Data Cloud for Distributed Data Mining via Pipelined MapReduce

Zhiang Wu, Jie Cao, and Changjian Fang

Jiangsu Provincial Key Laboratory of E-Business,
Nanjing University of Finance and Economics, Nanjing, P.R. China
zawuster@gmail.com, {caojie690929,jselab1999}@163.com

Abstract. Distributed data mining (DDM) which often utilizes autonomous agents is a process to extract globally interesting associations, classifiers, clusters, and other patterns from distributed data. As datasets double in size every year, moving the data repeatedly to distant CPUs brings about high communication cost. In this paper, data cloud is utilized to implement DDM in order to move the data rather than moving computation. MapReduce is a popular programming model for implementing data-centric distributed computing. Firstly, a kind of cloud system architecture for DDM is proposed. Secondly, a modified MapReduce framework called pipelined MapReduce is presented. We select Apriori as a case study and discuss its implementation within MapReduce framework. Several experiments are conducted at last. Experimental results show that with moderate number of map tasks, the execution time of DDM algorithms (i.e., Apriori) can be reduced remarkably. Performance comparison between traditional and our pipelined MapReduce has shown that the map task and reduce task in our pipelined MapReduce can run in a parallel manner, and our pipelined MapReduce greatly decreases the execution time of DDM algorithm. Data cloud is suitable for a multitude of DDM algorithms and can provide significant speedups.

Keywords: distributed data mining (DDM), Cloud Computing, MapReduce, Apriori, Hadoop.

1 Introduction

The last decade has witnessed the successful development of agents and data mining techniques which have been applied to a variety of domains - marketing, weather forecasting, medical diagnosis, and national security [1,2]. As data mining become more pervasive, the amount of data is increasing larger. The great amount of data is often partitioned into many parts and distributed in many sites. Distributed data mining (DDM) is a process to extract globally interesting associations, classifiers, clusters, and other patterns from distributed data, where data can be partitioned into many parts either vertically or horizontally [3,4].

Agents are scatted to many sites for handling distributed problem-solving, cooperation and coordination. A high performance DDM system is designed to

L. Cao et al. (Eds.): ADMI 2011, LNAI 7103, pp. 316–330, 2012.

control agents and exploit the synergy of agents. Traditional DDM system has been designed to take advantage of powerful, but shared pools of CPUs. Generally speaking, data is scattered to the processors, the computation is performed using a message passing, the results are gathered, and the process is repeated by moving new data to the CPUs [5]. As CPU cycles become cheaper and data sets double in size every year, the main challenge for efficient scaling of applications is the location of the data relative to the available computational resources - moving the data repeatedly to distant CPUs is becoming the bottleneck [6].

On the basis of cluster computing, P2P, Grid computing and Web 2.0, cloud computing rapidly emerges as a hot issue in both industrial and academic circles [7]. Cloud computing has been adhering to the belief that moving computation is cheaper than moving data since its birth. Therefore, cloud computing is suitable for solving computation-intensive and data-intensive problems in DDM. MapReduce has emerged as an effective method to implement the *data-centric* belief held by cloud computing. MapReduce has been applied to a multitude of domains for data processing, such as machine learning, satellite data processing, PageRank, scientific data analysis, etc. Since traditional MapReduce stores all middle results in file system and most of DDM applications produce a large amount of middle results, preserving storage for temporary files is extremely expensive and inefficient. Moreover, in traditional MapReduce framework, the reduce task does not start running until all map tasks are finished. This sequential execution manner between map task and reduce task increases the job completion time.

Motivated by the above remarks, this paper proposes a kind of data cloud system architecture for DDM. We make an improvement in the traditional MapReduce framework, namely, pipelined MapReduce framework. The dataflow and the fault tolerance strategy of pipelined MapReduce are addressed in detail. We then discuss the implementation of Apriori algorithm, which is a well-known algorithm for tackling this problem, in MapReduce framework. At last, experimental evaluation of our work is presented.

2 Related Work

2.1 Cloud Computing

In 2007, IBM and Google first proposed cloud computing. Currently, providers such as Amazon, Google, Salesforce, IBM, Microsoft and Sun Microsystems have begun to establish new data centers for hosting cloud computing applications in various locations around the world to provide redundancy and ensure reliability in case of site failures [8]. Data cloud, which is proposed by R. Grossman and Y. Gu in 2008, refers to a class of cloud systems that provide resources and/or data services over the Internet [5].

The Google's MapReduce programming model, which serves for processing large data sets in a massively parallel manner, is a representative technique in cloud computing [9]. Hadoop developed by the Apache Software Foundation is an open source MapReduce framework [10]. The key components of Hadoop are

HDFS (Hadoop Distributed File System) and MapReduce. HDFS is a distributed file system designed to run on hardware, and it also manages all files scattered in nodes and provides high throughput of data access. MapReduce provides a programming model for processing large scale datasets in distributed manner. Jobs submitted to Hadoop consist of a map function and a reduce function. Hadoop breaks each job into multiple tasks. Firstly, map tasks process each block of input (typically 64MB) and produce intermediate results, which are key-value pairs. These are saved to disk. Next, reduce tasks fetch the list of intermediate results associated with each key and run it through the reduce function, which produces output. Hadoop is selected as core middleware in data cloud proposed in this paper.

MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion. In the meanwhile, MapReduce works well on unstructured or semi-structured data, for it is designed to interpret the data at processing time. In other words, the input keys and values for MapReduce are not an intrinsic property of the data, but they are chosen by the programmer. The input of MapReduce job can be either a text format or data stored in Hbase which is a distributed, column-oriented database provided by Hadoop.

Agents' pro-activeness is suitable for collecting information about resource status and databases. Traditional monitoring systems generate alarm when a failure occurs by monitoring resources continuously. The storm of alarm often occurs. Utilizing agent to gather information can avoid "the storm of alarm" and extract valid knowledge from data which is known as data intelligence in [11].

2.2 Mining Association Rules upon Paralleling (MARP)

Association rules mining is an important branch of data mining. Since the mass-data often is often distributed in many sites and putting all data together to amass a centralized database is unrealistic, MARP employs distributed computing technology to implement concurrent data mining algorithms. MARP developments endeavor to scale up data mining algorithms by changing existing sequential techniques into parallel versions. Implementing data mining algorithms within MapReduce framework belongs to MARP. However, parallel algorithms within MapReduce have at least two advantages over traditional MARP techniques.

- Parallel algorithms within MapReduce can process unstructured or semi-structured data conveniently.
- Since MapReduce tries to colocate the data with the compute node, data access of algorithms within MapReduce is fast for it is local.

"Market-Basket Analysis problem", a classic association rules mining problem, is taken as a case study. The supermarket owners may be interested in finding associations among its items purchased together at the check-stand [12]. Apriori is a seminal algorithm for finding frequent itemsets using candidate generation [13]. It is characterized as a level-wise complete search algorithm using

anti-monotony of itemsets, "if an itemset is not frequent, any of its superset is never frequent". Since Apriori is time consuming and the transaction database is currently distributed in many sites, it is necessary to apply parallel and distributed methods to Apriori. However, the major difficulty lies in that computing support of an itemset should scan the whole database, but each node only stores a split of the whole database. To tackle this difficulty, two kinds of methods have been presented. The first method takes pre-treatment on database to decompose the database into several fragments, and then each node could run totally independently on each fragment of database to compute frequent itemsets using a classical sequential algorithm of Apriori. At last, final results are obtained from these nodes. Although V. Fiolet has suggested three fragmentation methods [12], these methods are time consuming and none of them can be applied to all databases. Furthermore, the pre-processing on database leads to restore data, and data should be moved among clusters (or PC servers), which violates the essence of cloud computing-moving computation is cheaper than moving data. The second method is called *CountDistribution* which is incorporated in this paper. In cloud environment, assuming that the database is initially distributed in a horizontal split, namely non-overlapping subsets of records are randomly stored in clusters (or PC servers). Each node can thus independently get partial supports of the candidates from its local database fragmentation. Reducer then executes reduce function to compute the sum of supports with the same key. Note that only the partial counts need to be communicated, rather than the records of the database. Since the partial counts are represented as (*key, value*) pairs in MapReduce, the method can minimize communication.

3 Data Cloud System Architecture for DDM

Based on campus grid environments we have designed and implemented layered data cloud system architecture as Fig. 1 depicted. Physical cloud resources along with core middleware form the basis of the system. The user-level middleware aims to provide PaaS (platform as a service) capabilities. The top layer focuses on application services by making use of services provided by the lower layer services. Emerging DDM applications such as social security, enterprise, stock markets and scientific workflows can operate at the highest layer of the architecture. The representative data mining algorithms used by DDM applications such as Apriori, PageRank, kNN and k-Means can be implemented in data cloud system, and some of them even can be implemented in MapReduce framework to improve their performance.

- System level: Physical resources integrated by data cloud system consist of two clusters each with 32 blade servers and several PC servers. The total number of CPU reaches 140.
- Core middleware: This layer is comprised of two sub-layers: VM (virtual machine) and Hadoop. VM management and deployment transparently virtualizes these servers and shares their capacity among virtual instances of

servers. These VMs are isolated from each other, which aid in achieving fault tolerant behavior and isolated security context. On the top of VMs, Hadoop framework is deployed, on which Java programs based on MapReduce model can be executed. All data are stored on HDFS. The master node called *JobTracker* is the point of interaction where the user submits jobs, along with location of the input data on the HDFS. The *JobTracker* assigns and distributes the map and reduce tasks to the *TaskTrackers* which assumes the role of worker nodes. The *TaskTracker* performs the task and updates the status to the *JobTracker*. In the MapReduce framework of this architecture, pipeline is added between Mapper and Reducer. The new MapReduce framework is called Pipelined MapReduce framework which will be discussed in the next section.

- User-level middleware: This layer provides Web 2.0 programming paradigms such as JavaScript with AJAX, Ruby with Ruby on Rail, and PHP etc. Users can utilize Web APIs to create novel browser-based applications. This layer also provides the programming environments and composition tools that facilitate the creation, deployment, and execution of applications in clouds.
- Security management: This module provides authentication and permission control for cloud users. Single sign-on is adopted by cloud system architecture proposed in this paper. Cloud user obtains long-term certificate from Certificate Authority (CA), and user can encrypt this long-term certificate to generate temporary proxy certificate which is used to identify user by data cloud. The deadline of proxy certificate is often short (i.e. 12 hours in our data cloud) to decrease the harm created by various network attacks.
- Agent-based monitoring system: Agents are deploy to all resources including clusters, servers, software, database and jobs. Agent is a flexible and powerful toolkit for displaying, monitoring and analyzing results to make the best use of the collected data. Status data gathered by agents is stored to Tivoli data warehouse and is displayed in portal. Since virtualization technology is used in data cloud, the relation between agent and resource is many-to-one.

4 Pipelined MapReduce Framework

In the data cloud system proposed in this paper, we make an improvement in the traditional MapReduce framework, namely, Pipelined MapReduce Framework. In traditional MapReduce framework, the output of each Mapper is managed by the *OutputCollector* instance and stored in an in-memory buffer. The *OutputCollector* is also responsible for spilling this buffer to disk (i.e., HDFS in Hadoop) when the output reaches capacity of memory. The execution of a reduce task includes three phases: *shuffle* phase, *sort* phase and *reduce* phase. The *shuffle* phase fetches intermediate results from each Mapper. However, *a reducer cannot fetch the output of a mapper until JobTracker informs that the mapper is finished*. The output produced by Reducers may be required by the next map step, which is also written to HDFS. There are at least two disadvantages within the traditional MapReduce framework.

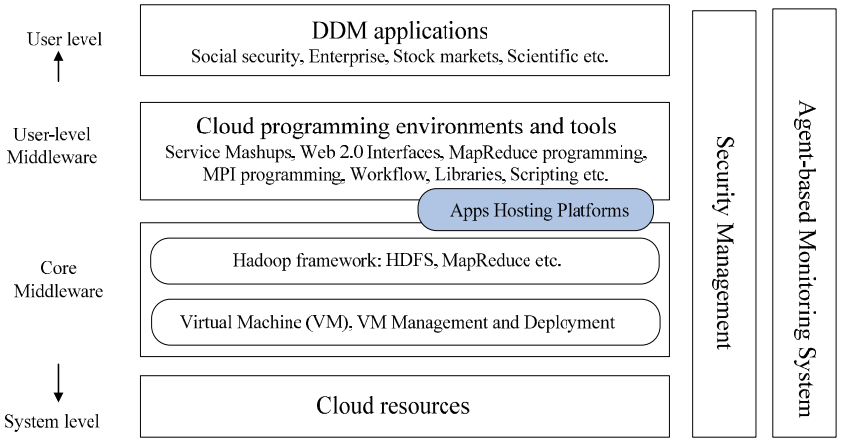

Fig. 1. Data cloud system architecture for DDM

- HDFS should maintain enough storage for temporary files. Since each file has three copies in HDFS in default manner and most of DDM applications will produce a large amount of middle results, preserving storage for temporary files is extremely expensive and inefficient.
- Because Mapper and Reducer execute in a serial manner, and both Mappers and Reducers should spend plenty of time in reading middle data from HDFS, the execution speed of the traditional MapReduce framework is very slow.

To solve these above-mentioned disadvantages, pipeline is added between Mapper and Reducer. Middle data is stored in pipeline files and only final results are written to HDFS. Moreover, when the Mapper is executing, it simultaneously pushes middle data to Reducer via pipeline. Reducers then pull the middle data synchronously. Pipelined MapReduce Framework makes Mappers and Reducers execute in a parallel manner and also enhance the robustness of the fault tolerance.

4.1 The Dataflow of the Pipelined MapReduce Framework

Fig. 2 compares the dataflow between traditional and pipelined MapReduce framework. The dataflow on the right depicts the approach utilized by the pipelined MapReduce framework. Each Mapper obtains its input data from HDFS, and when the Mapper is executing, it simultaneously pushes middle data to pipeline. It is noted that when the intermediate data exceeds the memory size, the intermediate data should also be written to HDFS. Each Reducer synchronously pulls data from pipeline and starts running. The middle data

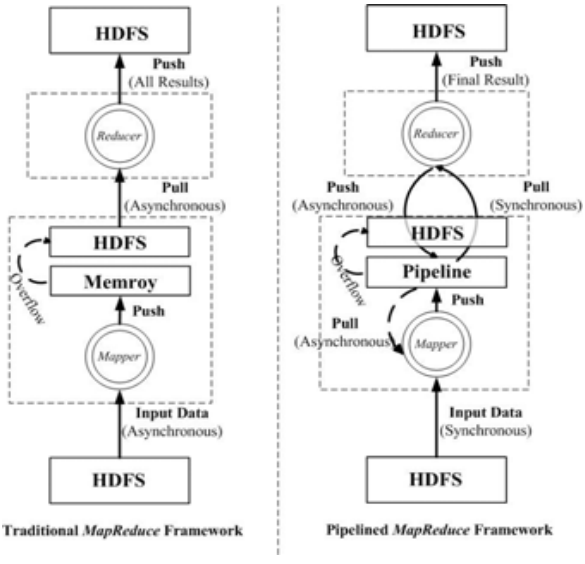

Fig. 2. The dataflow comparison between traditional and pipelined MapReduce framework

produced by the Reducer, which may be required by the next map step, is also pushed to pipeline. Final result produced by the Reducer is written to HDFS.

The precondition that reducers and mappers can run concurrently is that the reduce function of the application is incrementally computable. The applications exist commonly, such as sorting, graph algorithms, Bayes classification, TF-IDF (Term Frequency - Inverse Document Frequency), Apriori, and so on. The pipeline between Mapper and Reducer is implemented through utilizing TCP socket. Each Reducer contacts every Mapper upon initiation of the job, and opens a socket which will be used to pipeline the output of the map function. As each map output is produced, the Mapper determines which partition the record should be sent to, and immediately sends it via the appropriate socket. A Reducer accepts the pipelined data that it has received from each Mapper and then stores it an in-memory buffer. The Reducer may start running on the basis of these partial data. Once the Reducer learns that all Mappers have completed, it continues to perform the remaining task and writes the output to pipeline or to HDFS. One practical problem in the above-mentioned method is that when the number of mappers becomes large, each reducer should maintain a large number of TCP connections. To reduce the number of concurrent TCP connections, we restrict the number of connections with mappers at once. The reducer obtains data from the remaining map tasks in the traditional Hadoop manner.

4.2 The Fault Tolerance Strategy of the Pipelined MapReduce Framework

It is known that the role of cluster node is divided into *master* and *worker*. Both mappers and reducers are *worker* nodes. Although Hadoop could handle master failure, the case unlikely exists [14]. The *master* detects worker failure via periodic heartbeats. New fault tolerance strategy is implemented in the pipelined MapReduce framework. The reducer treats the output of a pipelined map task as "tentative" until the *JobTracker* informs the reducer that the mapper has committed successfully. The reducer merges together spill files generated by the same uncommitted mapper. Log is added to reducer to record which mapper produced each pipelined spill file. Thus, if a mapper fails, each reducer can ignore any tentative spill file produced by the failed map attempt. The *JobTracker* will then restart a new mapper. If a Reducer fails and a new copy of the task is started, all the input data that was sent to the failed reducer must be sent to the new reducer. To prevent mappers from discarding their output after sending it to pipeline, mappers should retain their output data until the entire job is completed successfully. This allows the output of mappers to be reproduced if any reducer fails.

5 Case Study

Finding frequent itemsets from a transaction database and deriving association rules is called "Market-Basket Analysis" problem which is one of the most popular data mining problems. This section takes "Market-Basket Analysis" problem as a case study. Although "Market-Basket Analysis" problem has been described in many papers, to make it clear and easy for understanding, we briefly describe it. Given a transaction database D, the number of its records is denoted as $|D|$. We assume there are n distinct items in D, denoted as I=$\{i_1, i_2, , i_n\}$.

Remark 1. The support of an itemset X is the percentage of records in the database D, which contains this itemset X. The support measures how interesting the itemset is, that is, its frequency in the database. The support of the itemset X can be calculated by equation (1).

$$support(X) = \frac{D(X)}{|D|} \tag{1}$$

where D(X) is the number of records containing itemset X in database D, and $|D|$ is the total number of records of database D. The "Market-Basket Analysis" problem is to find out all association rules whose support is greater than the given thresholds. The support threshold is defined by user.

5.1 Apriori Algorithm within the Framework of MapReduce

A well-known algorithm for the computation of frequent itemsets is the Apriori algorithm which is used as follows:

- to compute the supports of items, and then to identify frequent items (frequent 1-itemsets)
- to generate candidate 2-itemsets, to count their supports, and then to identify frequent 2-itemsets
- " to generate candidate 3-itemsets, to count their supports, and then to identify frequent 3-itemsets, and so on ...

To compact the search space of Apriori, the guiding principle "every subset of a frequent itemset has to be frequent" is utilized.

As section 2 described, this paper incorporates *CountDistribution* method to implement Apriori within the framework of MapReduce. In cloud environment, assuming that the database is initially distributed in a horizontal split, namely non-overlapping subsets of records are randomly stored in clusters (or PC servers). These nodes which store the splits of the database execute map function and become *Mappers*. A program implementing the map function of Apriori is sent to all mappers. Each mapper can thus independently get partial supports of the candidates from its local database fragmentation. All partial results from mappers are collected to one or several nodes called *Reducer(s)*. And then, Reducer executes reduce function to compute the sum of supports with the same key and global frequent itemsets $F_k - 1$ are obtained. Note that only the partial counts need to be communicated, rather than the records of the database. Since the partial counts are represented as *(key, value)* pairs in MapReduce, this method can minimize communication. Fig. 3 depicts the MapReduce framework of Apriori.

Since Apriori algorithm utilizes iteration to obtain all frequent itemsets, it needs to scan database at most *n+1* times when the maximum size of frequent itemsets is set at F_n. Therefore, *mappers* should be started at most *n+1* times. Within the framework of MapReduce, though reduce operation also can be run in parallel manner, *reducer* of Apriori is run in a single node because the reduce operation of Apriori only needs to carry out simple count operation. The pseudocode of map and reduce function executed by *mappers* and *reducer* respectively for the Apriori is as follows.

The map function for the Apriori

```
Input:the split of database,
      and the last global frequent itemset Fk-1
Output: (key, value)pairs
1: Ck := AprioriGen (Fk-1)
2: for each element c in Ck
3:  scan the split of database and count support for c
4:  generate pair (c,c.support)
5: end for
6: return Union((c,c.support))
```

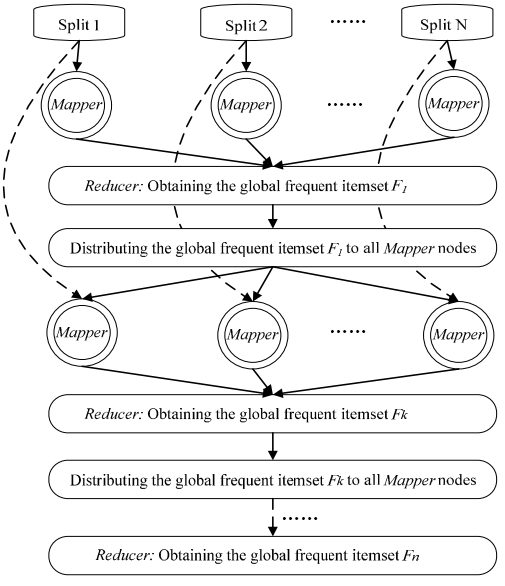

Fig. 3. MapReduce framework of Apriori

The reduce function for the Apriori

```
Input: (c, c.support) pairs generated by Mapper nodes
Output: the global frequent itemset Fk
1: for all (c, c.support) pairs with same c
2:  compute the sum of their support
3:  denote the sum as (c, sum_support)
4: end for
5: Fk := {all (c, sum_support) pairs | c.sum_supportmin_sup}
6: return Union(Fk)
```

In the Map function, AprioriGen function generates new candidates denoted as C_k based on the last global frequent itemset $F_k - 1$. Each mapper scans its split of database and counts support for all elements in C_k. In the Reduce function, all (key, value) pairs from all mappers are grouped by key and supports of the pairs with the same key are added. The pairs whose sum of support is greater than threshold are selected into the current global frequent itemset F_k.

This case study indicates that the MapReduce framework provided by data cloud system architecture exhibits good scalability to other DDM algorithms. We needn't take any pre-treatment on database and needn't move data among clusters. Only map function and reduce function of DDM algorithms need to be provided. Data cloud can automatically move programs containing map function to a multitude of nodes storing the input data.

5.2 Performance Evaluation

We have built a data cloud system based on the architecture shown in Fig. 1. Machines integrated by this data cloud system consist of two clusters each with 32 blade servers in SEUGrid (Southeast University Grid) and an army of PC servers in our laboratory. Two VMs are created in each PC servers with 2.03GH and 1GB of RAM. Two clusters in SEUGrid utilize Red Hat Enterprise 4.0 and all VMs in PC servers utilize Fedora Core 11, and on the top of Linux, Hadoop 0.20.1 is installed and configured as core middleware. The transaction database contains two attributes: one is the transaction identifier and the other is the list of items. In the first experiment, we investigate the effect of the number of *mappers* on the execution time to obtain all frequent itemsets utilizing Apriori. A transaction database is created with 100,000 records and 500 kinds of items. The transaction database is split on average according to the number of *Mapper* nodes. In other words, equal number of records is stored in all *mapper* nodes. Fig. 4 shows the effect of number of *mappers* on execution time. It indicates that with the increase of the number of *mappers*, the execution time decreases. At the beginning, the execution time decreases dramatically with the increase of the number of *mappers*. When the number of *mappers* reaches a threshold (i.e. 16 in Fig. 4), the execution time varies moderately or even increases slightly (i.e. 24, 28 and 32 in Fig. 4). The reason for the above-mentioned phenomenon is that with the increase of *mappers*, the number of records stored in each *mapper* node decreases. When the number of records reduce to a threshold, the time of scanning the database is hard to further decline. Therefore, the time of *Reducer* and communications is rising, thus playing the major role in the whole execution time.

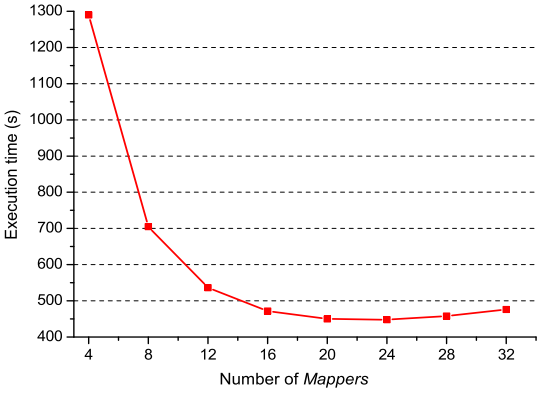

Fig. 4. Effect of number of Mappers on execution time

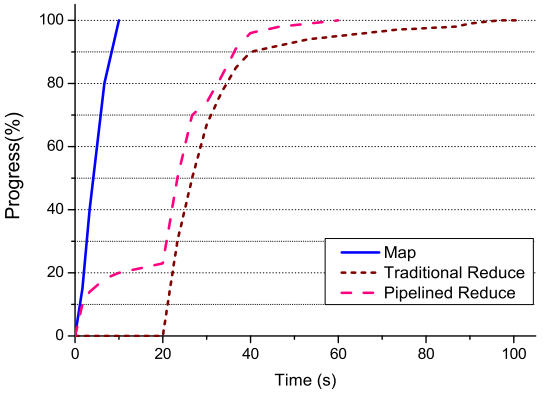

Fig. 5. Comparison of map and reduce task completion times between traditional and pipelined MapReduce

The second experiment compares pipelined MapReduce with traditional MapReduce. We analyze map progress and reduce progress in an iteration of Apriori (i.e. computing F_k based on $F_k - 1$). Hadoop provides support for monitoring the progress of task executions. As each task executes, it is assigned a *progress score* in the range $[0, 1]$, based on how much of its input that the task has consumed. There are 80,000 records split on 8 mappers including 500 kinds of items. The first iteration computing F1 based on the input from HDFS is selected as a statistical object. Fig. 5 describes the progress score of map task and reduce task along with the timelapse. The map tasks in both traditional and pipelined MapReduce are same for their input is read from HDFS. There exists obviously difference between the progress of traditional reduce and the progress of pipelined reduce. Traditional reduce task starts after the map task and it consumes more time than pipelined reduce task. However, pipelined reduce task starts immediately after the map task starts, and they are running in a parallel manner (i.e. from 0s to 10s). These results suggest that pipelined MapReduce framework can substantially reduce the response time of a job. In the third experiment, we implement other applications in Hadoop and investigate the speedup as we scale the number of processor cores. The following are brief descriptions of the selected applications:

- WordCount: It counts the frequency of occurrence for each word in a set of files. Mappers process different sections of the input files and return intermediate data that consist of a word (key) and a value of 1 to indicate that the word was found. The reducers add up the values for each word (key).
- Kmeans: It implements the popular Kmeans algorithm to group a set of input data points into clusters. Since Kmeans is iterative, in each iteration, mappers find the distance between each point and each mean and assign the point to the closest cluster. For each point, we emit the cluster ID as the key

and the data vector as the value. Reducers gather all points with the same cluster ID, and finds their mean vector.
- Reverse Index: It traverses a set of HTML files, extracts all links, and compiles an index from links to files. Each mapper parses a collection of HTML files. For each link it finds, it outputs an intermediate pair with the link as the key and the file info as the value. The reducer combines all files referencing the same link into a single linked-list.

We evaluate the speedup of four algorithms as we scale the number of processor cores used in data cloud. The concept of speedup stems from high-performance computing (HPC). *Speedup* is defined as the ratio of the execution time of an algorithm on one node to the execution time of the same algorithm on several nodes. Fig. 6 presents the experimental result. Data cloud provides significant speedups for all processor counts and all algorithms. With a large number of cores, the speedup of some applications cannot be improved further (i.e., WordCount, Kmeans, Apriori), where the reason is similar to the analysis in the first experiment. The speedups of Kmeans and Apriori are smaller than other algorithms, because they contain iteration process. Fitting iterative algorithms into the MapReduce framework leads to major overheads compared to sequential code and reduces the overall speedup.

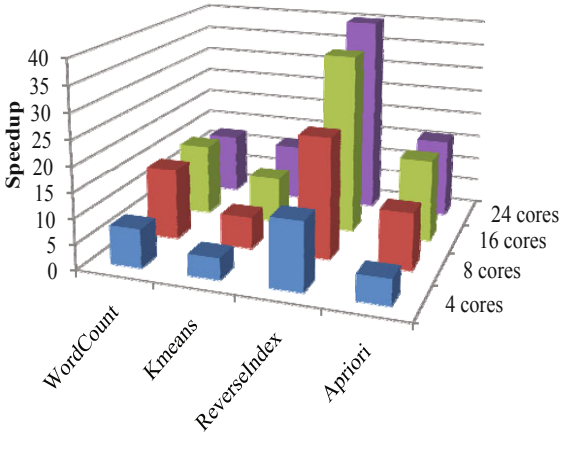

Fig. 6. Speedup of four algorithms with increase of processor cores

6 Conclusion

The increasing amount of data has led to concerns regarding the execution efficiency of DDM. The *data-centric* belief held by cloud computing can be utilized to enhance the execution efficiency of DDM. In this paper, we propose a kind of data cloud system architecture for DDM. Pipelined MapReduce framework is presented and implemented in data cloud. We then take Apriori algorithm as

a case study and implement Apriori within the framework of MapReduce. This case study indicates that data cloud system architecture proposed in this paper exhibits good scalability to various DDM algorithms. The fragmentation (or pretreatment) methods of database needn't to be considered. Users should only provide map function and reduce function of DDM algorithms to data cloud. Data cloud can move computation in terms of the initial data location. We conduct several experiments in order to evaluate our work. The experimental results indicate that the execution time can be reduced remarkably with moderate number of mappers. Performance comparison between traditional MapReduce and our pipelined MapReduce has shown that: (1) the map task and reduce task in our pipelined MapReduce can run in a parallel manner; (2) our pipelined MapReduce greatly decreases the execution time of DDM algorithm. Data cloud is suitable for a multitude of DDM algorithms and can provide significant speedups.

Acknowledgments. This research is supported by National Natural Science Foundation of China under Grants No.71072172, the program for New Century Excellent Talents in university under Grants No.NCET-07-0411, Jiangsu Provincial Key Laboratory of Network and Information Security (Southeast University) under Grants No. BM2003201, Transformation Fund for Agricultural Science and Technology Achievements under Grants No. 2011GB2C100024 and Innovation Fund for Agricultural Science and Technology in Jiangsu under Grants No. CX(11)3039.

References

1. Cao, L., Gorodetsky, V., Mitkas, P.A.: Agent Mining: The Synergy of Agents and Data Mining. IEEE Intelligent Systems 24(3), 64–72 (2009)
2. Pech, S., Goehner, P.: Multi-agent Information Retrieval in Heterogeneous Industrial Automation Environments. In: Cao, L., Bazzan, A.L.C., Gorodetsky, V., Mitkas, P.A., Weiss, G., Yu, P.S. (eds.) ADMI 2010. LNCS, vol. 5980, pp. 27–39. Springer, Heidelberg (2010)
3. Yi, X., Zhang, Y.: Privacy-preserving naïve Bayes classification on distributed data via semi-trusted mixers. Information Systems 34(3), 371–380 (2009)
4. Cao, L.: Domain-Driven Data Mining: Challenges and Prospects. IEEE Transactions on Knowledge and Data Engineering 22(6), 755–769 (2010)
5. Grossman, R., Gu, Y.: Data mining using high performance data clouds: experimental studies using sector and sphere. In: Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 920–927 (2008)
6. Szalay, A., Bunn, A., Gray, J., Foster, I., Raicu, I.: The Importance of Data Locality in Distributed Computing Applications. In: NSF Workflow Workshop (2006)
7. Above the clouds: A Berkeley View of Cloud computing. UCB/EECS-2009-28 (2009)
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599–616 (2009)

9. Ralf, L.: Google's MapReduce programming model - Revisited. The Journal of Science of Computer Programming 70(1), 1–30 (2008)
10. Hadoop: The Apache Software Foundation, http://hadoop.apache.org/core
11. Cao, L., Luo, D., Zhang, C.: Ubiquitous Intelligence in Agent Mining. In: Cao, L., Gorodetsky, V., Liu, J., Weiss, G., Yu, P.S. (eds.) ADMI 2009. LNCS, vol. 5680, pp. 23–35. Springer, Heidelberg (2009)
12. Fiolet, V., Toursel, B.: Distributed Data Mining. Scalable Computing: Practice and Experience 6(1), 99–109 (2005)
13. Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., Mclachlan, G.J., Ng, A., Liu, B., Yu, P.S., Zhou, Z., Steinbach, M., Hand, D.J., Steinberg, D.: Top 10 algorithms in data mining. Knowledge and Information Systems 14(1), 1–37 (2008)
14. Hadoop, W.T.: The Definitive Guide. O' Reilly Publishers (2010)