ComMapReduce: An Improvement of MapReduce with Lightweight Communication Mechanisms

Linlin Ding[†], Guoren Wang[†*], Junchang Xin[†], Xiaoyang Wang[†], Shan Huang[†], Rui Zhang[‡]

[†]*College of Information Science and Engineering, Northeastern University, China.*
E-mail[*]: wanggr@mail.neu.edu.cn
[‡]*Department of Computing and Information Systems, The University of Melbourne, Australia.*
E-mail: rui@csse.unimelb.edu.au

Abstract

As a parallel programming framework, MapReduce can process scalable and parallel applications with large scale datasets. The executions of Mappers and Reducers are independent of each other. There is no communication among Mappers, neither among Reducers. When the amount of final results are much smaller than the original data, it is a waste of time processing the unpromising intermediate data. We observe that this waste can be significantly reduced by simple communication mechanisms to enhance the performance of MapReduce. In this paper, we propose ComMapReduce, an efficient framework that extends and improves MapReduce for big data applications in the cloud. ComMapReduce can effectively obtain certain *shared information* with efficient lightweight communication mechanisms. Three basic communication strategies, Lazy, Eager and Hybrid, and two optimization communication strategies, Prepositive and Postpositive, are proposed to obtain the *shared information* and effectively process big data applications. We also illustrate the implementations of three typical applications with large scale datasets on ComMapReduce. Our extensive experiments demonstrate that ComMapReduce outperforms MapReduce in all metrics without affecting the existing characteristics of MapReduce.

Keywords:
MapReduce, Hadoop, Communication Mechanism

1. Introduction

The volume of data and information is growing at a remarkable speed in recent years. For example, Yahoo! [1] and Facebook [2] process over 120 terabytes and 80 terabytes of data everyday, respectively. Because of the rapid increase of large scale data, it is difficult for centralized algorithms to do analysis and computations on such large scale datasets. Therefore, there is an urgent need for distributed algorithms for data storage and processing. MapReduce [3] and its public available implementation, Hadoop[1], have emerged as the de facto standard programming framework for performing large scalable and parallel tasks on large scale datasets. This programming framework is scalable, fault tolerant, cost effective and easy to use. Non-experts can deploy their applications simply by implementing a Map function and a Reduce function, whereas the data management, task management, fault tolerance and recovery are provided transparently. The success of MapReduce and Hadoop has resulted in their deployment in many companies [4, 5, 6, 7] and research communities [8, 9, 10, 11, 12, 13, 14].

In MapReduce framework, after the Map tasks complete, there are usually large amount of intermediate data to be transferred from all Map nodes to all Reduce nodes in the shuffle phase. The shuffle phase needs to transfer the data from the local Map nodes to Reduce nodes through the network. Large amount of data to be transferred in the shuffle phase causes considerable overhead in the MapReduce process. Reducing the amount of intermediate data of Map tasks can lead to significant improvement in the efficiency of MapReduce.

In addition, in actual big data applications, the users are usually interested in obtaining a relatively small subset from the large scale dataset. That is to say, the amount of final results are much smaller than the original data. For example, a top-k query is naively processed by enumerating all the data records and choosing the best k. When k is much smaller than the original dataset, there will be some time waste in processing the unpromising data. In MapReduce framework, the implementations of Mappers and Reducers are completely independent of each other without communication among Mappers or Reducers. When processing this type of applications on MapReduce, there are large amount of unpromising intermediate data to be transferred in the shuffle phase. These unpromising intermediate data will be finally abandoned anyhow, leading to the waste of disk access, network bandwidth and CPU resources. If this type of applications can obtain certain *shared information* to filter and reduce the amount of intermediate

[1] http://hadoop.apache.org/

data, the amount of data transferred in the shuffle phase can be decreased and the waste of time processing the unpromising data can be mitigated. Therefore, without affecting the existing characteristics of MapReduce, if the *shared information* can be obtained by simple and efficient lightweight communication mechanisms, the performance of MapReduce can be improved.

In this paper, we propose a new framework named ComMapReduce to improve the MapReduce framework. ComMapRedcue introduces simple lightweight communication mechanisms to generate the *shared information* and deal with large scale datasets in the cloud. ComMapReduce adds a lightweight communication function to effectively filter the unpromising intermediate data of Map phase and the input of Reduce phase. ComMapReduce further extends MapReduce to enhance the efficiency of big data applications without scarifying the existing characteristics of MapReduce. In summary, the contributions of this paper are as follows:

- A framework with simple lightweight communication mechanisms, named ComMapReduce, is proposed. The Mappers and Reducers of ComMapReduce use the *shared information* to improve the framework performance without sacrificing the advantages of the original MapReduce framework.
- Three basic communication strategies, Lazy, Eager, Hybrid, and two optimization communication strategies, Prepositive, Postpositive, are proposed to obtain the *shared information* and effectively process big data applications.
- The implementations of three applications on ComMapReduce, *k*NN, skyline and join, are given to demonstrate the wide applicability of the ComMapReduce framework.
- Extensive experiments are conducted to evaluate our ComMapReduce framework using many synthetic datasets. The experimental results demonstrate that ComMapReduce outperforms the original MapReduce in all metrics.

This paper extends an earlier published conference paper [15] in several substantial aspects. First, we add more details about the background, motivation and the related work. Second, based on the proposed framework, we provide a universal programming interface of our ComMapReduce framework, which makes programming on our framework easier. Third, in Hybrid Communication Strategy, we propose an optimization criterion to identify the most suitable preassigned

period of time that enhances the performance of this communication strategy. Fourth, we present three typical applications of ComMapReduce, *k*NN, skyline and join, to demonstrate the wide applicability of our framework. Finally, we added more experiments to evaluate the performance of our proposed framework.

The rest of this paper is organized as follows: After analyzing the execution of a simple top-*k* query on MapReduce framework, we propose our ComMapReduce framework in Section 2. Three basic communication strategies and two optimization communication strategies are investigated in Section 3. Section 4 presents three applications of ComMapReduce, *k*NN, skyline and join queries. Section 5 reports the experimental results. Section 6 reviews the related work and the conclusions are given in Section 7.

2. ComMapReduce Framework

In this section, after analyzing MapReduce framework in depth in Section 2.1, we propose our ComMapReduce framework in Section 2.2. The availability and scalability of ComMapReduce are analyzed in Section 2.3.

2.1. MapReduce Analysis In-depth

MapReduce is a parallel programming framework processing the large scale datasets on clusters with numerous commodity machines. By automatically hiding the distributed low level techniques, such as automatic data management, transparent fault tolerance and recovery mechanisms, MapReduce provides a simple and efficient programming interface. An overview of the execution course of a MapReduce application is shown in Figure 1. When a MapReduce job is processed in the cluster, as the brain of the whole framework, the Master node schedules a number of parallel tasks to run on the Slave nodes. First, in Map phase, each Map task independently operates a non-overlapping split of the input file and calls the user-defined Map function to emit its intermediate $<$*key*,*value*$>$ tuples in parallel. Second, once a Map task completes, each Reduce task fetches all the particular intermediate data remotely. This course is called the shuffle phase in MapReduce. The shuffle phase incurs considerable performance overhead due to the transferring of large volumes of data occupying most disk access and network bandwidths of the cluster. Finally, the Reduce tasks process all the tuples with the same *key* and invoke the user-defined Reduce function to produce the final outputs written into the Distributed File System.

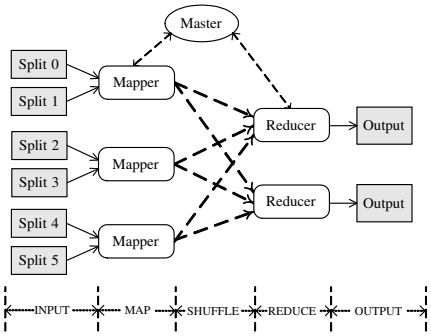


Figure 1: MapReduce Overview

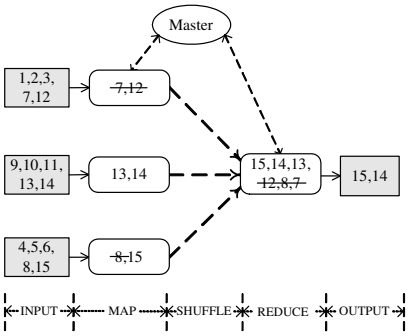


Figure 2: Top-*k* Query on MapReduce (*k*=2)

EXAMPLE 1. (Simple Top-k Query) The initial input dataset has 15 data from '1' to '15'. The top-2 query searches the two biggest data from '1' to '15'. Obviously, '14' and '15' are the final results.

Figure 2 briefly describes the top-*k* query of EXAMPLE 1 processing on MapReduce framework. There are three Mappers and one Reducer on the platform. In Map phase, the original fifteen data are partitioned into three splits as the input of three Mappers and then processed by the Mappers. Each Mapper reads the partition of its split and computes its local top-*k* query results by calling a user-defined Map function in parallel. For example, the first Mapper reads data '1', '2', '3', '7', '12' and generates '7' and '12' as its intermediate data. The intermediate data are held in the local disk. In the shuffle phase, the Reducer pulls all the intermediate data from each Map task, including disk reading at Map tasks nodes, the network traversal and disk writing at the Reduce task node. In Reduce phase, the Reducer gets the intermediate data of the three Mappers and then invokes a user-defined Reduce function to emit the final results of this top-2 query.

To decrease the data volume transferred in the shuffle phase, MapReduce uses an optimization, named Combiner, to perform a reduce operation of a Map task's intermediate data. While Combiner is a local data compression, it has no processing information of the other Mappers. It cannot obtain a global *shared information* to process unpromising data compression for further filtering in the shuffle phase. Therefore, Combiner cannot play an efficient role in the global data compression.

If we can use simple lightweight communication mechanisms to obtain certain *shared information* for pruning the superfluous data, the stress of network and disks can be released. Just as this simple example, before the intermediate data of three Mappers are written into their disks, if the Mappers can first receive data '13' as a *shared information* to prune their unpromising intermediate data with simple

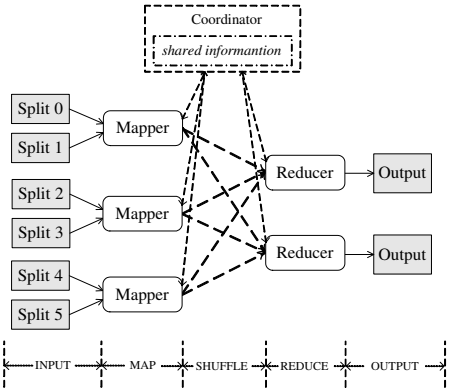


Figure 3: ComMapReduce Overview

```
public interface COORDINATOR<InfoType>
{
    InfoType sharedinformation;
    setup ();
    send ();
    receive ();
    find ();
}
```

Figure 4: Interface of ComMapReduce

communication mechanisms, the output of Mappers can be decreased drastically. The input of Reducer can be decreased from data '15', '14', '13', '12', '8', '7' to data '15', '14', '13'. Based on this idea, our ComMapReduce is proposed.

2.2. ComMapReduce Overview

For optimizing MapReduce framework, we propose ComMapReduce, an efficient lightweight communication framework extending MapReduce by generating the *shared information* to prune more unpromising data for big data applications.

Figure 3 shows the architecture of ComMapReduce framework. Our ComMap Reduce framework inherits the basic MapReduce framework and takes Hadoop Distributed File System (HDFS)[2], the open source implementation of MapReduce's GFS [16], to store the original input data and the final results of each application. In ComMapReduce framework, the Master node and the Slave nodes play the same roles as the original MapReduce, so the Master node is not shown in Figure 3 and the following figures for convenience. As discussed above, ComMapReduce can obtain the *shared information* to shorten the time latency of big data applications with simple lightweight communication mechanisms. In this paper, to avoid putting extra pressure on the Master node, a new node, named Coordinator node, is deployed on ComMapReduce to store and generate the *shared information*. The Coordinator node can communicate with the Mappers and Reducers with simple lightweight communication mechanisms. The Coordinator node can also receive and store some temporary variables, and then generates a message

[2]http://hadoop.apache.org/common/hdfs/

package with the *shared information* of applications. The message package with the *shared information* can be any format according to different applications, such as a data value, a array, a table and so on. Each Mapper obtains the *shared information* from the Coordinator node to prune its data inferior to the others. After filtering, the output of Mappers and the input of Reducers both decrease significantly. In summery, ComMapReduce is an efficient parallel programming framework with global *shared information* to filter the unpromising data. It cannot only process the one pass big data applications, but also the iterative big data analysis applications. Therefore, our ComMapReduce framework can be used in a wide application areas.

In Figure 3, when a client submits an application to the Master node, the Master node schedules several parallel tasks to process this application. In Map phase, the original inputs of the client are divided into several splits. The Mappers on clusters process the splits in parallel and generate their intermediate data. Then, each Mapper computes its local *shared information* according to the application and sends it to the Coordinator node. After that, the Coordinator node gains the most optimal one as the global *shared information* from the local *shared information* it receives according to the features of the application. Simultaneously, the Mappers receive the global *shared information* from the Coordinator node to filter their unpromising intermediate data to be transferred in the shuffle phase. Therefore, the amounts of the intermediate data can be cut down, so as to improve the utility of bandwidth and CPU resources, and then decrease the latency time. Finally, in Reduce phase, the final results of this application are generated.

In MapReduce, the users need to implement Map interface and Reduce interface to implement a MapReduce application. ComMapReduce is easy to use only by adding a programming interface to make sure how to generate the *shared information*. The interface is shown in Figure 4. The users can extend this interface to implement the basic algorithms. In this interface, *shared information* can be any type described by *InfoType*. Function *send* is used to send *shared information*, while the *receive* function is used to receive the *shared information*. Function *find* is used to identify the *shared information* with our communication strategies.

2.3. ComMapReduce Analysis

2.3.1. ComMapReduce Availability

The availability of ComMapReduce is dependent on MapReduce. The recovery strategies of the Master node and the Slave nodes are the same as MapReduce. The fault tolerance of the input and output of ComMapReduce is reached by replication in the underlying HDFS. When a Map task fails to generate the intermedi-

ate data, the Map task re-executes on another node. The intermediate data become unavailable of the failed Map task, but it does not affect the Map task generating its local *shared information*, because the *shared information* is obtained after the Map task completes. For the recovery of the Coordinator node, easy method of periodic checkpoints can be used to check it whether fails or not. If the Coordinator node fails, a new copy can be deployed from the last checkpoint state to implement the unprocessed Mappers. However, there is only a single Coordinator node of ComMapReduce framework, so its failure is unlikely. Our current implementation stops the ComMapReduce computation if the Coordinator node fails. The clients can check and restart the ComMapReduce operation according to their desire. Therefore, ComMapReduce has its availability the same as MapReduce.

2.3.2. ComMapReduce Scalability

The scalability of ComMapReduce inherits the scalability of MapReduce. The Master node and the Slave nodes retain the same functions of MapReduce, so they own the same scalability as MapReduce. To illustrate the scalability of ComMapReduce, we only need to illustrate that the scalability of ComMapReduce is not affected by the Coordinator node. In the following, we analyze the scalability of the Coordinator node from two aspects: time complexity and memory usage.

Time Complexity: Suppose that the number of system setting Map tasks is M indicating that the number of Map tasks implementing at the same time is no more than M. Each Map task sends its *shared information* to the Coordinator node. That is to say, the maximum number of message packages sent to the Coordinator node at the same time is M. However, the probability of all Mappers completing at the same time is so low that this situation scarcely happens. As a result, if we can illustrate the scalability of ComMapReduce is not affected in this extreme case, the scalability is not affected certainly in the other cases. Further, if we can illustrate the time complexity of the Coordinator node is the same as the Master node, the scalability of ComMapReduce is not affected by the Coordinator node.

- First, the time complexity of the Coordinator node receiving the *shared information* of Map tasks is the same as the time complexity of the Master node receiving Map tasks requirements of locating data chunks.

- Second, the time complexity of the Coordinator node identifying the global *shared information* is the same as the time complexity of the Master node identifying the data chunks locations to Map tasks.

- Third, the time complexity of the Coordinator node sending a global *shared information* to Map tasks is equal to the time complexity of the Master node sending the locations of data chunks to Map tasks.

Therefore, the scalability of ComMapReduce is not affected by the Coordinator node from the aspect of time complexity.

Memory Usage: The memory usage of the Coordinator node is much lower than the Master node for only storing the *shared information* and other variables of communication strategies. Therefore, the scalability of ComMapReduce is not affected by the Coordinator node from the aspect of memory usage.

From the above analysis, we draw a conclusion that ComMapReduce has the same availability and scalability as the original MapReduce.

3. ComMapReduce Communication Strategies

In order to effectively prune the unpromising data to be transferred in the shuffle phase, we propose some communication strategies to communicate with the Coordinator node. Without influencing the results correctness of applications, these strategies can identify the global *shared information* stored in a message package. ComMapReduce does not change the functions of the Master node in MapReduce, so we only present the communication strategies of the Coordinator node. We design three basic communication strategies to illustrate how to communicate with the Coordinator node to obtain the global *shared information*. Furthermore, two optimization communication strategies are proposed to enlarge the ways of receiving and generating the *shared information*. In this section, a complex top-*k* query example is adopted to illustrate the basic three communication strategies, Lazy, Eager and Hybrid, and two optimization communication strategies, Prepositive and Postpositive.

EXAMPLE 2. (Complex Top-k Query) The input dataset is 50 data, from '1' to '50'. There are five Mappers and one Reducer. Each Mapper processes 10 data respectively. Top-5 query returns the 5 biggest data as the final outputs, '50', '49', '48', '47' and '46'.

3.1. Lazy Communication Strategy

The principle of Lazy Communication Strategy (LCS) is that the Coordinator node generates a global *shared information* after all Mappers complete. It receives the local *shared information* of all Mappers and chooses the most optimal one as the global *shared information*. All the Mappers wait for the global *shared information* from the Coordinator node to filter their intermediate data before writing

into their local disks and then generate the input of Reducers. The Reducers compute the final results with less input than the Reducers in MapReduce.

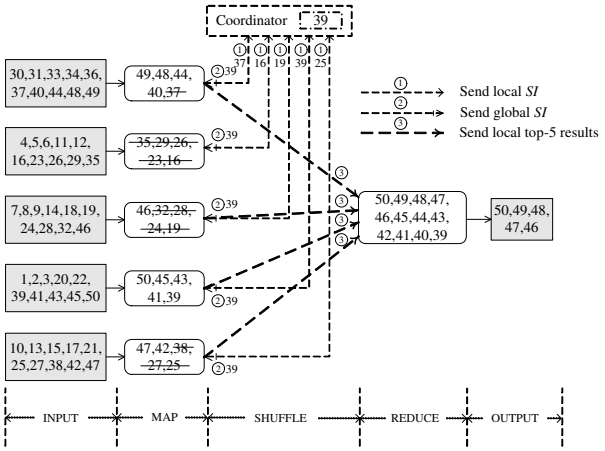


Figure 5: Top-*k* Query Processing on ComMapReduce with LCS

Figure 5 shows the complex top-*k* query processing on ComMapReduce with LCS. Each Mapper processes its own data and generates its local *shared information* (shown as *SI* in Figure 5 and the following figures and equation), the value of the *k*th one of its intermediate data, and then sends it to the Coordinator node. After the Coordinator node receives all the message packages with local *shared information* of Mappers, '37', '16', '19', '39', '25', it chooses the biggest one, '39', as the global *shared information*. The intermediate data smaller than '39' are all filtered. The final input of Reducer is about 48% of the original dataset without filtering. By obtaining the most optimal one from the local *shared information* of all Mappers as the final global *shared information*, LCS filters the intermediate data which are not the final results largely.

3.2. Eager Communication Strategy

The principle of Eager Communication Strategy (ECS) is as follows. After each Mapper completes, it sends its local *shared information* to the Coordinator node and receives a global *shared information* simultaneously. If the content of this global *shared information* is null, the Mapper processes its split normally. Otherwise, the Mapper prunes its intermediate data with the global *shared information*. Once receiving a message package containing a local *shared information* of another Mapper, the Coordinator node chooses the better one as a new global

shared information by comparing with the original one. The Coordinator node sends the message package containing the new global *shared information* to the completed Mappers before the intermediate data writing into the local disks. This course still repeats until all the Mappers on the platform complete.

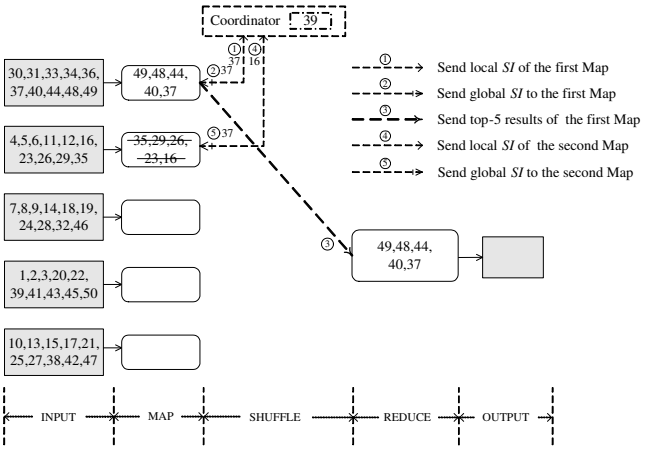


Figure 6: Top-*k* Query Processing on ComMapReduce with ECS

Figure 6 shows the complex top-*k* query processing on ComMapReduce with ECS. Six steps are taken to illustrate the implementations of the first and the second Mapper. Step 1. After the first Mapper completes, it sends data '37' to the Coordinator node as its *shared information*. Step 2. After receiving '37', the Coordinator node sets '37' as the global *shared information* and sends it to the first Mapper. Step 3. The final intermediate data of the first Mapper are data '49', '48', '44', '40', '37' after filtering. Step 4. The second Mapper sends its local *shared information* '16' to the Coordinator node. Step 5. After the Coordinator node receives '16', it chooses the bigger one '37' as the global *shared information*. Step 6. The second Mapper filters its intermediate data with '37'. All the intermediate data of the second Mapper are smaller than '37', so there is no intermediate data of the second Mapper without showing in Figure 6. This course continues until the last Mapper completes and then generates the final results.

The Coordinator node can send a message package with the global *shared information* in time with ECS instead of waiting for all the Mappers completing. The Mappers can receive a global *shared information* immediately to filter their unpromising intermediate data.

3.3. Hybrid Communication Strategy

In order to both effectively filter the unpromising intermediate data with a better global *shared information* and improve the response speed of applications, Hybrid Communication Strategy (HCS) is proposed. The main workflow of HCS is as follows. Suppose that one Mapper completes, it sends its local *shared information* to the Coordinator node at once. However, the Coordinator node waits for a preassigned a period of time (t_w) to receive the other Mappers' message packages with *shared information* instead of generating a global *shared information* immediately. After receiving the other completed Mappers' *shared information*, the Coordinator node chooses the most optimal one as the global *shared information* according to the application. The Mappers receive the global *shared information* from the Coordinator node to prune their intermediate data. The same course repeats until all the Mappers in the system complete. Nevertheless, when the last Mapper completes, it sends its *shared information* to the Coordinator node and receives the global *shared information* immediately.

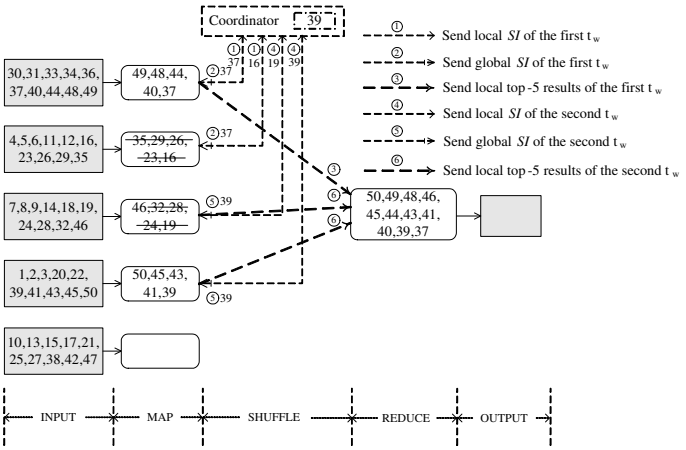


Figure 7: Top-*k* Query Processing on ComMapReduce with HCS

Figure 7 shows the complex top-*k* query processing on ComMapReduce with HCS. For simple description, we suppose that there are two completed Mappers in each preassigned a period of time t_w. Six steps are taken to illustrate the implementations of the first and the second t_w. Step 1. The first Mapper generates '37' as its local *shared information* and sends it to the Coordinator node. After receiving '37', the Coordinator node waits for the second Mapper completing in the first t_w and receives its local *shared information* '16' instead of generating

a global *shared information* in time. Step 2. After receiving '37' and '16', the Coordinator node chooses the larger one, '37', as the global *shared information* and sends it to the first and the second Mapper. Step 3. The first and the second Mapper filter their intermediate data by '37'. Step 4. In the second t_w, the third and the fourth Mapper complete and send their local *shared information* '19' and '39' to the Coordinator node. Step 5. After receiving '19' and '39', the Coordinator node compares '19' and '39' to the original one '37' and sets '39' as a new global *shared information*. Step 6. After receiving the global *shared information* '39', the third and the fourth Mapper filter their superfluous intermediate data and then send the rest to the Reducer.

HCS neither waits for the completion of all the Mappers to generate a global *shared information* nor immediately receives a global *shared information* after each Mapper completes. It can both effectively filter the unpromising intermediate data of Mappers and do not need to wait for a long time to generate a better global *shared information*.

We can see that identifying a suitable t_w is important to improve the performance of HCS. In actual big data applications, when processing a part of Mappers, the Coordinator node may generate the same *shared information* during the processing course. That is to say, the temporary global *shared information* trends to the optimal global *shared information* of this application. There is no need to wait for a t_w to send the global *shared information*. So, we can choose a smaller t_w and then shorten the response time. Conversely, in each t_w, the message packages with local *shared information* change sharply, that is to say, the temporary *shared information* is not stable. The Coordinator node should wait for a longer t_w to receive more local *shared information* and generate a more suitable global *shared information*.

$$t_{wnew} = \begin{cases} \max(T, \frac{n}{m} \times t_{wold}) & \textit{no new SI during } t_{wold} \\ \min(\frac{n}{m} \times t_{wold}, m \times T) & \textit{new SI during } t_{wold} \end{cases} \quad (1)$$

Equation 1 illustrates the method of changing t_w, where *T* means the time of sending the message package with *shared information*, tested by the ratio of the size of the message package and the bandwidth of the network; *n* and *m* stand for the number of Mappers and Reducers on the platform. When there is no new *shared information* during t_{wold}, we can decrease t_w and set t_{wnew} to the maximum of T and $\frac{n}{m} \times t_{wold}$. When there is new *shared information* during t_{wold}, we can increase t_w and set t_{wnew} to the minimum of $\frac{n}{m} \times t_{wold}$ and $m \times T$. Therefore, we can identify an optimal period time t_w by Equation 1.

3.4. Optimization Communication Strategy

In practical applications, the number of Mappers is much larger than the number of the system slots, about hundreds multiples. Even adopting the above communication strategies to filter the unpromising data, there are still numerous unpromising data that can be filtered ulteriorly. Therefore, we propose two optimization strategies to enhance the performance of ComMapReduce on big data applications. The two optimization communication strategies are orthometric to the above communication strategies. Therefore, they can be combined with any communication strategy above. In addition, the two optimization strategies can be used together to further improve the performance of ComMapReduce.

3.4.1. Prepositive Optimization Strategy

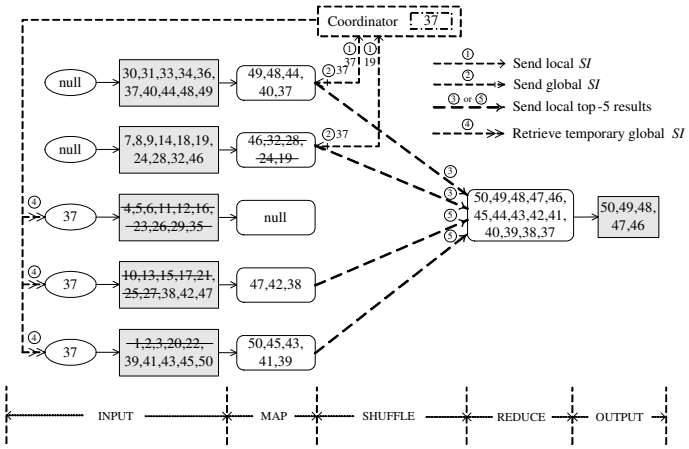


Figure 8: Top-*k* Query Processing on ComMapReduce with PreOS

The principle of Prepositive Optimization Strategy (PreOS) is as follows. After a part of Mappers complete, the Coordinator node generates a temporary global *shared information* by any communication strategy above. The other unprocessed Mappers can retrieve the temporary global *shared information* in their initial phases from the Coordinator node, and then prune their data by the global *shared information*. The data to be sorted by Mappers decreases greatly, so the sorting time of Mappers also decreases.

Figure 8 shows the complex top-*k* query processing on ComMapReduce with PreOS. We take five steps to illustrate the workflow of PreOS. For simple description, from Step 1 to Step 3, the Coordinator node generates a temporary global

shared information ‘37’ with HCS after the first and the second Mapper complete. Step 4. The other three unprocessed Mappers retrieve ‘37’ from the Coordinator node in their initial phases and filter their input data that cannot be the final results. Step 5. The third to the fifth Mappers send their intermediate data to the Reducer only with PreOS not using any communication strategy above.

In practical applications, when the number of Mappers is much more than the number of the system slots, the Mappers run in several batches. The unprocessed batches of Mappers can filter the unnecessary data in time with PreOS. They retrieve a temporary global *shared information* from the Coordinator node in their initial phases to enhance the performance of ComMapReduce.

3.4.2. Postpositive Optimization Strategy

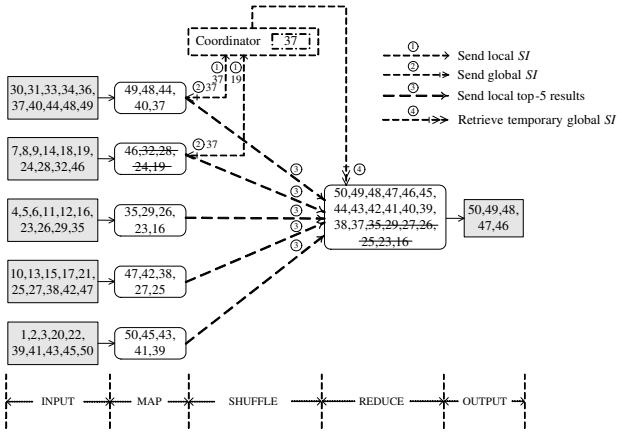


Figure 9: Top-*k* Query Processing on ComMapReduce with PostOS

When all the Mappers in the system are processed in one batch, Postpositive Optimization Strategy (PostOS) is proposed to further filter the input data of Reducers. The main idea of PostOS is as follows. After the intermediate data of Mappers are sent to the Reducers, the Reducers first retrieve a temporary global *shared information* from the Coordinator node and filter their input data again. As the global *shared information* changes along the running of Mappers, it is necessary to filter the input of Reducers again. Therefore, the Reducers process the input data after filtering to enhance the performance of ComMapReduce.

Figure 9 shows the complex top-*k* query processing on ComMapReduce with PostOS. Only four steps are taken to illustrate the implementation of PostOS simply. From Step 1 to Step 2, after the first and the second Mapper complete, the Co-

ordinator node generates a temporary global *shared information* '37' with HCS. Step 3. All Mappers send their intermediate data to the Reducer without using any communication strategy above. Step 4. After receiving the intermediate data of the Mappers, the Reducer retrieves the temporary global *shared information* '37' from the Coordinator node and prunes its input data again. The data smaller than '37' are all filtered. Therefore, the input of Reducer decreases significantly again so as to relieve the workload of Reducer.

4. Typical Applications of ComMapReduce

ComMapReduce can effectively prune the unpromising intermediate data, especially when the users are interested in a respective small subset from large scale datasets. ComMapReduce can be used in a wide application fields, such as the applications of top-*k* query, *k*-nearest neighbor search, skyline, top-*k* skyline, top-*k* join, top-*k* group. The users only need to identify the *shared information* of big data applications to implement the applications with the communication strategies. In this section, we take three big data applications as examples to illustrate the executions of these applications on ComMapReduce, *k*NN, skyline and join.

4.1. kNN Query Processing

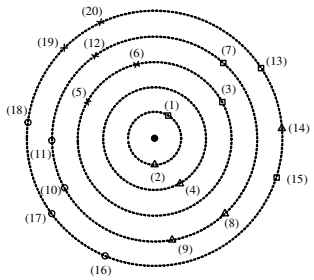


Figure 10: Data Space of *k*NN Query

*k*NN query is widely used in the areas of data mining, machine learning, pattern recognition and so on. Based on a specific similarity function, a *k*NN query returns the *k* data $(k \geq 1)$, which are the most similar to the query data according to the similarity function. ComMapReduce can effectively process big data *k*NN query with any communization strategy of ComMapReduce in Section 3. We only present the processing course of *k*NN query by Hybrid Communication Strategy for convenience in the following example.

Figure 10 shows the data space of a concentric circles. The radius of each two adjacent concentric circles is set as unit one. There are twenty data, number from *(1)* to *(20)*, and the query point is the center point. The similarity function is the distance between the query point and the candidate data, easily evaluated by the number units of the radius interval. For example, the distance between data *(3)* and the query point is 3 units. The *k*NN query returns the two data nearest to the query point, data *(1)* and *(2)*.

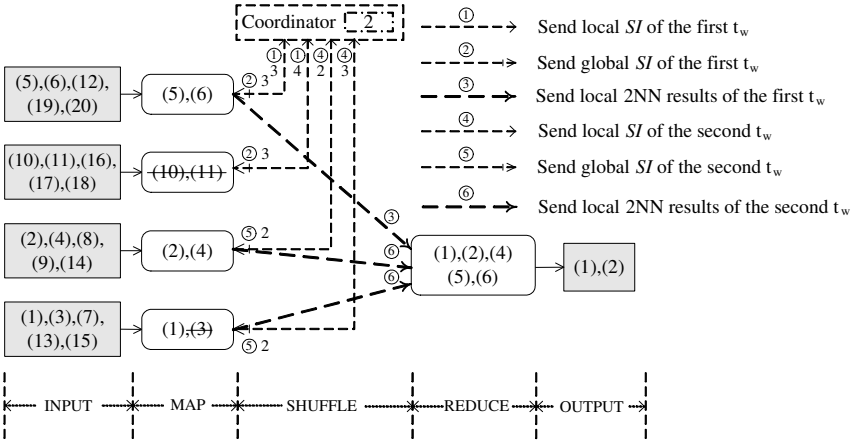


Figure 11: *k*NN Query Processing on ComMapReduce with HCS

The implementation of the above *k*NN query is shown in Figure 11. The twenty data are divided into four splits and processed by four Mappers. Suppose there are also two completed Mappers in a t_w. In the first t_w, the first Mapper processes data *(5)*, *(6)*, *(12)*, *(19)*, *(20)* and generates its intermediate data *(5)*, *(6)*. The first Mapper emits the *k*th data distance nearest to the query data as its local *shared information*, distance units 3, sending to the Coordinator node. The Coordinator waits for the local *shared information* of the second Mapper, distance units 4, and chooses the optimal one, distance units 3, as the temporary global *shared information*. After the first and the second Mapper receive distance units 3, they filter their intermediate data which distance units are smaller than units 3, data *(10)* and *(11)*. In the second t_w, the Coordinator node receives the local *shared information* of the third and the fourth Mapper, distance units 2 and 3. It chooses the smaller one, distance 2, to compare with the original global *shared information*, and then generates the new global *shared information*, distance units 2. The third and the fourth Mapper filter the unpromising data *(13)*. In Reduce phase, the Reducer generates the final results of this *k*NN query.

4.2. Skyline Query Processing

Skyline query has gained considerable attention from database and information retrieval research communities. A skyline query returns a set of data that cannot be dominated by any other data in the given dataset. The dominance relationship is a search criterion. One data dominates another one means that it is not worse in all search criterions and is better in at least one criterion. For example, a classical skyline query is to help the travelers find the hotels both near to the beach and with cheap price. ComMapReduce can also process the skyline query with large scale datasets, whatever adopting any communication strategy in Section 3. We use the following simple example to illustrate the implementation of skyline query on ComMapReduce with Hybrid Communication Strategy.

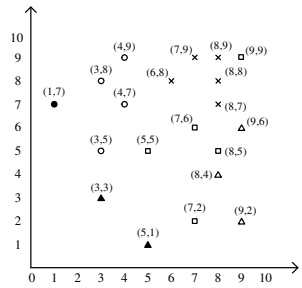


Figure 12: Data Space of Skyline Query

Figure 12 shows the data space of a skyline query in two-dimensional space with twenty data points. The solid data points, (*1,7*), (*3,3*), (*5,1*), are the final results of this skyline query. After each Mapper completes, it chooses the skyline point with the optimal filtering ability as its local *shared information* sending to the Coordinator node. The filtering ability of a skyline point can be determined by any skyline algorithm. In this example and our experiments in Section 5, we adopt the Sliding Window Skyline Monitoring Algorithm (SWSMA) [17] to evaluate the filtering ability of the skyline points.

Figure 13 shows the the implementation of the skyline query above with HCS on ComMapReduce. The twenty data points are divided into four splits and processed by four Mappers. We still suppose there are two completed Mappers in one t_w. In the first t_w, when the first Mapper completes, it chooses the data point (*3,5*) with the optimal filtering ability according to the algorithm SWSMA, and sends it to the Coordinator node as its local *shared information*. After the second Mapper completes, it sends its local *shared information*, data point (*6,8*), to the Coordinator node. The Coordinator node chooses the data point (*3,5*) as the global *shared*

information and sends it to the completed Mappers in the first t_w. After that, the intermediate data of the second Mapper are all filtered. The same course is applied to the second t_w. The Coordinator node receives the local *shared information* of the third and the forth Mapper, data points (*5,5*) and (*3,3*), and chooses (*3,3*) as the global *shared information* after comparing with data point (*3,5*). The intermediate data of the third Mapper are all filtered. After that, the Reducer generates the final results of this skyline query.

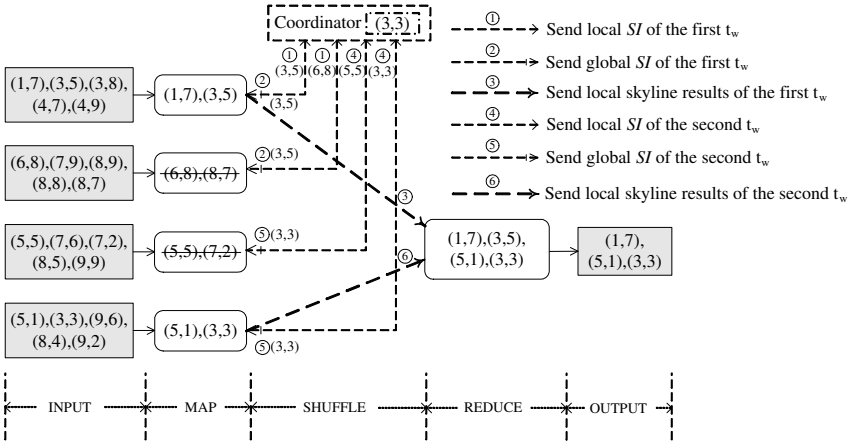


Figure 13: Skyline Query Processing on ComMapReduce with HCS

4.3. Join Query Processing

In recent years, based on the observations from the main Web 2.0 companies, like Yahoo!, Google and Facebook, log processing emerges as an important application of data analysis done by MapReduce. MapReduce can be used to compare statistics information and analyze business insights from the results of log processing. The log information is collected and stored in files. That is to say, join processing among the log files is essential for many data analysis and is also a research hotspot of MapReduce. There are several research achievements on the join query applications by MapReduce [8, 9, 10, 11, 18, 19, 20].

After analyzing the implementations of join queries on MapReduce, we can see that there are numerous intermediate data to be transferred and copied in the shuffle phase. They lead to the overhead of the network bandwidth and disk access. So the shuffle phase occupies most time of join queries. Simultaneously, in big data join applications, the final results are always much smaller than the original join files. Especially, when processing multi-table join, there maybe no final

results by the reason of the uneven distribution of the join attributes. Therefore, if the needed join attributes can be obtained ahead of time, the unpromising intermediate data can be filtered, decreasing the cost of network transmission and disk access. ComMapReduce is an improvement of MapReduce with lightweight communication mechanisms. When processing join query of log files, ComMapReduce can store the *shared information* in the message package to filter the unpromising data. Therefore, ComMapReduce can process join query effectively. In the following, we only take equi-join between two tables as a simple example to illustrate that our ComMapReduce framework can process join query.

ID	C1
n_1	t_A
n_2	t_B
n_1	t_C
n_1	t_D
n_2	t_E
n_2	t_F

(a) R_1

ID	C2
n_1	t_a
n_2	t_b
n_3	t_c
n_4	t_d
n_5	t_e
n_6	t_f

(b) R_2

Figure 14: Two Join Tables

Two join tables are shown in Figure 14 with the join attribute *'ID'*. In Map phase, to identify the original table, each Mapper adds a *tag* to make sure the source table of one <*key,value*> pair, where *tag* 1 presents the data from table R_1 and *tag* 2 presents the data from table R_2. Therefore, the intermediate data of Mappers are extracted as <*key,value*> pairs in this way. For example, $< n_1, 1, t_A >$ means this <*key,value*> pair is from table R_1. An example of equi-join between two join tables on ComMapReduce is shown in Figure 15 by HCS of ComMapReduce. In Map phase, after processing table R_1, the Map task queue stops, and all the Mappers that process table R_1 send their local *shared information* (join attribute *'ID'*) to the Coordinator node. After that, the Coordinator node merges the local *shared information* and gains the global *shared information*, table $T(n_1, n_2)$, and sends it to the other Mappers. The Coordinator node restarts the Map task queue and begins to process table R_2. The intermediate data not having those attributes (n_1, n_2) can be filtered because those data cannot be the final results, avoiding the following processing of these intermediate data. After filtering, the cost of network and disk access can be decreased and the latency time of join query can also be shortened. One Reducer generates the final results of this equi-join between the two tables.

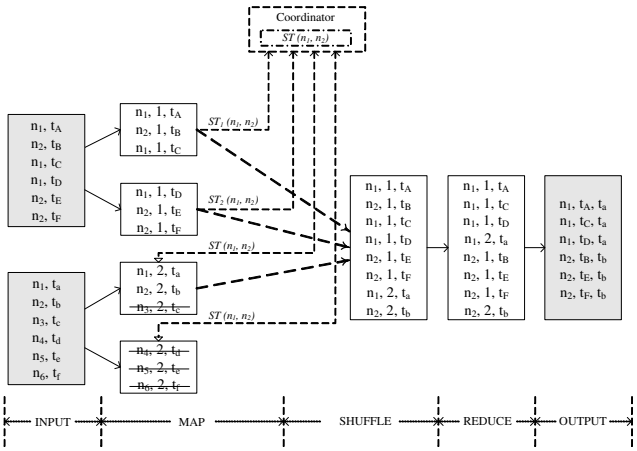


Figure 15: Join Processing of ComMapReduce

In ComMapReduce, we choose Bloom Filter [21] to store the *shared information*. We modify the Map task queue to make the tables scanned in a defined order. We adopt *join ratio*, similar to the *join cardinality*, to determine which file should be scanned first. The *join ratio* equals the ratio of the number of attributes and the size of the table. The table with the smaller *join ratio* should be scanned first, because its *shared information* is fewer and has more optimal filtering ability. For example, table R_1 contains 2 *'ID'* attributes, n_1, n_2, and 6 data records, so its *join ratio* is $\frac{2}{6}$. Table R_2 contains 6 *'ID'* attributes and 6 data records, so its *join ratio* is $\frac{6}{6}$. Therefore, we scan table R_1 first. When processing big data join applications, the *join ratio* is statistical, such as setting counter during the course of data storing to compute *join ratio* or through sampling to evaluate it. If the *join ratio*s of the two tables are the same, we scan the smaller size table first.

5. Experiments

In this section, four types of applications are taken as examples to evaluate the performance of our ComMapReduce framework. Two communication strategies of ComMapReduce, ECS, HCS and two optimization strategies, PreOS and PostOS, are taken to compare with MapReduce (MR).

We do not evaluate LCS based on the following observations. First, all the completed Mappers must wait for the message package with the optimal global *shared information* until writing their intermediate data into their disks. That means the first completed Mappers must wait for a long time to generate its out-

puts. Second, in actual applications, the running Mappers cannot complete as they have not received the message package with the global *shared information* while the non-running Mappers cannot run as there is not enough slots for the Mappers. Therefore, LCS is not widely used in actual applications and there is no need to evaluate this communication strategy.

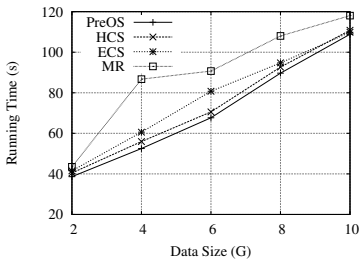
5.1. Experimental Setup

We set up a cluster of 9 commodity PCs in a high speed Gigabit network, with one PC as the Master node and the Coordinator node, the others as the Slave nodes. Each PC has an Intel Quad Core 2.66GHZ CPU, 4GB memory and CentOS Linux 5.6. We use Hadoop 0.20.2 and compile the source codes under JDK 1.6. Two main experimental benchmarks are the running time of query processing and the number of Reducer input records. One data record contains 10B. We compare ECS, HCS and PreOS to MapReduce processing top-*k* and *k*NN query. We compare ECS, HCS and PostOS to MapReduce processing skyline query. We use HCS to process the equi-join of two tables and compare with MapReduce. The experimental parameters of top-*k*, *k*NN, skyline and join query are as follows:

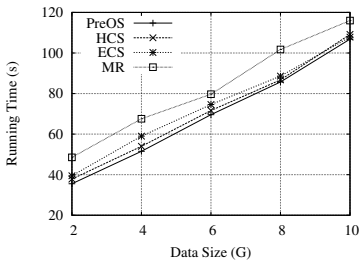
- The top-*k* query is evaluated in uniform distribution and *zipf* distribution. The default size of dataset is 10G, ranging from 2G to 10G. The default number of *k* value of top-*k* query is 1000.
- The *k*NN query is evaluated in uniform distribution and clustered distribution. The default size of dataset is 10G, ranging from 2G to 10G. The default number of *k* value of *k*NN query is 1000 and the default number of data dimensions of *k*NN query is 4.
- The skyline query is evaluated in independent distribution and anti-correlated distribution. The default size of dataset is 1G, ranging from 0.2G to 0.4G. The default number of data dimensions of skyline query is 4.
- The join query is evaluated in uniform distribution and *zipf* distribution. The default size of each table is 2G, ranging from 2G to 20G. The default attribute distribution proportion is 5%, with a range of 1%, 5%, 10%, 65%, 75%.

5.2. Experiments of Top-k Query

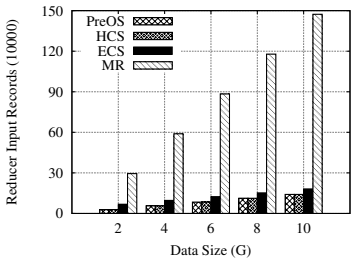
Figure 16 shows the performance of changing data size of top-*k* query. The number of Reducer input records of our three strategies are much smaller than



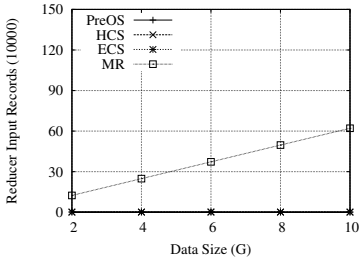
(a) Uniform Distribution



(b) $zipf$ Distribution

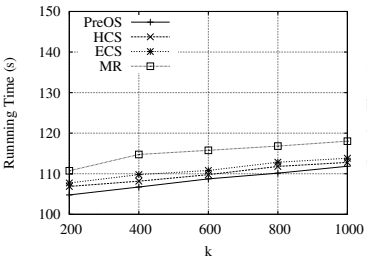


(c) Uniform Distribution

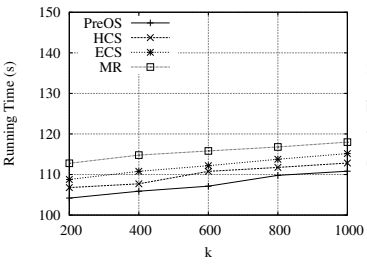


(d) $zipf$ Distribution

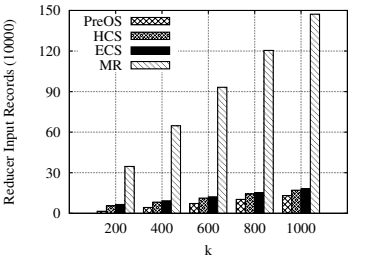
Figure 16: Changing Data Size of Top-k Query



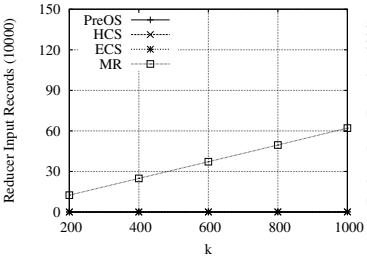
(a) Uniform Distribution



(b) $zipf$ Distribution

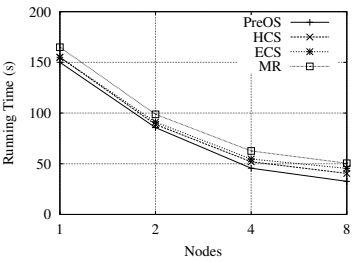


(c) Uniform Distribution

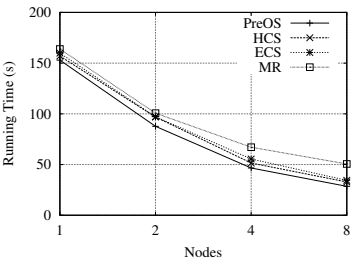


(d) $zipf$ Distribution

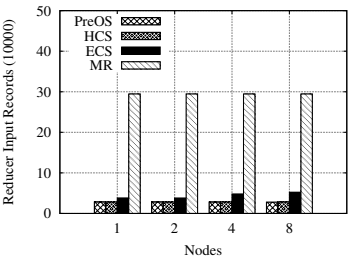
Figure 17: Changing k of Top-k Query



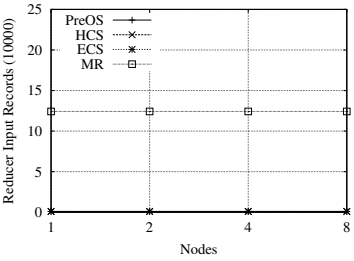
(a) Uniform Distribution



(b) $zipf$ Distribution



(c) Uniform Distribution



(d) $zipf$ Distribution

Figure 18: Changing Cluster Size of Top-k Query

MapReduce. This phenomenon illustrates that ComMapReduce can effectively filter the unpromising data with high performance of processing top-*k* query. The similar changing trend of two distributions illustrates that different data distributions have no influence on ComMapReduce framework. That shows that our ComMapReduce has the same scalability as the original MapReduce. The running time of PreOS, HCS and ECS are shorter than MapReduce. One reason is that the experimental environment is a high speed Gigabit network and the transmitting time itself is very short. Another reason is that our ComMapReduce can prune numerous superfluous data and save the running time. In *zipf* distribution, the data are skewed to the query results. It illustrates that it can quickly reach the optimal global *shared information* in Map phase. So, in Reduce phase, our three efficient communication strategies only have 1000 records for its effective filtering ability without obviously showing in Figure 16(d).

Figure 17 illustrates the performance of changing *k* of top-*k* query. PreOS, HCS and ECS are better than MapReduce both in uniform distribution and *zipf* distribution. The similar changing trend of different data distributions also shows the high scalability of ComMapReduce. As the *k* of top-*k* query increases, the number of final results also increases, so the number of Reducer input records increases certainly. In Figure 17(a) and Figure 17(b), the unprocessed Mappers can retrieve a temporary global *shared information* to filter their unpromising data in the initial phase by PreOS, so its running time is shorter than the others.

Figure 18 shows the performance of changing the cluster size to evaluate top-*k* query. As the number of Slave nodes increases, the running time decreases for enlarging the parallelism of the system. The similar changing trend of different data distributions illustrates that ComMapReduce can reach the same scalability of MapReduce. In addition, the running time and the number of Reducer input records of ComMapReduce are both better than MapReduce.

Table 1: Changing t_w of HCS

t_w (s)	1	2	5	10
HCS (s)	126.4	110.8	152.7	200.3

Table 1 shows the performance of changing t_w of HCS processing top-*k* query. When setting larger t_w, the unnecessary waiting time of the *shared information* adds running time. When setting smaller t_w, there maybe so much time wasting in sending and receiving the *share information*. So, a suitable t_w is important to the executions of big data applications.

5.3. Experiments of kNN Query

Figure 19 shows the performance of changing data size of *k*NN query. As the data size increases, the running time and the number of Reducer input records both increase sharply. The number of Reducer input records of ComMapReduce is extremely fewer than MapReduce showing the efficient filtering ability of ComMapReduce, about *14%* of the number of Reducer input records of MapReduce. PreOS is better than HCS and ECS by obtaining a global *shared information* in the initial phases of Mappers and filtering numerous unpromising data.

Figure 20 illustrates the performance of changing the dimensions of *k*NN query. As the dimensions of *k*NN query increases, the running time and the number of Reducer input records increase accordingly as both the computation and transmission cost all increase. The number of Reducer input records is much fewer than MapReduce showing the efficient filtering ability of ComMapReduce.

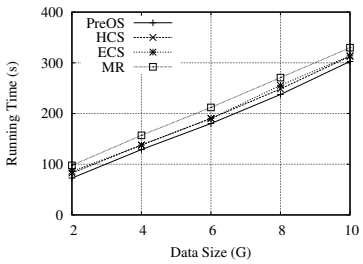
Figure 21 shows the performance of changing *k* of *k*NN query. With increasing *k*, the final results increase, so the running time and the number of Reducer input records also increase. The running time of ComMapReduce and the number of Reducer input records of ComMapReduce are much superior to MapReduce.

5.4. Experiments of Skyline Query

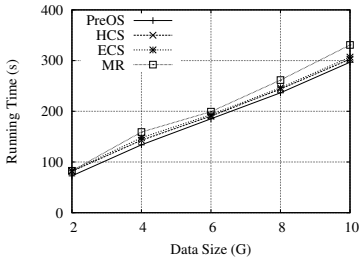
When we deal with the skyline query in parallel, the time latency of query can get the lowest bound with the number of Mappers changing from 3 to 6. The number cannot reach the maximum Mappers parallel running on the cluster, so the Map tasks can finish in a batch and there is no effect using PreOS to evaluate the performance of skyline query. Therefore, we choose PostOS to compare with the other strategies.

Figure 22 shows the performance of changing data size of skyline query. As the data size increases, the running time and the number of Reducer input records increase accordingly. ComMapReduce is more optimal to MapReduce under different data distributions. In *anti-correlated* distribution, although the data are skewed to the final results and the computation cost increases accordingly, the performance of ComMapReduce is still better than MapReduce.

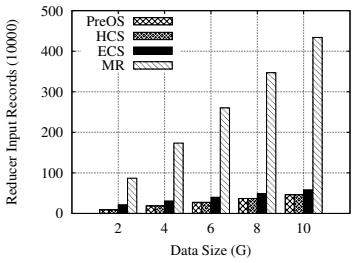
Figure 23 illustrates the performance of changing the dimensionality of skyline query. The running time and the number of Reducer input records all increase by increasing the dimensionality of skyline query. The cost of calculation increases with increasing dimensionality. In Figure 23(b) and Figure 23(d), the data are skewed to the final results, so the number of unpromising data is small, but our ComMapReduce is better than MapReduce.



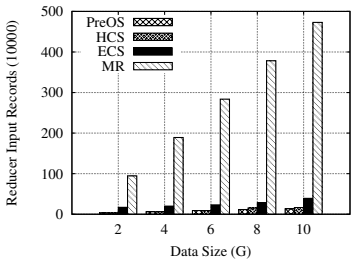
(a) Uniform Distribution



(b) Cluster Distribution

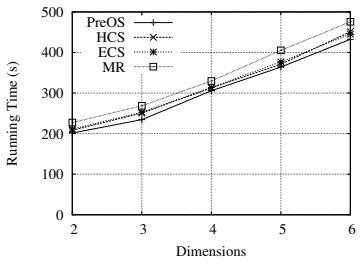


(c) Uniform Distribution

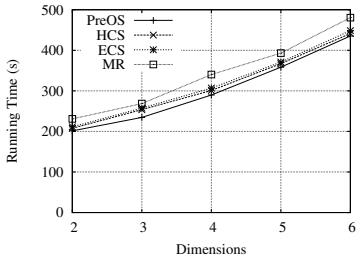


(d) Cluster Distribution

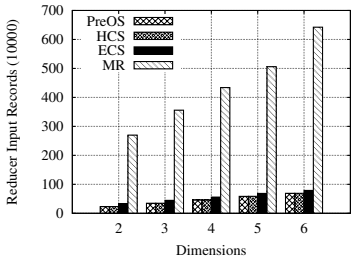
Figure 19: Changing Data Size of kNN Query



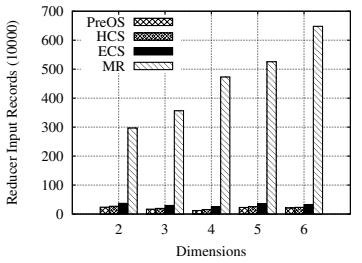
(a) Uniform Distribution



(b) Cluster Distribution

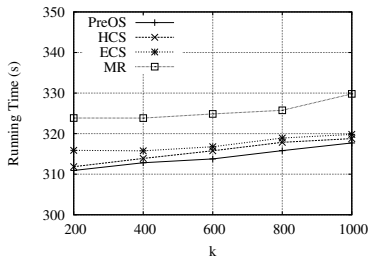


(c) Uniform Distribution

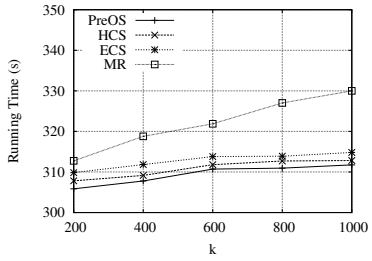


(d) Cluster Distribution

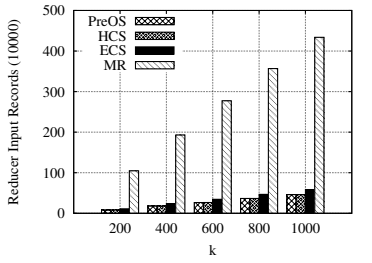
Figure 20: Changing d of kNN Query



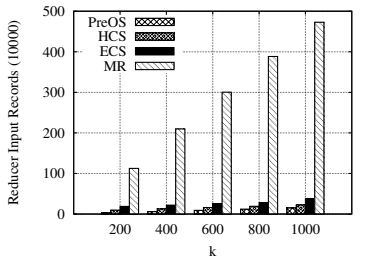
(a) Uniform Distribution



(b) Cluster Distribution



(c) Uniform Distribution



(d) Cluster Distribution

Figure 21: Changing k of kNN Query

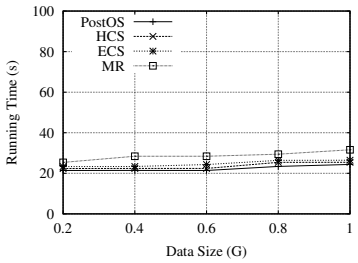
Figure 24 shows the performance of changing the cluster size of skyline query. The running time decreases as the number of nodes increases for enhancing the parallelism of the system. The same changing trend of different distributions shows that ComMapReduce has high scalability. ComMapReduce filters the unpromising data with efficient communication mechanisms, so the number of Reducer input records of ComMapReduce is fewer than MapReduce. Without using efficient filtering strategies, the number of Reducer input records of MapReduce keeps a constant.

5.5. Experiments of Join Query

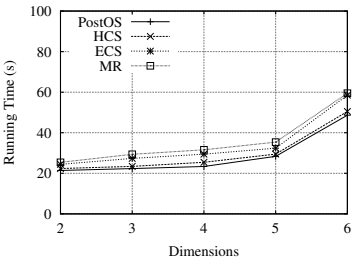
Figure 25 shows the performance of changing the size of each join table in two distributions with the same attribute percentage 1%. With the increasing of table size, the size of homologous *shared information* also increases, so the running time of join query also becomes longer. Our ComMapReduce can filter the unpromising attributes data, so the performance of ComMapReduce is better than MapReduce in Figure 25(a) and 25(b). The Filter Percentage means the filter percentage of ComMapReduce contrary to MapReduce. Although the size of join table increases, our ComMapReduce still reach the similar filter percentage. Therefore, we can see that ComMapReduce has nice scalability.

Figure 26 illustrates the performance of changing the attribute percentage of the join table with the size of 2G. In Figure 26(a) and 26(b), MapReduce cannot filter the unpromising intermediate data and process the same data volume in every evaluation, so it keeps a stable performance. When the attribute percentage is 1%, 5%, 10%, our ComMapReduce has better performance than MapReduce. The reason is that small join attribute percentage and smaller size of *shared information* mean that ComMapReduce can filter more unpromising data. However, with the increasing number of attribute percentage, the *shared information* also increases, so the performance of ComMapReduce decreases. In actual experiments, when the attribute percentage reaches to 65%, the performance of ComMapReduce is similar to MapReduce and is lower than MapReduce with 75%. This further demonstrates that our ComMapReduce is better suitable for the situation that the final results is much smaller than the original data.

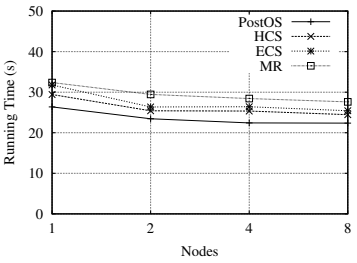
Figure 27 shows the performance of changing the *order* of the two tables. MapReduce keeps the same performance under different *order*. We can see that the *order A* can filter more intermediate data contrary to the *order B* in two distributions. Therefore, the *order* of the two tables is important to enhance the performance of ComMapReduce framework.



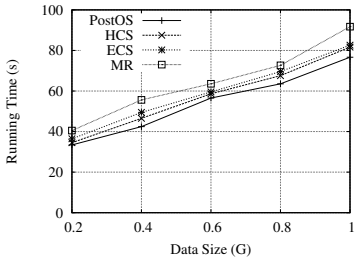
(a) Independent Distribution



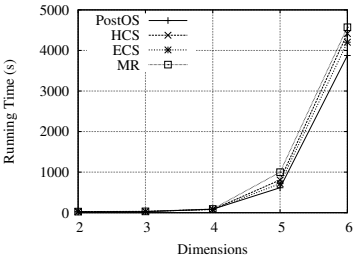
(a) Independent Distribution



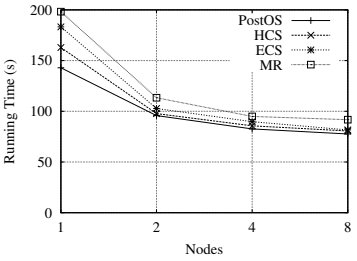
(a) Independent Distribution



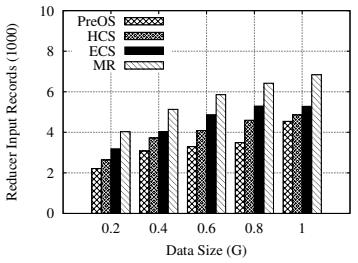
(b) Anti-correlated Distribution



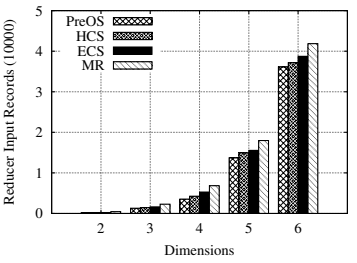
(b) Anti-correlated Distribution



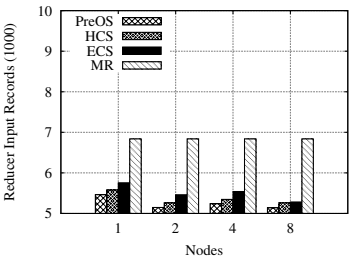
(b) Anti-correlated Distribution



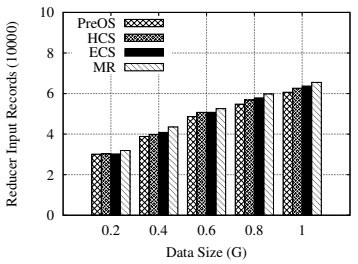
(c) Independent Distribution



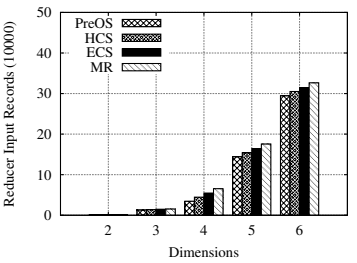
(c) Independent Distribution



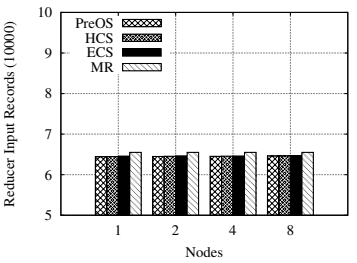
(c) Independent Distribution



(d) Anti-correlated Distribution



(d) Anti-correlated Distribution

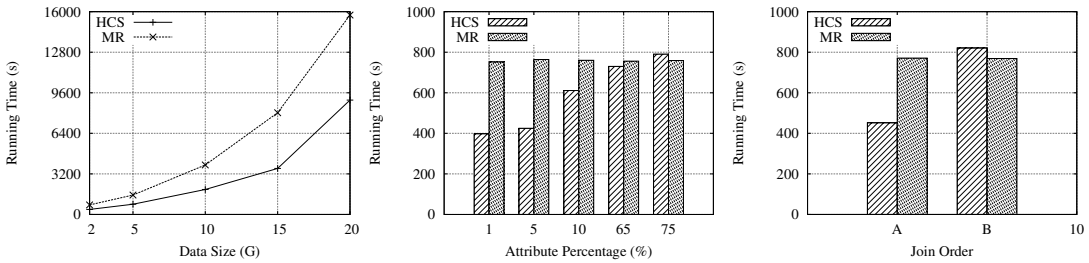


(d) Anti-correlated Distribution

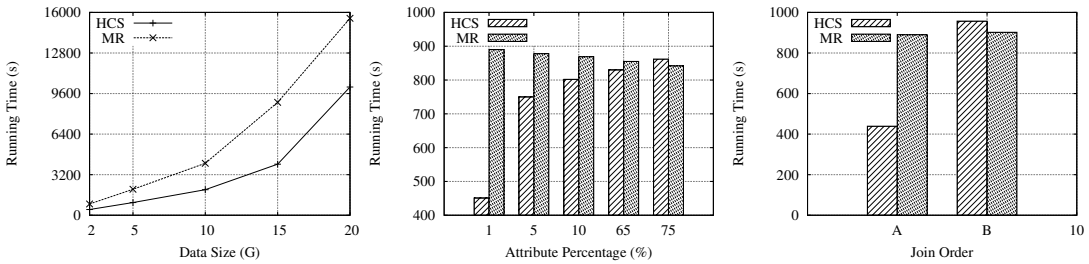
Figure 22: Changing Data Size of Skyline Query

Figure 23: Changing Dimensionality of Skyline Query

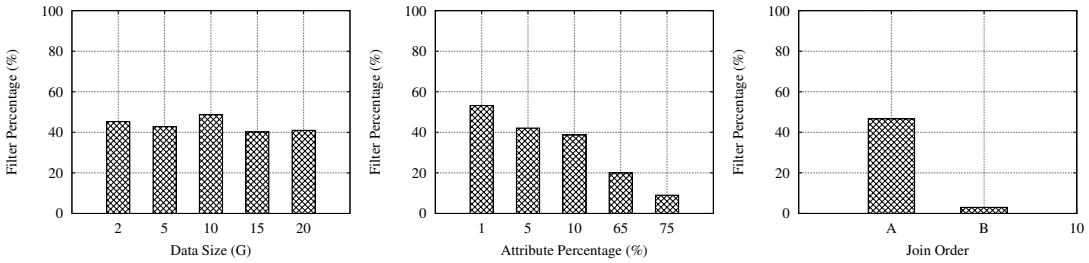
Figure 24: Changing Cluster Size of Skyline Query



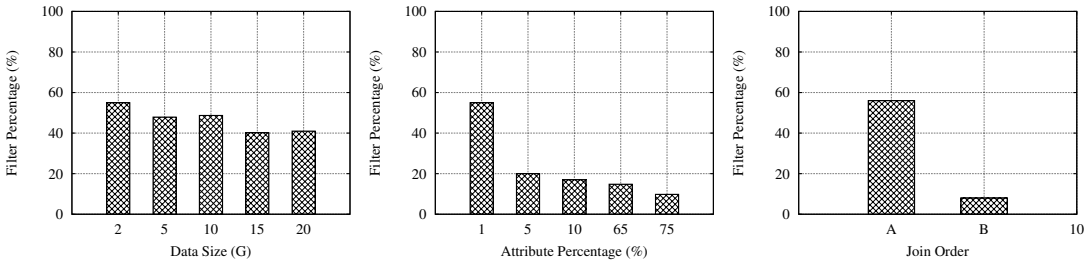
(a) Uniform Distribution



(b) $zipf$ Distribution



(c) Uniform Distribution



(d) $zipf$ Distribution

Figure 25: Changing Each Table Size of Join Query

(a) Uniform Distribution

(b) $zipf$ Distribution

(c) Uniform Distribution

(d) $zipf$ Distribution

Figure 26: Changing Attribute of Join Query

(a) Uniform Distribution

(b) $zipf$ Distribution

(c) Uniform Distribution

(d) $zipf$ Distribution

Figure 27: Changing Join Order of Join Query

6. Related Work

Google's MapReduce [3] was first proposed in 2004 for parallel big data analysis in shared-nothing clusters. Hadoop is a famous open source implementation of MapReduce, where HDFS, Hadoop MapReduce, HBase [7] and Zookeeper[3] are respectively the implementations of Google's GFS [16], MapReduce [3], Big Table [22] and Chubby [23].

Based on Hadoop, there has been followed by a series of related systems, including Dryad [24], Hive [5], Pig [4] and HadoopDB [6]. Pig [4] is a data flow language and execution environment for exploring very large datasets, helping the users specifying more complex queries in an easier way. HadoopDB [6] is a hybrid architecture of MapReduce and relational database, which tries to combine the advantages of both. Hive [5] manages data stored in HDFS and provides a query language based on SQL. These systems mainly focus on query language and programming interfaces. ComMapReduce enhances the efficiency of MapReduce with lightweight communication mechanisms to obtain the *shared information*.

The research of query processing has become a focus of cloud computing and distributed environments [25, 26, 27, 12, 28, 29, 30, 31, 20, 32, 33]. MRShare [25] proposes a sharing framework tailored to MapReduce, which transforms a batch of queries into a new batch by merging jobs into groups and evaluating each group as a single query. Jun Li et. al. [26] propose a new framework to handle continuous data stream queries using the MapReduce in the Cloud. Paper [27] describes a modified MapReduce architecture that allows data to be pipelined between operators and supports online aggregation and continuous queries. Foto N. Afrati et. al. [12] propose several algorithms for efficient recursions in the MapReduce and discuss several alternatives for supporting recovery from failures without restarting the entire job. HaLoop [28] is an improvement of MapReduce supporting iterative applications, which improves their efficiency by making the task scheduler loop-aware and by adding various caching mechanisms. Paper [29] proposes two novel algorithms in MapReduce to perform efficient parallel *k*NN joins on large scale datasets. *k*NN queries have also been studies in various other contexts in recent years such as moving objects [34] and privacy preservation [35]. The above research achievements mainly process multiple queries, recursive queries or data stream queries and so on. ComMapReduce is a modified MapReduce framework with *shared information* and can process many big data applications effectively, not one special query.

[3] http://zookeeper.apache.org/

There is a rich research of join algorithms of MapReduce framework. Map-Reduce-Merge [8] adds a merge function to support the join algorithms and different implementations, extending MapReduce framework. Map-Join-Reduce [9] extends and improves MapReduce framework to process multi-way join in system level. Paper [10] studies the crucial implementation details of a number of remarkable join algorithms in MapReduce, such as repartition join, broadcast join, semi-join and per-split semi-join. Paper [11] describes an in-depth research of a special type of similarity join algorithm in MapReduce and proposes some efficient techniques for dealing with memory limitations. Paper [18] proposes efficient algorithms which can efficiently process parallel execution of arbitrary theta-joins in MapReduce. We only want to illustrate that ComMapReduce can process join query not to analyze all the join algorithms in detail.

7. Conclusions

MapReduce is a parallel programming framework processing the large scale datasets on clusters with numerous commodity machines. In order to enhance the performance of MapReduce, in this paper, we propose the design and evaluation of ComMapReduce. ComMapReduce is an improved version of MapReduce framework with lightweight communication mechanisms, supporting applications for large scale datasets. ComMapReduce maintains the main great features of MapReduce, availability and scalability, while filtering the unpromising data by communicating with the Coordinator node. Three communication strategies, Lazy, Eager and Hybrid, are presented to effectively implement the applications of big data on ComMapReduce framework. We also propose two optimizations, named PreOS and PostOS to enhance the performance of ComMapReduce. In addition, we give the processing courses of three applications on ComMapReducde, *k*NN, skyline and join, to illustrate the wide application fields of ComMapReduce. In the experiments, we take four query processing applications, top-*k*, *k*NN, skyline and join, as examples to evaluate the performance of ComMapReduce. The experimental results demonstrate that the performance of ComMapReduce is superior to MapReduce in all metrics. In the future, first, we intend to implement other applications on ComMapReduce framework. Second, we want to identify more optimized mechanisms of choosing the *shared information* according to different applications.

Acknowledgement. This research was partially supported by the National Natural Science Foundation of China under Grant Nos. 60933001, 61025007, 61100022, and 61073063; the National Basic Research Program of China under Grant No.

2011CB302200-G; the 863 Program under Grant No. 2012AA011004, the Public Science and Technology Research Funds Projects of Ocean Grant No. 201105033, and the Fundamental Research Funds for the Central Universities under Grant No. N110404009.

References

[1] Blake Irving. Big data and the power of hadoop. Yahoo! Hadoop Summit, June 2010.

[2] Mike Schroepfer. Inside large-scale analytics at facebook. Yahoo! Hadoop Summit, June 2010.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc.of OSDI*, pages 137–150, 2004.

[4] C. Olston, B. Reed, U. Srivastava, et. al. Pig Latin: A Not-so-foreign Language for Data Processing. In *proc.of SIGMOD*, pages, 1099–1110, 2008.

[5] A. Thusoo, J. S. Sarma, N. Jain, et. al. Hive-A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[6] A. Abouzeid, K. Baida-Pawlikowski, D. Abadi, et. al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[7] D. Carstoiu, E. Lepadatu, M. Gaspar. Hbase-non SQL Database, Performances Evaluation. *IJACT-AICIT*, 2(5): 42–52, 2010.

[8] H. Yang, A. Dasdan et. al. Map-reduce-merge: Simplified Relational Data Processing on Large Clusters. In *proc.of SIGMOD*, pages, 1029–1040, 2007.

[9] David Jiang, Anthony K. H. Tung, Gang Chen: MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters. *TKDE*, 23(9): 1299–1311, 2011.

[10] Spyros Blanas, Jignesh M. Patel, et. al. A comparison of join algorithms for log processing in MaPreduce. In *Proc.of SIGMOD*, pages, 975–986, 2010.

[11] Rares Vernica, Michael J. Carey, Chen Li: Efficient parallel set-similarity joins using MapReduce. In *Proc.of SIGMOD*, pages, 495–506, 2010.

[12] Foto N. Afrati, Vinayak R. Borkar, et. al. Map-reduce extensions and recursive queries. In *Proc.of EDBT*, pages, 1–8, 2011.

[13] J. Dittrich, J. Quian-Ruiz, A. Jindal, et. al. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1): 518–529, 2010.

[14] Eaman Jahani, Michael J. Cafarella, Christopher R: Automatic Optimization for MapReduce Programs. *PVLDB* 4(6): 385–396, 2011.

[15] LinLin Ding, Junchang Xin, Guoren Wang, Shan Huang: ComMapReduce: An Improvement of MapReduce with Lightweight Communication Mechanisms. In *Proc.of DASFAA*, pages, 150–168, 2012.

[16] S. Ghemawat, H. Gobioff, S. T. Leung: The Google file system. In *Proc.of SOSP*, pages, 29–43, 2003.

[17] Junchang Xin, Guoren Wang, Lei Chen, Xiaoyi Zhang, Zhenhua Wang. Continuously Maintaining Sliding Window Skylines in a Sensor Network. In *Proc.of DASFAA*, pages, 509–521, 2007.

[18] Alper Okcan, Mirek Riedewald: Processing theta-joins using MapReduce. In *Proc.of SIGMOD*, pages, 949–960, 2011.

[19] Foto N. Afrati, Jeffrey D. Ullman: Optimizing joins in a map-reduce environment. In *Proc.of EDBT*, pages, 99–110, 2010.

[20] Younghoon Kim, Kyuseok Shim: Parallel Top-K Similarity Join Algorithms Using MapReduce. In *Proc.of ICDE*, pages, 510–521, 2012.

[21] Bloom B. H., Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7): 422–426, 1970.

[22] Fay Chang, Jeffrey Dean, et. al. Bigtable: A Distributed Storage System for Structured Data. In *Proc.of OSDI*, pages, 205–218, 2006.

[23] Michael Burrows: The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc.of OSDI*, pages, 335–350, 2006.

[24] M. Isard, M. Budiu, et. al. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc.of EuroSys*, pages, 59–72, 2007.

[25] T. Nykiel, M. Potamias, et. al. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB* 3(1): 494–505, 2010.

[26] Jun Li, Peng Zhang, et. al. Continuous data stream query in the cloud. In *Proc.of CIKM*, pages, 2389–2392, 2011.

[27] T. Condie, N. Conway, et. al. Online aggregation and continuous query support in MapReduce. In *Proc.of SIGMOD*, pages, 1115–1118, 2010.

[28] Y. Bu, B. Howe, M. Balazinska, et. al. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1): 285–296, 2010.

[29] Chi Zhang, Feifei Li, Jeffrey Jestes: Efficient parallel kNN joins for large data in MapReduce. In *Proc.of EDBT*, pages, 38–49, 2012.

[30] Jie Pan, Yann Le Biannic, Frdric Magouls: Parallelizing multiple group-by query in share-nothing environment: a MapReduce study case. In *Proc.of HPDC*, pages, 856–863, 2010.

[31] Yongluan Zhou, Beng Chin Ooi, et. al. An adaptable distributed query processing architecture. *DKE*, 53(3): 283–309, 2005.

[32] Boliang Zhang, Shuigeng Zhou, Jihong Guan: Adapting Skyline Computation to the MapReduce Framework: Algorithms and Experiments. In *Proc.of DASFAA Workshops*, pages, 403–414, 2011.

[33] Keping Zhao, Yufei Tao, et. al. Efficient top-k processing in large-scaled distributed environments. *DKE*, 63(2): 315–335, 2007.

[34] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. Analysis and evaluation of V*-kNN: an efficient algorithm for moving kNN queries. *The VLDB Journal* 19(3): 307–332, 2010.

[35] Tanzima Hashem, Lars Kulik, and Rui Zhang. Privacy preserving group nearest neighbor queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages, 489–500. ACM, 2010.