PSON: A Parallelized SON Algorithm with MapReduce for Mining Frequent Sets

Tao Xiao Chunfeng Yuan Yihua Huang
Department of Computer Science and Technology
Nanjing University
Nanjing, China
xiaotao.cs.nju@stu.nju.edu.cn, cfyuan@nju.edu.cn, yhuang@nju.edu.cn

Abstract—**Many algorithms have been proposed in past decades to efficiently mine frequent sets in transaction database, including the SON Algorithm proposed by Savasere, Omiecinski and Navathe. This paper introduces the SON algorithm, explains why SON is very suitable to be parallelized, and illustrates how to adapt SON to the MapReduce paradigm. Then we propose a parallelized SON algorithm, PSON, and implement it in Hadoop. Our study suggests that PSON can mine frequent itemsets from a very large database with good performance. The experimental results show that when performing frequent sets mining, the time cost will increase almost linearly with the size of the datasets and decrease with approximately linear trend with the number of cluster nodes. Consequently, we conclude that PSON works well for solving the frequent set mining problem from massive datasets with a good performance in both scalability and speed-up.**

Keywords-frequent sets mining; parallelized SON algorithm; MapReduce; Hadoop

I. INTRODUCTION

Finding frequent sets in transaction database is fundamental and important in data mining domains. Many algorithms have been proposed in past decades to efficiently achieve such a goal [1, 2, 3, 4]. However, most algorithms were designed for serial computing environment and cannot fit into huge databases such as Amazon's sales database that can reach a scale of TBs or even PBs easily. The partitioning algorithm [1] (SON Algorithm) proposed by Savasere et al. is one of the serial algorithms for mining frequent sets. The SON algorithm is fundamentally different from other traditional frequent sets mining algorithms and has been proved to be able to reduce the I/O overhead significantly and lower CPU overhead in most cases. It is especially suitable for very large databases [1]. The key to the success of the SON algorithm is that it divides the entire database into several non-overlapping partitions and handles them separately, with some work handling the results from all the partitions altogether. Such a feature of the SON algorithm makes it naturally suitable to run with MapReduce, a parallel computing paradigm proposed by Google [5, 6].

In this paper, we propose a parallelized version of the SON algorithm, PSON, implemented with MapReduce in Hadoop, an open-source software framework developed by Apache [7, 8].

The rest of this paper is organized as follows. Section 2 briefly describes the concepts of frequent sets mining, Apriori algorithm, distributed file system, and MapReduce. Section 3 introduces the SON algorithm. Section 4 explains why the SON algorithm is suitable for being adapted to a MapReduce paradigm and describes how to parallelize it with MapReduce in details. Section 5 introduces our implementation of the PSON algorithm in Hadoop and evaluates the performance in scalability and speed-up. Section 6 concludes by discussing some issues and our future work.

II. BASIC CONCEPTS

A. Frequent Sets Mining

Let $\boldsymbol{I} = \{i_1, i_2, ..., i_n\}$ be a set of n distinct literals called *items* and $\mathcal{D}$ a set of transactions with variable lengths over $\boldsymbol{I}$, where each transaction in $\mathcal{D}$ is composed of a unique transaction id called *TID* and a set of items $i_j, i_k, ..., i_m$. A set of items is called an *itemset* and the number of the items in an itemset is called the *length* of an itemset.

Ashok et al. [1] defines the *support* of an itemset and *frequent itemset*. The *support* of a k-itemset I_k, which contains k distinct items, is the fraction of the transactions in $\mathcal{D}$ containing all of k items in I_k. If the support of I_k is equal to or above a user defined support threshold s, then I_k is called a *frequent k-itemset* in $\mathcal{D}$.

In general the transactions are assumed to be in the form $<TID, i_j, i_k, ..., i_m>$ and the items in each transaction are kept in a lexicographically-sorted order. Considering the nature of the application, the *TID*s can also be assumed to be monotonically increasing.

The goal of frequent sets mining is to find all frequent k-itemsets ($k = 1, 2, ...$) in $\mathcal{D}$.

B. Apriori Algorithm

Apriori algorithm was proposed as a classic algorithm for mining frequent sets in transaction database [3]. Here we focus on how it generates all frequent itemsets. In Apriori algorithm, multiple passes over the data are made to discover frequent itemsets. In the first pass, we count the support of individual items and determine which of them are frequent, i.e., with the supports that are no less than the user defined support threshold. In each subsequent pass, we start with a seed set of itemsets found to be frequent in the previous pass. This seed set is used for generating new potentially frequent itemsets, called *candidate itemsets*, and then we count the actual support for these candidate itemsets during the pass over the data. At the end of the pass, we determine which of

978-0-7695-4575-2/11 $26.00 © 2011 IEEE
DOI 10.1109/PAAP.2011.38

the candidate itemsets are actually frequent and further use the determined frequent sets as the seed for the next pass. This process continues until no new frequent itemsets are found.

C. MapReduce and DFS

MapReduce is a patented software framework introduced by *Google* in 2004 to support distributed computing on large data sets on clusters of computers [11]. In a MapReduce application, only two kinds of functions need to be written, called *Map* and *Reduce*, while the system manages the parallel execution, coordinates tasks that execute Map and Reduce, and also deals with the possibility that one of these tasks would fail to execute [4].

To support MapReduce computing, a *distributed file system* (*DFS*) is adopted to store massive amount of data across the cluster. In DFS, files are divided into chunks that are distributed across the cluster. Each chunk is duplicated multiple times at different nodes, typically three times, to provide reliability.

A typical MapReduce job is executed logically as follows:

- One chunk in DFS is fed in some user specified *key-value* pair format to one of the computing nodes called *mappers*, then results are emitted, probably in some different key-value pair format, to the framework.
- The framework gathers all the pairs and sorts them by their keys, then these pairs are sent to several computing nodes called *reducers*.
- Each reducer processes the pairs it receives and writes the results in key-value pair format, probably in some different format, to files in DFS.

Hadoop [7, 8] is an open-source implementation of Google MapReduce and is maintained by the Apache Software Foundation, which implements almost all the utilities needed for computing in a MapReduce paradigm, such as data redundancy, data splitting, task distribution and coordination, failure detection and recovery, etc. In Hadoop, *HDFS* (Hadoop Distributed File System) is selected as the underlying file system, which provides all the utilities a typical DFS should provide [11, 14].

III. SON ALGORITHM

A. Idea

In terms of the definition of frequent itemset, to determine all frequent itemsets, all combinations of items need to be generated and checked. If $|\boldsymbol{I}| = m$, the number of possible distinct itemsets is up to 2^m. Thus the time complexity for performing this process is $O(2^m)$. To reduce the combinatorial search space, the Apriori algorithm takes use of the following property: any subset of a frequent itemset must be frequent, and on contrary, any superset of a non-frequent itemset cannot be frequent.

This property is used by almost all the frequent sets mining algorithms to reduce the combinatorial search space, including the Apriori algorithm, which is the underlying algorithm for the SON algorithm.

A partitioning algorithm (or SON algorithm) in [1] was proposed to mine frequent itemsets in large database, which was proved to be able to reduce the I/O overhead significantly and lower the CPU overhead in most cases.

As described in [1], the SON algorithm can find all frequent itemsets in two scans of the database. In the first scan, it generates a set of all potentially frequent itemsets, which is a superset of all actual frequent itemsets and may contain false positives (an itemset that is frequent in a partition but not frequent in the entire database is called a *false positive*). No false positives are needed to be reported. Counters for each of the potentially frequent itemsets are set up and their actual supports are counted through the second scan of the database, in this way all actual frequent itemsets can be identified.

The SON algorithm can be executed in two phases: In phase I, the entire database will be divided into several non-overlapping partitions. One partition will be processed at a time to generate all frequent itemsets for the partition. At the end of phase I, all frequent itemsets are merged from all partitions to generate a set of all potentially frequent itemsets for the entire database and such a process is called *merge phase*. In phase II, the actual supports for these itemsets are generated and the actual frequent itemsets are identified. The right size of partitions will be chosen so that each partition can be cached in the main memory to speed up the processing during each phase.

B. Definition

A partition p belongs to the database $\mathcal{D}$ and can refer to any subset of the transactions contained in the database $\mathcal{D}$. Any two partitions are non-overlapping, i.e., $p_i \cap p_j = \emptyset$, where $i \neq j$. We can define *local support* for an itemset as the fraction of transactions containing that itemset in a partition. A *local frequent itemset* is an itemset whose support in a partition is no less than the user defined support threshold. A local frequent itemset may or may not be frequent in the context of the entire database. *Global support*, *global candidate itemset* and *global frequent itemset* are defined as above except they are in the context of the whole database. Our goal is to find all global frequent itemsets. Other notations used are shown in table 1.

Table 1: Notations

C_k^p	Set of local candidate *k*-itemsets in partition *p*
L_k^p	Set of local frequent *k*-itemsets in partition *p*
L^p	Set of all local frequent itemsets in partition *p*
C_k^G	Set of global candidate *k*-itemsets
C^G	Set of all global candidate itemsets
L_k^G	Set of global frequent *k*-itemsets

C. Algorithm in Details

The SON algorithm runs in three phases. Initially the database $\mathcal{D}$ is logically divided into n non-overlapping partitions. Phase I processes one of these partitions at a time, in each of n iterations. When processing partition p_i, the function **gen_freq_itemsets** takes p_i as input and generates frequent itemsets of all lengths as the output. In the merge phase the local frequent itemsets of same lengths from all n partitions are combined to generate the global candidate itemsets. In phase II, the algorithm sets up counters for each global candidate itemset and counts their supports for the entire database and thus identifies the global frequent itemsets.

The SON algorithm is shown in pseudo code as follows:

```
1)  $P \leftarrow$ partition_database($D$)
2)  $n \leftarrow$ number of partitions
3)  for $i \leftarrow 1$ to $n$ begin        // Phase I
4)          read_in_partition ($p_i \in P$)
5)          $L^i \leftarrow$ gen_freq_itemsets($p_i$)
6)  end
    // Merge Phase
7)  for ($i \leftarrow 2$; $L_i^j \neq \emptyset, j = 1, 2, 3, ..., n$; $i$++)
8)          do $C_i^G \leftarrow \bigcup_{j=1,2,...,n} L_i^j$ ;
9)  for $i \leftarrow 1$ to $n$ begin        // Phase II
10)         read_in_partition($p_i \in P$)
11)         foreach candidate $c \in C^G$ do
12)                 gen_count($c, p_i$)
13) end
14) $L^G \leftarrow \{c \in C^G \mid c.\text{count} \geqslant minSup\}$
```

Actually, the SON algorithm is based upon such a lemma: if a k-itemset I_k is global frequent, then it must appear as local frequent in at least one of these n partitions; or, if I_k is not local frequent in any of the n partitions, then it cannot be global frequent either. It is easy to prove the correctness of this lemma.

Now we will briefly introduce some important functions used in the SON algorithm.

1) Generation of local frequent itemsets

The procedure *gen_freq_itemsets* takes a partition and generates all frequent itemsets of all lengths for that partition. This procedure contains a sub-procedure, the prune step, which eliminates some extensions of (k-1)-itemsets that cannot be frequent from being considered for counting supports. This is easy to understand: if a (k-1)-itemset I_{k-1} is not frequent, then any k-itemset I_k containing I_{k-1} cannot be frequent, so counting support for I_k is not necessary.

2) Generation of global frequent itemsets

Since we have generated all local frequent k-itemsets for each partition, now we can generate global candidate (k+1)-itemsets by unioning all k-itemsets from these local frequent k-itemsets as in the merge phase, and identify later the global frequent (k+1)-itemsets as in phase II. Phase II also takes n (the number of partitions) iterations and each iteration processes one partition. For each partition p, the local support for a candidate itemset in p is generated. Adding all the local counts for that itemset from all the partitions gives the global count for that itemset in the context of the entire database.

IV. From SON to PSON

A. Motivations to Parallelize SON

Recall what happens in the SON algorithm: In phase I, the whole transaction database is divided into n non-overlapping partitions, and then each of the n partitions is processed for generating local frequent itemsets for that partition. Since these n partitions are non-overlapping and generating local frequent itemsets for any one partition needs no information or data from any other partitions, phase I can be executed by several nodes in parallel, i.e., these n partitions can be processed by n individual mapper nodes in parallel.

In the merge phase, all local frequent itemsets of the same lengths are combined to form global candidate itemsets, such a process perfectly fits into the shuffle and sort phase in MapReduce.

In phase II, each global candidate itemset generated from the merge phase is considered for counting its number of occurrences in each of the n non-overlapping partitions. Such a process can also be parallelized in a MapReduce paradigm, i.e., the n partitions can be considered for counting the number of occurrences of each global candidate itemset by n individual reducer nodes in parallel.

B. Design in Details

In our design, the entire transaction database is stored in *DFS*, where the entire database can be automatically divided into several chunks, corresponding to partitions in the SON Algorithm. Each chunk can be processed by individual nodes in parallel (limited by the number of available nodes) to generate local frequent itemsets for that chunk, and all local frequent itemsets from each chunk of same lengths are combined to form global candidate itemsets. Recall that the whole database is already distributed as many non-overlapping partitions by DFS, therefore for each global candidate itemset, its number of occurrences in each chunk can be computed in parallel. Adding up all the numbers of occurrences of that global frequent candidate itemset from each partition gives its number of occurrences in the entire database, thus we can identify whether it is global frequent. We summarize this map-reduce sequence as follows:

We run two MapReduce jobs to generate all global frequent itemsets. In the first job, all global candidate itemsets are generated; in the second job, actual global frequent itemsets are identified from those global candidate itemsets generated by the first job. We assume that before the two MapReduce jobs are started, the entire transaction database is already stored in the DFS and thus divided into some number of partitions. The details of these two MapReduce jobs will be described as follows:

1) The first MapReduce job

Each mapper node takes one partition of the database and generates all local frequent itemsets for that partition using the algorithm of Section III.*C*. Then a set of key-value pairs likes <*F*, 1> is emitted, where *F* is a local frequent itemset for that partition. The value is always 1 and is useless, only indicating that *F* is a frequent itemset for that partition.

Because of the MapReduce framework's shuffle and sort utility, pairs with the same key, i.e., the local frequent itemset itself, are always sent to the same reducer node. Each reducer node emits to a file in DFS one and only one key-value pair like <*F*, 1> for an itemset *F* it receives. Merging the pairs in all these files gives us the global candidate itemsets.

Figure 1 illustrates the first MapReduce process.

2) The second MapReduce job

Before the second MapReduce job is started, each node is given a full duplicate of the global candidate itemsets generated by the first MapReduce job, which is feasible considering its relatively small size. Thus each mapper node has a partition of the database and a full duplicate of the global candidate itemsets. Each mapper node counts the number of occurrence of each global candidate itemset in the partition the mapper is assigned and then emits a set of key-value pairs like <*C*, *v*>, where *C* is one of the global candidate sets and *v* is the number of occurrences of *C* in that partition.

Each reducer node receives some global candidate itemsets as keys and sums the associated values for each of them. The result is the number of occurrences of these global candidate itemsets in the entire database. Since the number of transactions in the entire database can be known easily, it is easy for each reducer node to identify which of these global candidate itemsets it receives are actual frequent. Only the actual frequent itemsets are emitted out finally.

Figure 2 illustrates the second MapReduce job process.

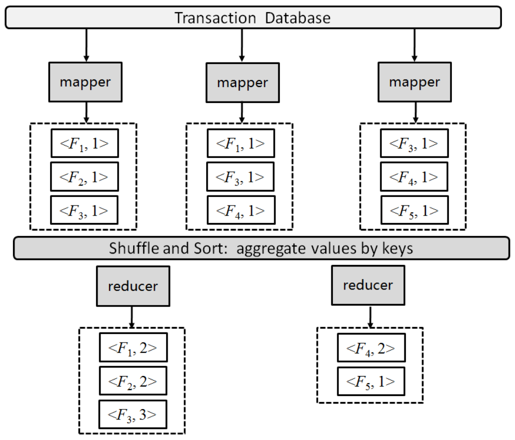

Figure 1: The first MapReduce job process

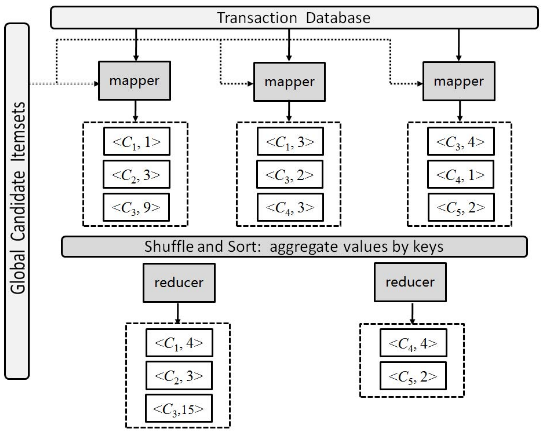

Figure 2: The second MapReduce job process

V. IMPROVEMENT AND OPTIMIZATION

The section above describes the main steps of how to implement the PSON algorithm in a MapReduce paradigm without implementation details. Actually, some factors can have significant impacts on the performance of PSON, for example, how to choose the number of partitions, how to prune the global candidate itemsets and what buffer strategy to adopt to achieve a tradeoff between I/O and performance [1].

1) Choosing the number of partitions

The number of non-global-frequent itemsets contained in the global candidate itemsets should be as small as possible. Otherwise much effort would be wasted in counting global supports for them. As the number of partitions increases the number of false candidates also increases and hence the global candidate size also increases. The chunk size of transaction database in DFS, such as *HDFS*, can be set manually, thus the number of partitions can reach an optimal value through setting an appropriate chunk size. This partially depends on the characteristics of the transaction database itself and we have not found a generally optimal answer to all kinds of database.

2) Counting for global candidate itemsets

According to our design, for each of the global candidate itemsets, its number of occurrences in each partition should be counted during Phase II. Actually when an itemset appears as local frequent in all partitions in Phase I, we can determine whether it is global frequent immediately, since we can get its number of occurrences in the entire database. Of course, we need to adjust our design slightly, e.g., make the mapper node emit each local frequent itemset with its number of occurrences in that partition, instead of only an integer of 1.

3) Buffer strategy

It is ideal that the size of main memory of each node is large enough so that each partition can be handled in main memory without any additional I/O. This can be achieved easily by dividing the entire database into more partitions such that each partition is small enough to be handled in the main memory. However, as described as above, too many partitions can lead to too many false candidates which will waste efforts for counting. A solution is to set the partitions size to an intermediate level and resort to some DFS I/O as buffer strategy when the entire partition cannot be handled in the main memory, which has been proved to be a good tradeoff by our experiments.

VI. Implementation and Experimental Results

We have implemented our PSON algorithm in Hadoop 0.21.0, where *Hadoop* (short for *Apache Hadoop*) is a software framework that supports data-intensive distributed applications under a free license. Inspired by Google's MapReduce and *Google File System* (*GFS*) paper [12], it enables applications to work with thousands of nodes and petabytes of data [11]. Actually it implements almost all the utilities described in MapReduce. Our PSON algorithm runs on a cluster consisting of 21 nodes, where each node has two quad-core processors running at 2.8 GHz, 24 GB memory and a 4TB disk. Two of these nodes act as namenodes, which only do some administrative and coordinating work, while the remaining 19 nodes perform the role of computing and storing data.

We analyze the performance through evaluation of scale-up and speed-up, which are the most fundamental and important factors to evaluate a parallel algorithm. In our experiments, we used *IBM Quest Market-Basket Synthetic Data Generator* [15][16] to generate the initial datasets. After generated, these data need to be transferred into standard form as <*TID*, i_1, i_2, ..., i_m> and then stored into HDFS. The average length of transactions for all datasets is 15. The support threshold is set to 0.075%.

The first experimental result explains why PSON is superior to the traditional SON algorithm in the context of very large database. From figure 3 we can see that as the size of the database increases, the time cost by SON increases much faster than PSON. When the size of the database reaches a threshold of hundreds of GB, SON cannot finish running in an acceptable period of time, but PSON can.

The second experimental result we want to observe is how the size of the local frequent itemsets and the global candidate itemsets changes as the number of partitions varies. In this experiment, the size of the database is 50GB. It can be seen from table 2 that as the number of partitions increases, the numbers of both local frequent itemsets and global candidate itemsets increase as well. However, it should be noted that there is a large overlap among local frequent itemsets [1]. The reason is obvious: a local frequent itemset may exist in several of these partitions. This will not affect the correctness of PSON, because after the shuffle and sort phase, all the duplicates of a local frequent itemset will be sent to only one reducer node.

Next we evaluate the scale-up performance of the PSON algorithm in the third experiment by varying the number of transactions from 1 million to 500 billion, with sizes ranging from 6GB to 440GB respectively. We can see from figure 4 that the time cost increases almost linearly with the size of the datasets. From this result we can conclude that PSON can achieve a good performance in scale-up.

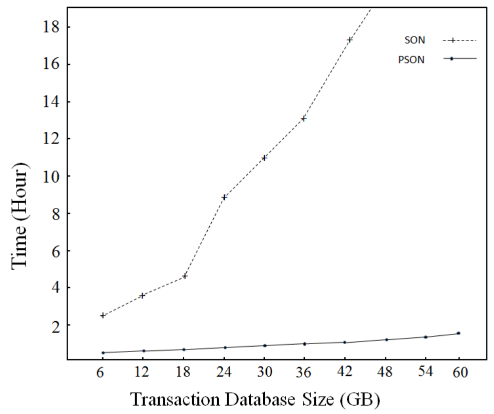

Figure 3: Comparison on time costs of SON and PSON

Table 2: Variation of local frequent itemsets and global candidate itemsets against the number of partitions

Number of Partitions	Size of Largest L^p	Size of C^G
2	45004	56802
4	53803	64045
8	89118	90418
14	111653	129467
20	174273	183643
30	218891	229493

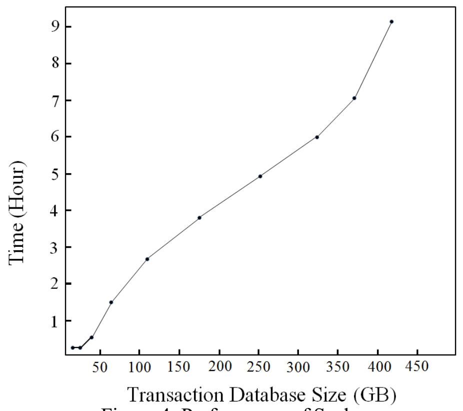

Figure 4: Performance of Scale-up

Figure 5 presents the results of the fourth experiment, which aims to evaluate the performance of speed-up. This experiment tries to find all frequent itemsets contained in an 80GB dataset on different number of running nodes in a cluster. We can see that as the number of running nodes increases, the time cost shows an approximately linear decrease. This result indicates that the PSON algorithm can achieve a good performance in speed-up. The reason why the time cost increases more than the proportion to the number of running nodes is that more running nodes would involve more network overheads.

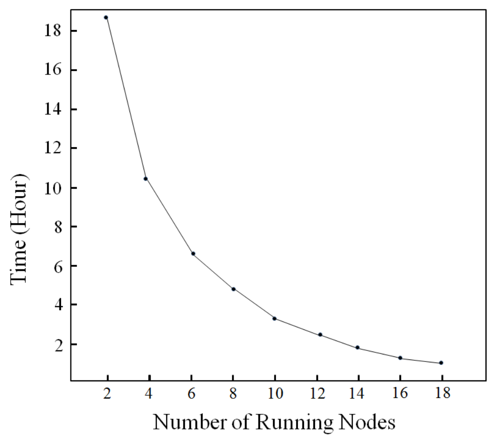

Figure 5: Performance of Speed-up

VII. Conclusions

In this paper, we proposed PSON, a parallelized SON algorithm implemented with MapReduce in Hadoop, to mine frequent itemsets from a very large transaction database. The major contribution of our work is that PSON makes it possible to mine frequent sets in a very large transaction database with an acceptable time cost. Further we have discussed several optimizing schemes that should be concerned when implementing PSON, for example, how to choose the number of partitions, how to prune the global candidate itemsets and what buffer strategy to adopt in order to achieve a tradeoff between I/O and performance. From experimental results we conclude that PSON can achieve a good performance in both scalability and speed-up, and is particularly meaningful and useful when dealing with huge amount of data in clusters, whereas it is almost impossible to fulfill such a big challenge with an acceptable time cost in a single node.

For our future work, we plan to optimize the PSON implementation in Hadoop, for example, reducing I/O and network overhead further, and achieving a better tradeoff between the number of partitions and performance.

References

[1] A. Savasere, E. Omiecinski, and S. Navathe, "An efficient algorithm for mining association rules in large databases", in *Proceedings of the 21st VLDB Conference Zurich*, Swizerland, 1995.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases", in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[3] R. Agrawal, R. Srikant, "Fast algorithms for mining association rules", in *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29-September 1 1994.

[4] A. Rajaraman, J. D. Ullman, *Mining of Massive Datasets*, unpublished.

[5] J. Dean, S. Ghemawat, "MapReduce: simplified data processing on large clusters", in *Proceedings of OSDI*, 2004.

[6] http://en.wikipedia.org/wiki/MapReduce.

[7] T. White, *Hadoop: The Definitive Guide*, Sebastopol, CA:O'Reilly, 2009.

[8] J. Venner, *Pro Hadoop*, Berkey, CA:Apress, 2009.

[9] C. Lam, *Hadoop in Action*, Greenwich, CT:Manning, 2010.

[10] J. Lin, C. Dyer, *Data-Intensive Text Processing with MapReduce*, San Rafael, CA: Morgan & Claypool, 2010.

[11] http://en.wikipedia.org/wiki/Apache_Hadoop.

[12] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system", in *19th ACM Symposium on Operating Systems Principles*, Lake George, NY, October, 2003.

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

[14] http://hadoop.apache.org/hdfs/.

[15] http://www.almaden.ibm.com/cs/disciplines/iis/#assocSynDat a.

[16] http://www.cs.indiana.edu/~cgiannel/assoc_gen.html.