Hori-Vertical Distributed Frequent Itemsets Mining Algorithm on Heterogeneous Distributed Shared Memory System

Margahny H. Mohamed[†] **and Hosam E. Refaat**[††]

Dept. of Computer Science, Faculty of Computers and Information, Assuit University, Egypt

Summary
The big challenge in discovering association rules is to find the largest frequent itemsets. Sequential algorithms do not have analytical ability, especially in terms of run-time performance, for such very large databases. Therefore, we must rely on high performance parallel and distributed computing. We present a new parallel algorithm for frequent itemset mining, called *HoriVertical* algorithm. The algorithm passes the database only one time and starts a new stage with the finished itemsets while some other itemsets in the same stage have not been finished yet. Also, the new algorithm is based on partitioning the database vertically and horizontally. We present the result on the performance of our algorithm on various databases, and compare it against well known algorithms.

Key words:
Parallel Systems, Distributed shared memory, data mining, Association rule, Linda system, Tuple-space, Jini, JavaSpace.

1. Introduction

Business organizations are increasingly turning to the automatic extraction of information from large volumes of routinely collected business data. Association rule mining (ARM) finds interesting associations and/or correlation relationships among large set of data items. Association rules show attribute value conditions that occur frequently together in a given dataset. Discovering this association rules in data can guide the decision making. A typical and widely-used example of association rule mining is Market Basket Analysis. The problem of mining association rules can be formally stated as follows: Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. Let DB be a database transactions, where each transaction consists of a set of items such that $T \subseteq I$. The support of an itemset X, denoted $\sigma(X)$, is the percentage of transactions in DB which it occurs as a subset. Given an itemset $X \subseteq I$, a transaction T contains X iff $X \subseteq T$. A X *frequent* or *large* if its support is more than a user-specified *minimum support* (*min_sup*) value(S). An itemset is maximal if it is not a subset of any other itemset [1].

An association rule is an implication of the form $X \Rightarrow Y$ has support S in the DB if the probability of the transaction in DB contains both X and Y is $\sigma(X \cup Y) = S$. Where, $X, Y \subseteq I$ and $X \cap Y = \phi$. The *confidence* of the rule is the conditional probability that a transaction contains Y, given that it contains X, and is given as $\sigma(X \cup Y) / \sigma(Y)$. A rule is *frequent* if its support is greater than min_sup, and it is *strong* if its confidence is more than a user-specified *minimum confidence* (*min_conf*). The task of mining association rules is to find all the association rules whose support is larger than a minimum support threshold and whose confidence is larger than a minimum confidence threshold. The data mining task for association rules can be broken into two steps. The first step consists of finding all large itemsets, i.e., itemsets that occur in the database with a certain user-specified frequency, called minimum support. The second step consists of forming implication rules among the large itemsets [10]. In this paper, we only deal with the first step.

An interesting algorithm, Apriori [3], has been proposed for computing large itemsets at mining association rules in a transaction database. Because databases are increasing in terms of both dimension (number of attributes) and size (number of records), one of the main issues in a frequent itemset mining algorithm is the ability to analyze very large databases. Sequential algorithms do not have this ability, especially in terms of run-time performance, for such very large databases [10], [8]. The only way to have efficient ARM algorithm is to make it parallel. To construct a parallel system, there are three models, namely; distributed memory systems(DMS), shared memory systems (SMS), and distributed shared memory systems (DSMS). The distributed shared memory is the newest parallel technique [20].

In distributed memory systems, each node in the system has its private memory. If any node needs data from another node, it will send a request message to it. Hence, this system is also called"message passing". In the message passing systems, if any node needs to send a message to another node, it must know the receiver node address. The direct connection between nodes will increase the performance but the parallel application will

Manuscript received November 5, 2010
Manuscript revised November 20, 2010

depend on the system structure (machine addresses) [12]. The second type is ”shared memory”. The shared memory systems are based on the existence of a global memory shared among all nodes in the system. A shared memory system has various advantages, such as; it is simple, it eases the data sharing and it eases the implementation of the parallel application. If any node needs to send a message to another node, it sends the message to the shared memory and the receiver node takes it from the shared memory. In another words, no synchronization required between nodes [5].

The DSMS systems have some countable advantages over the DMS based ones, these are; the application level ease of use, the DSMS is portable, and it is easy to share data and processes. Since the sender does not need to know the address of the receiver, then the structure of the application becomes more simpler and the application code is more readable. Also, in DSMS based systems, it is easy to share data and processes among the nodes by inserting the data or processes in the shared memory like SMS. This is done by constructing a virtual shared memory using the available distributed memories system. Moreover, the DSMS has standard operations that make parallel programming portable and more comfortable [17]. The Jini system is an extension of the Java environment. The DSMS had been implemented as a service in Jini system. A JavaSpace is a service in Jini system that implements the DSMS model. A JavaSpace inherits the advantages of Jini and the Java platforms [16].

Most parallel data mining algorithms is based on database partitioning (vertically or horizontally) [11], [2], [7], [22], [6], [19], [18], [21]. Both vertical and horizontal partitioning algorithms require all system nodes to be synchronized at the end of database pass to exchange the count or frequent itemsets. In the heterogeneous systems all node hasn’t equivalent resources. This means that if the system has a slow node, all other nodes will block until the slow node finished. In huge database, the vertical or horizontal partitioning may yield a big enough task to poor resources node to block all system. The solution of this problem is in the following points. The first point is to minimize the I/O operations. In other words, the number of database scanning must be minimized. The tasks that will be distributed must be in a reasonable size for all nodes to load it in its local memory. The second point is to minimize the dependency between system nodes. In other words, some node can start a new stage (k - itemset) while the other nodes do not finish the previous stage($k-1$ itemset). Our algorithm can control the number of database scanning to be one. Also, the database division is done vertically and horizontally. So, the task of finding frequent itemsets is divided into reasonable task size.

The HoriVertical algorithm can start a new stage while the current stage is not finished yet. For example, suppose that at the first stage (L_1) the items A, B are finished and both of them are large and the other items like C, D are not finished yet. Then the algorithm can create a new task for counting the itemset " AB ", that can be processed by any system clients. When the other item like C is finished and becomes large, the algorithm can start joining these items with the all finished items like A, B and then pruning to create the new tasks. This allows creating new tasks for different stages. So, no node will be idle, because there are lots of new independent tasks in the distributed shared memory that can be taken to process.

To compare our algorithm performance we choose one of the most important algorithms that makes vertical database partitioning that called *Eclat* algorithm [23]. Also, we use one of the newest parallel association rule algorithm, introduced by Limine, that called ”*Workload Management Distributed Frequent itemsets mining*” (*WMDF*) [4]. The WMDF algorithm is based on the horizontal database partitioning and it makes load balancing between system nodes.

Section(2) introduces a JavaSpace service as a DSMS implementation over the Jini system. Section(3) introduces our new algorithm, that is called HoriVertical algorithm. Section(4) shows the result of our algorithm compared by the Eclat, WMDF and Apriori algorithms.

Fig. 1. Jini system architecture

2. Jini-JavaSpace System

The Jini system is an extension of the Java environment. A JavaSpace is a service in Jini system that implements the DSMS model. This section introduces the Jini system and the JavaSpace service.

2.1 Jini System:

Jini system is a distributed system based on the idea of federating group of users and the resources required by these users to have a large monolithic system [16], [15]. Jini system extends the Java environment from a single

virtual machine to a network of virtual machines. Jini provides a framework to construct distributed applications of any size. This is done by the *lookup* service which is the core of the Jini system since it is responsible for finding and resolving the service. The Jini system uses *Remote Method Invocation* (*RMI*) to accomplish the communication among services, see figure 1. RMI enables passing full objects (code and data) around the network. The power of the Jini comes from the services, since services can be anything joined to the network. The lookup service is the "*main service*" in the Jini system. The services can be joined to the lookup service by the discover/join protocols. This gives the Jini system hierarchical lookup. The lookup service is essential since it is a *meta-service* or *naming service* that keeps track of all existing Jini services on the network. Sun Microsystems default implementation of this lookup service is called *REGGIE* [15].

2.2 JavaSpace Service:

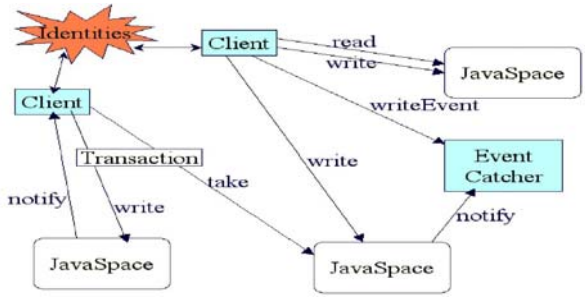

Fig. 2. JavaSpace model

JavaSpace is a distributed shared memory service that is implemented over Jini System [14]. The object that can be written in JavaSpace service is called "*entry*". The entry can contain data or/and processes. Sometimes the entry is called *tuple*. JavaSpace contains the following operations: take, takeIfExists, read, readIfExists, write, notify, snapshot. The write operation is to write an entry in JavaSpace. To read an entry from the JavaSpace, the read() or readIfExists() operation is used. The consecutive reading operation of the same template may return different entries even if JavaSpace contents are not changed. The difference between these two versions of reading is that; readIfExist-s() is not blocked if the tuple is not found in the space, it returns a null tuple if there is no matching tuple. Take() or takeIfExists() are two operations that extract entries from JavaSpace. In other words, these operations are similar to read and readIfExists() operations except that; taking operations remove the entry from the space. The snapshot operation is to take a copy of existing entry, but this copy is not updated in spite of the changes that may occur in the original entry. The notify operation is used to define an event that triggers when a specific entry is written [14]. See figure 2.

3. The Horizontal-Vertical Distributed Frequent Itemset Mining Algorithm

In this section we introduce a new parallel frequent itemset mining algorithm. Our algorithm only scans the database in the initial stage ($k = 1$). In the other stages ($k > 1$) the algorithm depends on the distributed shared memory, because the size of the database will be shrunk. So, in the initial stage of HoriVertical algorithm the database is divided horizontally and vertically at the same time, see figure 3. So, the first stage of our algorithm is to create a task for each database partition. Then, write the tasks in the distributed shared memory to be executed by the clients. These tasks in the form of JavaSpace entry are called "*taskEntry*". Each client will take one taskEntry after anther to execute it. The initial stage of the HoriVertical algorithm, is called *Initial_task_creation* procedure, and is shown in algorithm(1).

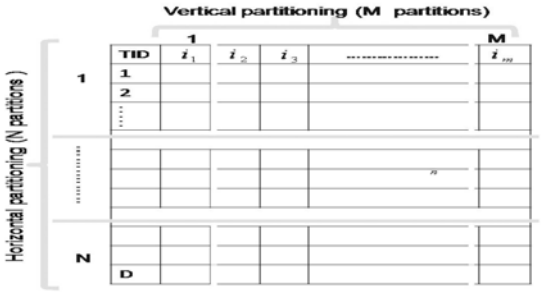

Fig. 3. Database partitioning

Algorithm 1: Initial_task_creation procedure in HoriVertical algorithm

```
for all i ∈ I do  /*all item in the database*/
    for j = 1;j ≤ N;j ++ do
        Create taskEntry( DBj,i)
        Write taskEntry( DBj,i ) in the JavaSpace
    end
end
```

The database partitioning is done in two dimensions (vertically and horizontally). At the first dimension, horizontally division, the database is divided in to N parts. So, the database DB will be divided into ($DB_1, DB_2, \ldots, DB_N$). Then, $DB = \cup_{i=1}^{N} DB_i$. As the number of database parts increases, the size of each part decreases. If the size of database partition is very small, all nodes will waste their time in taking and retrieving the JavaSpace tasks. The best choice of N will depend on the number of nodes and its resources [4]. For a given

minimum support threshold S, an itemset x is globally frequent if it is frequent in DB; its support $x.\sup$ is greater than $S \times DB$, and is locally frequent in a node N_i if it is frequent in DB_i; its support $x.\sup_i$ is greater than $S \times DB_i$.

- Property 1: A globally frequent itemset must be locally frequent in at least one node [7].

- Property 2: All subsets of a globally frequent itemset are globally frequent [7].

The second dimension is the vertical partitioning. The database will be divided vertically depending on the items. For example, to determine the transaction Id list (TIDList) of item $i \in I$, the transaction ID that contains this item must be count in all horizontal partitions.

Algorithm 2: The taskEntry Algorithm

```
if (k=l) then
  /*The 1-itemsets (need database scan)*/
  for all transactions t ∈ DB_i do
    if item i ∈ t t then
      resultList.add(TID) /* Add transaction Id into
                           result itemset list*/
    end
  end
else
    /*At the stages k > 1. We have two itemsets X and
     Y must be joined into new itemset XY and count
     it's frequent*/
    resultList = X.TIDList ∩ Y.TIDList
end
Encapsulate the resultList into a resultEntry.
Return a resultEntry that to the DSMS (JavaSpace)
```

Algorithm3: The master Process in HoriVertical algorithm

```
V_Ck /*Vector of stages Candidate itemsets */
Call Initial_task_creation() procedure ;
Call ResultCollector() thread ;
Call taskCreator() thread;
while true do
  if all tasks finished and C_k.size < 1 then
    Kill ResultCollector() thread
    Kill taskCreator() thread
    Break the loop
  end
end
```

The "taskEntry" is a JavaSpace entry that contains algorithm for counting the frequency of the itemset i. The taskEntry algorithm contains two cases, see algorithm(2). The first case, when $k = 1$, the algorithm must scan its database partition to determine the TID list (that is called "*resultList*"). The second case, if $k > 1$. At this stage the taskEntry contains two TID lists of two itemsets (X and Y). Also at this case, the client must create a resultList that contains the intersection between the two itemset(X, Y) TIDLists. Then, the client will create an entry called "*resultEntry*" that encapsulates the resultList. The resultEntry must be written to the JavaSpace by the client.

Algorithm4: The ResultCollector thread in HoriVertical algorithm

```
while true do
  if new resultEntry written then
    Take resultEntry;
    Update V_Ck ;
  end
end
```

Algorithm 5: The taskCreator thread in HoriVertical algorithm

```
while true do
    if new itemset finished then /*All taskEntries for
                                   this itemset finished*/
      if the itemset is large then
        Join the itemset with the other finished
         itemSets in the same stage;
        Prune the new candidate;
        Create a taskEntry for the new candidate;
        Write the new taskEntry into JavaSpace;
         update V_Ck ;
      else
        Delete this itemset from the V_Ck ;
      end
    end
end
```

What we have discussed so far is the initial stage of the HoriVertical algorithm, and the mechanism of database partitioning. Also, we have introduced the taskEntry algorithm that discusses the work of the system clients. Now, we will discuss the main structure of the HoriVertical algorithm. As seen in algorithm(3), it uses a vector V_{C_k} to register candidates and the related information for each stage. The algorithm starts by calling the Initial task creation procedure to start creating the 1-itemset tasks, as seen in algorithm(1). The next step the algorithm calls the resultCollector thread, see algorithm(4). This thread is responsible for collecting the resultEntry from the JavaS-pace and update the V_{C_k} vector. Then, the HoriVertical algorithm calls the taskCreator thread. The taskCreator thread checks if there is any finished itemset. The itemset is finished if all entryTasks that count this itemset are finished. If the finished itemset is not large, it

deletes this itemset from the V_{C_k} vector. If the itemset is large, it will join this itemset to the other finished large itemset in the same stage. After joining, the algorithm must make prune to the new candidates. Then, it creates a new taskEntry for the new candidates and puts it in the JavaSpace. The taskCreator thread is seen in algorithm (5). The algorithm will finish if all tasks in the V_{C_k} are finished and the size of the last stage of this vector is less than 1. This means that, all taskEntries created by the algorithm are finished and the last stage of the algorithm does not have any candidates. At this point, the algorithm must kill all the threads it has and the large frequent itemsets is exists in V_{C_k} vector.

2.1 *A Method of Performance Enhancement:*

The most costly stage in the HoriVertical algorithm is the initial stage for finding the first frequent itemset (L_1). This is because it needs to scan the database (I/O operations). In this section, we discuss the best method of partitioning for enhancing the performance of this stage.

Suppose that the size of the database is $|DB| = D$ (number of transactions). So, the time needed for scanning all the database one time is:

$$T = P*D \tag{1}$$

Where P is the time for reading one transaction. Suppose that, the required time for reading one transaction from the database is one unit of time (regardless how many items the transaction has). Then, the time needed for s-canning all the database is D. If database is partitioned only horizontally into N partitions. Where, N is the number of nodes in the parallel system. Theoretically, the time for scanning the database one time in this parallel system will be.

$$T \cong \sum_{i=1}^{N} P_i * |DB_i| \tag{2}$$

Let $P_i = 1 \ \forall i = \{1,...,N\}$. In other words, all nodes take a unit of time in reading a transaction from its database partition. So, the required parallel execution time for scanning the database using all nodes is:

$$T \cong \frac{D}{N} \tag{3}$$

Let the number of items in the database is m. And let the database is divided into N horizontal partitions and m vertical partitions in the initial stage of HoriVertical algorithm. This means that, for a horizontal database partition DB_i will be scanned m time, because it is divided into m vertical partitions. So, the approximated parallel execution time required for this stage is:

$$T \cong \frac{m.D}{N} \tag{3}$$

This time is very large. For example, let the number of items in the database is 100 items and the number of nodes equals 10 in our system. Then, the total approximated time in the initial stage will be like scanning the database 10 times. So, the best case in this stage is when the number of vertical partitions equals 1. In this case, the database is divided only horizontally. But, if there is a slow node, all other nodes will be idle until this node is finished. Moreover, the algorithm can not start finding the 2-itemset (L_2) until finishing 1-itemset (L_1). This is because the total frequent count of all items is not finished. The worst case in this stage is to divide the database vertically depending on all items (m partitions). Then, the parallel execution time for the initial stage of our algorithm is like scanning the database m/N times. As we promised, that our algorithm will scan the database only one time. So, the database will be partitioned into N vertical and horizontal partitions.

3. Experimental Results

In this section, we compare the performance of our algorithm with the Eclat, WMDF and Apriori algorithms. All experiments have been performed on five PCs. These PCs can be heterogeneous, but in performance test we would like to unify the resource of the system nodes. This because to highlight the effect of other parameter like, database size, minimum support and number of nodes in the system. These PCs have a CPU of type Intel(R) Core(TM) 2Duo 1.6 G.H and 2GB RAM. The intercommunication between the machines is done by 100 Mbps Ethernet. The software environment is as follows; Windows XP professional, Java JDK 1.4.2 04 [13], Jini(TM) Technology Starter Kit v2.0.2 [16] and a free visual platform for Jini 2.0 is called Inca X(TM) [9].

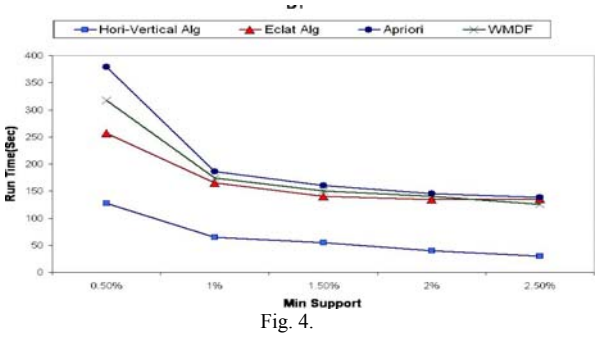

Fig. 4.

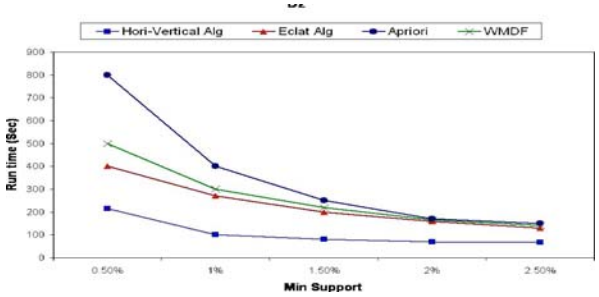

Fig. 5.

To measure the performance of the algorithms we use three synthetic datasets: D1=T10I4D10K, D2=T10I4D100K, D3=T10I10D1000K. The dataset T10I4D10K meant an average transaction size of 10, an average size of the maximum potentially frequent itemsets of 4, and 10000 generated transactions. These datasets generated from the IBM Almaden Quest research group. Also, we use one real datasets called "Chess". The Chess datasets contains 3196 transactions, the average transaction size is 34 and it contains 75 items. Each experiment has been performed 4 times. The average of the four runs is taken and used for analysis.

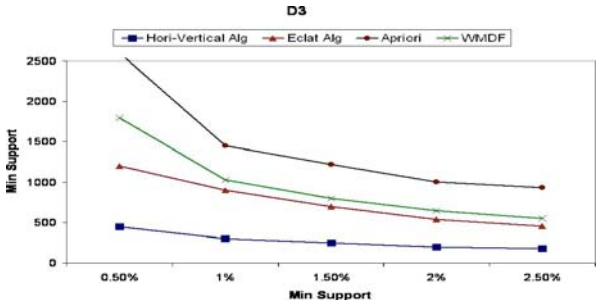

Fig. 6.

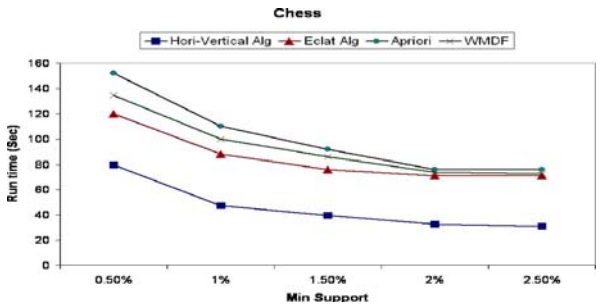

Fig. 7.

The first of the experimental results is shown in Figure 4. This experiment compares our algorithm with the other three algorithms using D1 datasets. From this figure, we can notice that the Eclat algorithm is better than the WMDF. This is because the Eclat algorithm scans the database only three times. But at the big minimum support, the performance of the WMDF is going to be better than Eclat. Figure 5 shows the same test using D2 as datasets. We can notice that, at the big minimum support the performance of the Eclat, WMDF and Apriori algorithms are closed. The performance comparison using big datasets (D3) is shown in figure 6. The real datasets test done on the Chess database is seen in figure 7. In all of the previous figures we notice that, the HoriVertical algorithm has the best performance. Also, the Apriori algorithm has the worst performance, because this algorithm is a sequential and runs on one machine. The Eclat algorithm have performance better than the WMDF algorithm. This is because, the Eclat algorithm scans the database three times and the WMDF algorithm scans the database a lot of times.

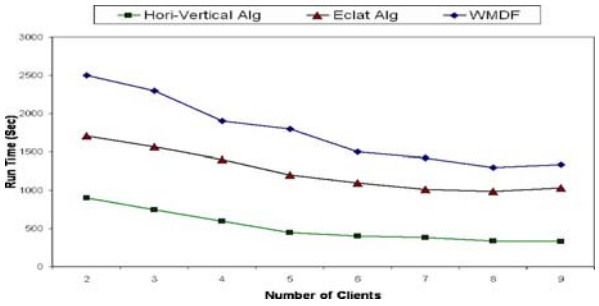

Fig. 8.

Because of apriori is a sequential algorithm (run on one processor), it is unfair to compare it with the parallel algorithms on multiple processors. So, we measure the performance of the parallel algorithms (Ecalt, WMDF, HoriVertical) using different number of clients in the system, as seen in figure 8. This test is done using the biggest dataset we have (D3) and in case of minimum support equals 0.5%. From this figure we can notice that, the WMDF curve is not smooth because the redistribution of the database blocks can raise the communications. Also, our algorithm has the best performance.

4. Conclusion

Through this paper, we have presented HoriVertical algorithm. It is based on DSMS which has various advantages over the other parallel models. It has powerful features, such as; scanning the database only one time and processing different stages of large itemsets at the same

time. Also, we have introduced a new database partitioning that is based on dividing the database vertically and horizontally into equivalent parts. Such new partitioning technique reduces the dependencies between nodes in the distributed system. Also, there is no synchronization at the end of each stage. Moreover, we have compared our algorithm with a well-known algorithm Eclat, Apriori algorithm and a new load balanced algorithm called WMDF. Finally, we have shown how the HoriVertical algorithm has better performance than the Apriori, WMDF and Eclat algorithms.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, Mining association rules between sets of items in large databases., In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216, Washington (1993).

[2] R. Agrawal and J. C. Shafer, Parallel mining of association rules, IEEE Transactions On Knowledge And Data Engineering, Volume 8 pages:962-969 (1996).

[3] R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases, Proceedings of the 20th International Conference on Very Large Data Bases, pages 487–499 (1994).

[4] L. M. Aouad, N Le-Khac, and T. M. Kechadi, Distributed frequent itemsets mining in heterogeneous platforms, Engineering, Computer and Architecture, Volume 1 (2007).

[5] U. Badawi., A single system image supporting distributed objects, Ph.D. thesis, Dept. of Mathematics, Faculty of Science,Cairo University, nov. 2000.

[6] D. W. Cheung and Y. Xiao, Effect of data skewness in parallel mining of association rules, Lecture Notes in Computer Science, Volume 1394 (1998).

[7] T. Vincent W. Ada D. Cheung, H. Jiawei and Y. Yongjian, A fast distributed algorithm for mining association rules., 4th Intl. Conf. Parallel and Distributed Info. Systems, (1996.).

[8] M. Hahsler, G. Bettina, K. Hornik, and C. Buchta, Introduction to arules a computational environment for mining association rules and frequent item sets, 2010.

[9] inca X, Inca x(tm) community edition, available from Incax WWW Site (http://www.incax.com/download.com), 2007.

[10] H. Jiawei and M. Kamber, Data mining: Concepts and techniques, second edition (the morgan kaufmann series in data management systems), 2 ed., vol. 2, Morgan Kaufmann; 2 edition, November 2005.

[11] S. Kotsiantis and D. Kanellopoulos, Association rules mining: A recent overview, GESTS International Transactions on Computer Science and Engineering, Volume 32 Pages:71–82 (2006).

[12] T. G. Mattson, Programming environments for parallel and distributed computing: A comparison of p4, pvm, linda and tcgms-g., ftp Server, ftp.cs.yale.edu (1995).

[13] Sun Microsystems., Java development kit, vol. 1.4.2 04, available from Sun Microsystems WWW Site (http://www.sun.com/products/jdk), 2004.

[14] Sun Microsystems, Javaspaces specification, vol. 2.0.2, available from Sun Microsystems WWW Site (http://java.sun.com/products/javaspaces), jun 2008.

[15] Sun Microsystems, Jini architecture specification, vol. v2.0.2, available from Sun Microsystems WWW Site (http://www.sun.com/jini/), jun 2008.

[16] Sun Microsystems, Jini technology core platform specification., vol. v2.0.2, available from Sun Microsystems WWW Site (http://www.sun.com/jini/), jun 2008.

[17] H. E. Refaat, New mechanism to integrate fault tolerance in a distributed shared memory based system, Computer science, Cairo Uni, 2007.

[18] A. Schuster and R. Wolff, Communication-efficient distributed mining of association rules, ACM SIGMOD Int'l. Conference on Management of Data, Santa Barbara, California, pp. 473-484. (2001).

[19] P. Tang and M. Turkia, Parallelizing frequent item-set mining with fp-trees., Technical Report titus.compsci.ualr.edu/ ptang/papers/par-fi.pdf, Department of Computer Science,University of Arkansas at Little Rock (2005).

[20] M. Tomasevic, J. Protic, and V. Milutinovic., Distributed shared memory: Concepts and systems., IEEE Parallel and Distributed technology, 4(2):63-79 (1996).

[21] D. YaJun and L. HaiMing, Strategy for mining association rules for web pages based on formal concept analysis, Appl. Soft Com-put. volume 10 pages:772–783 (2010).

[22] M. J. Zaki, Parallel and distributed association mining: A survey, IEEE Concurrency 7 (1999), 14–25.

[23] M. J. Zaki, S. Parthasarathy, and L. Wei, A localized algorithm for parallel association mining, In Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, 1997, pp. 321–330.

Marghny H. Mohamed, Dept. of Computer Science, Faculty of Computers and Information *Science, Asyut University, Asyut, Egypt., date of birth: June 1965, received the PhD* degree in computer science from the University of Kyushu, Japan, in 2001, and the MS from Asyut university in computer science, in 1993 and BS degrees in Mathematics from Asyut University, Egypt, in 1988. He is an associate professor in the Department of Computer Science, University of Asyut. He has many publications which in the fields of Data Mining, Text Mining, Information Retrieval, Web Mining, Machine Learning, Pattern Recognition, Neural Networks, Evolutionary Computation, Fuzzy Systems. Dr. Marghny is a member of the Egyptian mathematical society and Egyptian syndicate of scientific professions., he is a member of some research projects in Asyut university, Egypt. He is a Manager of the project entitled "Medical Diagnostic System for Endemic Diseases in Egypt Using Self Organizing Data Mining".

Hosam E Refaat, has been graduated from the faculty of Science, Asyut university, Egypt, in 1998. In October 2006, he has finished his master degree in the field of distributed systems from the faculty of Science, Cairo University, Egypt. Currently, he is a lecturer of Computer Science in King Khalid University– Kingdom of Saudi Arabia.