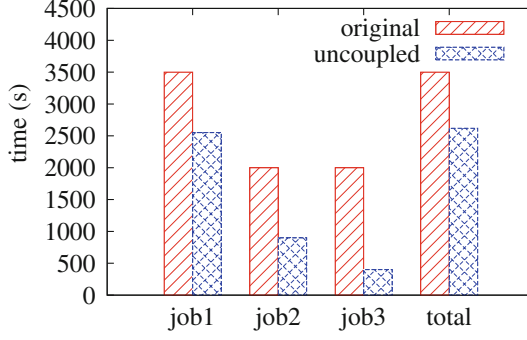
**Fig. 10.** Network traffic in uncoupled MapReduce

**System Performance.** Figure 11 shows the execution time of each job and the total completion time of all the workloads. In our test, the total time of the original version is 3500s, and that of the uncoupled version is 2620s. There is no reduce slot hoarding in the uncoupled version. The throughput increment is 34 % ( $I$ ) through Eq. 7, The total wall time is reduced by 25 % ( $r$ ) through Eq. 8, and the weighted wall time acceleration reaches 48 % ( $P$ ) through Eq. 11. The test shows that our model can balance the network load properly and improve the system throughput. The uncoupled MapReduce makes full use of disk storage since it can reduce 70 % data volume comparing to the original MapReduce according to our benchmark.

## 4.2 Real-World Example

The uncoupled MapReduce implementation based on Hadoop has been deployed in a production environment of Baidu supporting some business applications. The real-world example provides strong evidence on the effects of this work.

Baidu is the biggest search engine company in China. It has tens of clusters performing Hadoop jobs for many web data processing applications, and generates more than 3 PB data volume per day [17]. Although the clusters can deal with hundreds of jobs everyday, they still meet with some problems. For example, the CPU and network bandwidth utilization rates are not high in spite of full workload. The uncoupled version of Hadoop has been deployed in a server cluster with 70 nodes, which is one of the shared Hadoop platforms for many departments. The resource scale reaches about 560 cores, 1,120 GB memory, and 770 TB storage. The operating system is Red Hat Enterprise Linux AS release 4, and the Hadoop version is 0.19. Many kinds of jobs run in this real environment,



**Fig. 11.** Comparison of execution time in the micro-benchmark

such as log analysis, inverted index, web ranking, etc. Four kinds of jobs are used in our test, including CPU-intensive ones and I/O-intensive ones. The job information is shown in Table 2.

**Table 2.** The workload of the real-world example

Job name	Input size	Maps	Reduces
job1	3.4 TB	14700	1600
job2	3.2 TB	14400	1600
job3	353 GB	1700	800
job4	343 GB	1600	800

Figure 12 shows the execution time of each job and the total completion time of all the workload. In our test, the total time of the original version is 272 min, and that of the uncoupled version is 210 min. So we can get the rate of throughput increment:

$$I = \frac{t_{original}}{t_{uncoupled}} - 1 = 29.5\% \quad (12)$$

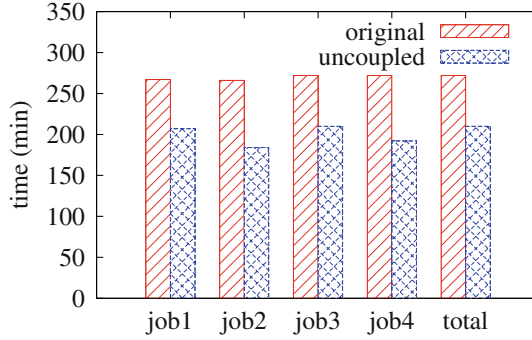
The rate of total wall time reduction:

$$r = \frac{t_{original} - t_{uncoupled}}{t_{original}} = 22.8\% \quad (13)$$

And the weighted wall time acceleration:

$$P = 1 - \frac{\sum_{i=1}^4 t_{2i}}{\sum_{i=1}^4 t_{1i}} = 26.3\% \quad (14)$$

In addition, job3 and job4 in the original version cannot get reduce slots when their map tasks are completed. They are delayed by 51 min and 63 min respectively because of reduce slot hoarding problem.



**Fig. 12.** Comparison of execution time in the real-world example

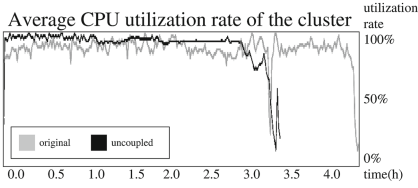
Figures 13, 14, 15, and 16 show the comparison of the original version and the uncoupled version in resources utilization. The workload is the above examples. In the uncoupled version, it took 3.5h to complete all the jobs, and the last map task finished in about 2.5h. While in the original version, it took 4.5h to complete all the jobs, and the last map task finished in about 3.3h.

Figure 13 illustrates the average CPU utilization rate of the cluster. The CPU utilization rate in the uncoupled version is higher than that in the original version. The reason can be explained from the map and reduce tasks respectively. In the map tasks, intermediate data is pushed and received by the corresponding servers, which results in more data preprocessing and transferring operations. In the reduce tasks, data is processed faster due to data locality.

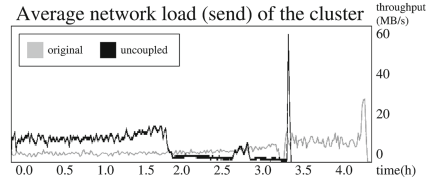
Figure 14 shows the network load (send throughput) of the cluster. The uncoupled version uses more network bandwidth in the map tasks to transfer data. Each map will push all of its output data to its corresponding reduce tasks, instead of data being pulled partially by reduce tasks in the original version. There is little network bandwidth in reduce phase of the uncoupled version, because the reduce tasks will read data from their local disks. The peak in the end shows that HDFS uses more bandwidth to backup the output data of reduce tasks including the results and completion information of jobs.

Figures 15 and 16 show the disk I/O (write and read throughput) of the cluster. In the uncoupled version, more disk I/O workload is involved in the map task than that in the reduce task, because the map task will read data from disks and push them to the mapped nodes where the servers receive and write data. While in the original version, the average throughput of disks is relatively low. Shuffle is an ineffective and complicated phase, which has a significant impact on the execution time and system throughput.

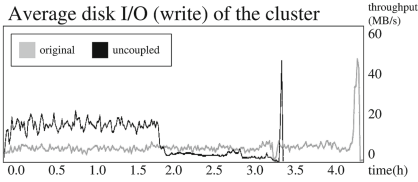
Overall, the uncoupled version of Hadoop increases the resource utilization rates, and avoids the waste of network and disk bandwidth.



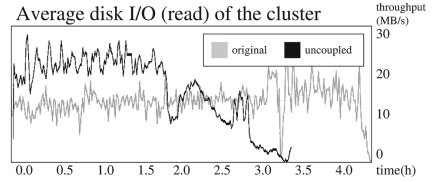
**Fig. 13.** Comparison of the average CPU utilization rate of the cluster



**Fig. 14.** Comparison of the average network load (send) of the cluster



**Fig. 15.** Comparison of the average disk I/O (write) of the cluster



**Fig. 16.** Comparison of the average disk I/O (read) of the cluster

## 5 Related Work

There are a lot of researches on MapReduce. This section will introduce some related work. They aim at improving the system throughput and making full use of system resources.

- **The Pipeline In Hop** [11]. Hadoop Online Prototype can push the map tasks' output to reduce tasks so that there is no need to write the output as intermediate data into local disks. Then reduce tasks can go into the sorting and reducing phases without waiting for the finish of map tasks. Therefore, map and reduce tasks can work simultaneously, which can improve job performance and efficiency, and balance the network load. However, since no intermediate data stored in the local disk, the fault tolerance mechanism can not work, resulting in high cost for error recovering. In uncoupled MapReduce, we back up the intermediate data in local disk when pushing it to reduce tasks. So there is no need to re-execute map tasks when error happens in data transformation.
- **Copy-compute Splitting** [26]. This work is put forward by Facebook and includes two phases, copy and compute. The copy phase is I/O-intensive and needs to pull the map tasks' output from other nodes, while the compute phase is CPU-intensive. The reduce task consists of two kinds of tasks, the copy task and compute task, which could run simultaneously. The copy task notifies the compute task to work when it gets all the output from the map tasks. This technique can alleviate the reduce slot hoarding problem. However, it needs to set two variables, maxComputing and maxReducers. MaxComputing is the same as the reduce slot number in the original Hadoop. Limited by the number

of copy slots, it will also cause Copy Slot Hoarding problem when big jobs occupy all the copy slots.

- **Dynamic Weight Assignment** [24]. In this model, map and reduce tasks share the total number of slots. Each job can get its own slots according to its weight. At the beginning of a job, the map tasks get big weights and reduce tasks get small ones. Gradually, the map tasks get fewer slots while the reduce tasks get more slots. But the big job can still get a small number of reduce slots which can only alleviate reduce slot hoarding problem. Besides, it will also have an effect on the job priority. The job with high priority may not be scheduled in time and reduce tasks may be delayed, which can decrease the network bandwidth.
- **Weighted Shuffle Scheduling** [10]. In this model, the authors think that the shuffle phase needs to get a lot of data, which will occupy a large amount of the job operating time. Through the analysis of jobs in Facebook, it shows that shuffle can take about 33 % of the operating time in reduce tasks. Therefore, they think that shuffle plays an important role on the performance of the system. This technique can raise the efficiency of shuffle. They assign each flow (a flow is a socket connection between a map task and a reduce task) with a weight. Then the network bandwidth can be under control through this weight. However, it cannot address the reduce slot hoarding or unbalanced network load problem. The network bandwidth is still concentrated in reduce tasks.
- **Spark** [27]. MapReduce is built around an acyclic model that is not suitable for iterative algorithm and interactive data analysis. Spark is a new cluster computing framework supporting this kinds of work while retaining the scalability and fault tolerance of MapReduce. A resilient distributed dataset(RDD) is the main abstraction in Spark representing a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. This model can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39GB dataset with sub-second response time. However, it is not suitable for fine-grained and asynchronous update operations since Spark is a coarse-grained data parallel computing model.
- **MapReduce Energy Efficiency** [9]. Since data becomes larger and larger, the modern data center puts more and more attention on energy efficiency. Compression can reduce data dramatically and reduces the usage of disks. However, it may not improve energy efficiency since the compressing data needs additional computing. Reference [9] come up with a module to help improve the energy efficiency by 35–60% for the data read frequently on MapReduce. The module predicts the compression ratio and the visited frequency of the data to decide whether to compress. When the compression ratio is smaller than 0.2, the data will be compressed. When the compression ratio is between 0.2 and 0.4, the data will be compressed if the data will be frequently read. Under other conditions, the data will be stored and transferred directly. This model is useful in the practical application. Reference [14] also comes up with a good model to improve energy efficiency, it proves

that compressing and decompressing take up 7–11% time of map tasks while 37.9–41.2% time of reduce tasks since compressing and decompressing can increase the burden of CPU. This module introduces a specific hardware to do compressing and decompressing jobs instead of CPU, which has achieved a good performance. However, the efficient of these two systems is still low since they only focus on energy efficiency when compressing without concerning the coupled relationship between maps and reduces.

- **BlobSeer** [18, 19]. BlobSeer is a transparent compression module on BlobSeer File System(BSFS) aiming at reducing the data amount and improve the throughput of the system. This module can be implemented by Hadoop and used by MapReduce. Before compressing the total data, it compresses a small part of data using sampling method to predict the compression ratio of total data and determines whether compressing is useful. Users can choose compression algorithms according their needs, but only the chosen algorithm can be used when the system is running. Experiments has been done using algorithm LZO [4] and BZIP2 [2], and result shows that BlobSeer can reduce 40% data. However, it will cause CPU overload problem that have a bad effect on other jobs. In uncoupled MapReduce, we solve this problem using a specific hardware.

## 6 Conclusion and Future Work

Although there are a variety of optimizations to improve the localization rate of data in the MapReduce model, data transfer is inevitable. Reduce tasks need to pull intermediate data from map tasks, which will decrease the execution efficiency of jobs. In [10], it shows that shuffle can take about 33% of the operating time in reduce tasks. Meanwhile, data transfer efficiency is very low, which is a system bottleneck [25]. Since the MapReduce model is designed to be fault-tolerant, it's costly to store the replicated file especially when the result is huge. Impacted by these facts, the original MapReduce model results in three problems: reduce slot hoarding, underutilized network bandwidth, and inefficient storage.

In this paper, we propose an uncoupled MapReduce model to resolve these problems with the aim of improving the resource utilization and system throughput. Four techniques, including weighted mapping, data pushing, partial data backup and data compression, are implemented. This work has been practiced in Baidu, the biggest search engine company in China. In a real-world application, the test shows that the throughput can be increased by 29.5%, the total wall time is reduced by 22.8%, the weighted wall time acceleration reaches 26.3%, and reduce the result data storage by 70% compared with the original version of Hadoop. The resource utilization rates of CPU, network and disk are also increased.

Two improvements are planned: (1) Design a kind of scheduler for this model which can use the cluster resources more reasonably. (2) Monitor workloads and resources dynamically, instead of setting constant slots and weights. Hopefully, our paper would assist in the study of heterogeneous resources utilization.

**Acknowledgment.** We would like to thank Ruijian Wang, Maosen Sun, Chen Feng, Fan Liang, and Dixin Tang from Institute of Computing Technology, Chinese Academy of Sciences for the valuable discussions. We also thank Chuan Xu, Linjiang Lian, and Meng Wang from Baidu for their assistance and support. This research is supported in part by the Hi-Tech Research and Development (863) Program of China (Grant No. 2013AA01A213, 2011AA01A203).

## References

1. Apache hadoop. <http://hadoop.apache.org/>
2. Bzip2 compression. <http://www.bzip.org/>
3. Gridmix. <http://hadoop.apache.org/docs/stable/gridmix.html>
4. Lempel-ziv-oberhumer(lzo) compression. <http://www.oberhumer.com/opensource/lzo>
5. Quicklz. <http://www.quicklz.com/>
6. Snappy. <http://code.google.com/p/snappy/>
7. Cao, P., Felten, E.W., Karlin, A.R., Li, K.: A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.* **23**(1), 188–197 (1995)
8. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* **26**(2), 4:1–4:26 (2008)
9. Chen, Y., Ganapathi, A., Katz, R.H.: To compress or not to compress - compute vs. IO tradeoffs for MapReduce energy efficiency. In: *Proceedings of the First ACM SIGCOMM Workshop on Green Networking, Green Networking 2010*, pp. 23–28. ACM, New York (2010)
10. Chowdhury, M., Zaharia, M., Ma, J., Jordan, M.I., Stoica, I.: Managing data transfers in computer clusters with orchestra. In: *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM 2011*, pp. 98–109 (2011)
11. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce online. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI 2010*, pp. 1–15 (2010)
12. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation, OSDI 2004*, pp. 137–150 (2004)
13. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. *SIGOPS Oper. Syst. Rev.* **37**(5), 29–43 (2003)
14. Gu, X., Hou, R., Zhang, K., Zhang, L., Wang, W.: Application-driven energy-efficient architecture explorations for big data. In: *Proceedings of the 1st Workshop on Architectures and Systems for Big Data, ASBD 2011*, pp. 34–40. ACM, New York (2011)
15. Gu, Y., Grossman, R.L.: Sector and sphere: Towards simplified storage and processing of large scale distributed data (2008). [arXiv:0809.1181](https://arxiv.org/abs/0809.1181)
16. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys 2007*, pp. 59–72 (2007)
17. Ma, R.: Introduction to part of the baidu's distributed systems. <http://www.slideshare.net/cydu/sacc2010-5102684>

18. Nicolae, B., Moise, D., Antoniu, G., Bouge, L., Dorier, M.: Blobseer: Bringing high throughput under heavy concurrency to hadoop map-reduce applications. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–11 (2010)
19. Nicolae, B.: High throughput data-compression for cloud storage. In: Hameurlain, A., Morvan, F., Tjoa, A.M. (eds.) *Globe 2010*. LNCS, vol. 6265, pp. 1–12. Springer, Heidelberg (2010)
20. Padmanabhan, V.N., Mogul, J.C.: Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.* **26**(3), 22–36 (1996)
21. Seo, S., Jang, I., Woo, K., Kim, I., Kim, J.S., Maeng, S.: Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In: *IEEE International Conference on Cluster Computing and Workshops, CLUSTER 2009*, pp. 1–8 (2009)
22. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2010)
23. Wang, X.W., Zhang, J., Liao, H.M., Zha, L.: Dynamic split model of resource utilization in mapreduce. In: *Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds, DataCloud-SC 2011*, pp. 21–30. ACM, New York (2011)
24. Wang, X., Zhang, J., Liao, H., Zha, L.: Dynamic split model of resource utilization in MapReduce. In: *Proceedings of the 2nd International Workshop on Data Intensive Computing in the Clouds, DataCloud-SC 2011*, pp. 21–30 (2011)
25. Wang, Y., Que, X., Yu, W., Goldenberg, D., Sehgal, D.: Hadoop acceleration through network levitated merge. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011*, pp. 1–10 (2011)
26. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user MapReduce clusters. Technical report UCB/EECS-2009-55, EECS Department, University of California, Berkeley (2009)
27. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2010* (2010)

<http://www.springer.com/978-3-662-46334-5>

Transactions on Large-Scale Data- and

Knowledge-Centered Systems XVII

Selected Papers from DaWaK 2013

Hameurlain, A.; Küng, J.; Wagner, R.; Bellatreche, L.;

Mohania, M. (Eds.)

2015, XIII, 129 p. 57 illus., Softcover

ISBN: 978-3-662-46334-5