A Parallel Apriori Algorithm for Frequent Itemsets Mining

Yanbin Ye
Acxiom Corporation
1001 Technology Drive
Little Rock, Arkansas 72223, USA
Yanbin.Ye@acxiom.com

Chia-Chu Chiang
Department of Computer Science
University of Arkansas at Little Rock
2801 South University Ave.
Little Rock, Arkansas 72204, USA
cxchiang@ualr.edu

Abstract

Finding frequent itemsets is one of the most investigated fields of data mining. The Apriori algorithm is the most established algorithm for frequent itemsets mining (FIM). Several implementations of the Apriori algorithm have been reported and evaluated. One of the implementations optimizing the data structure with a trie by Bodon catches our attention. The results of the Bodon's implementation for finding frequent itemsets appear to be faster than the ones by Borgelt and Goethals. In this paper, we revised Bodon's implementation into a parallel one where input transactions are read by a parallel computer. The effect a parallel computer on this modified implementation is presented.

Keywords: Apriori, Association Rules, Data Mining, Frequent Itemsets Mining (FIM), Parallel Computing

1. Introduction

Finding frequent itemsets in transaction databases has been demonstrated to be useful in several business applications [6]. Many algorithms have been proposed to find frequent itemsets from a vary large database. However, there is no published implementation that outperforms every other implementation on every database with every support threshold [7]. In general, many implementations are based on the two main algorithms: Apriori [2] and frequent pattern growth (FP-growth) [8]. The Apriori algorithm discovers the frequent itemsets from a very large database through a series of iterations. The Apriori algorithm is required to generate candidate itemsets, compute the support, and prune the candidate itemsets to the frequent itemsets in each iteration. The FP-growth algorithm discovers frequent itemsets without the time-consuming candidate generation process that is critical for the Apriori algorithm. Although the FP-growth algorithm generally outperforms the Apriori algorithm in most cases, several refinements of the Apriori algorithm have been made to speed up the process of frequent itemsets mining.

In this paper, we implemented a parallel Apriori algorithm based on Bodon's work and analyzed its performance on a parallel computer. The reason we adopted Bodon's implementation for parallel computing is because Bodon's implementation using the trie data structure outperforms the other implementations using hash tree [3]. The rest of the paper is organized as follows. Section 2 introduces related work on frequent itemsets mining. Section 3 presents our implementation for parallel computing on frequent itemsets mining. Section 4 presents the experimental results of our implementation on a symmetric multiprocessing computer. Section 5 summarizes the paper.

2. Related work

Data mining known as knowledge discovery in databases (KDD) is the process of automatically extracting useful hidden information from very large databases [10]. One of central themes of data mining is association rules mining between itemsets in vary large databases. Finding frequent itemsets through the mining of association rules was first presented by Agrawal et. al. [1]. In the following section, we briefly introduce the concepts of frequent itemsets mining.

Let TDB = $\{T_1, T_2, T_3, \ldots, T_n\}$ be a database of transactions. Each transaction $T_i = \{i_1, i_2, i_3, \ldots, i_m\}$ contains a set of items from $I = \{i_1, i_2, i_3, \ldots, i_p\}$ where $p \geq m$ such that $T_i \subseteq I$. An itemset X with k items from I is called a k-itemset. A transaction T_i contains an itemset X if and only if $X \subseteq T_i$. The support of an

Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)
0-7695-2656-X/06 $20.00 © 2006 IEEE

itemset X in T denoted as supp(X) indicates the number of transactions in TDB containing X. An itemset is frequent if its support, supp(X) is greater than a support threshold called *minimum support*. For example, suppose we have a TDB = {T_1, T_2, T_3, T_4, T_5, T_6} and I = {A, B, C, D, E} where T_1 = {B, C, D, E}, T_2 = {B, C, D}, T_3 = {A, B, D}, T_4 = {A, B, C, D, E}, T_5 = {A, B, C}, and T_6 = {B, E}. Figure 1 presents an example to illustrate frequent itemsets mining.

Transactions	Itemset
T1	{B, C, D, E}
T2	{B, C, D}
T3	{A, B, D}
T4	{A, B, C, D, E}
T5	{A, B, C}
T6	{B, E}

(a)

{1-Itemsets}	Transactions
{A}	T3, T4, T5
{B}	T1, T2, T3, T4, T5, T6
{C}	T1, T2, T4, T5
{D}	T1, T2, T3, t4
{E}	T1, T4, T6

(b)

Figure 1. An example to illustrate frequent itemsets mining

Figure 1(a) lists all the transactions with their itemsets and Figure 1(b) lists five 1-itemsets contained by the transactions. Thus, for instance, supp({A}) = 3, can be directly achieved from Figure 1(b). For k-itemsets where k ≥ 2, for instance, supp({A, E}) = 1, and supp({A, B, D, E}) = 1 can be computed from Figure 1(a) using the same way for generating 1-itemsets.

2.1 The apriori algorithm

The Apriori algorithm as shown in Figure 2 was originally presented by Agrawal and Srikant [2]. It finds frequent itemsets according to a user-defined minimum support. In the first pass of the algorithm, it constructs the candidate 1-itemsets. The algorithm then generates the frequent 1-itemsets by pruning some candidate 1-itemsets if their support values are lower than the minimum support. After the algorithm finds all the frequent 1-itemsets, it joins the frequent 1-itemsets with each other to construct the candidate 2-itemsets and prune some infrequent itemsets from the candidate 2-itemsets to create the frequent 2-itemsets. This process is repeated until no more candidate itemsets can be created.

```
Apriori(T, minsup)
Inputs:
  TDB: Transaction Database
  minsup: Minimum Support
Outputs:
  Li: Frequent Itemset // Supp(Itemset) >= minsup
Temporary Variables
  Ck: Candidate Itemset
Steps:
  Get L1 for 1-itemsets;
  k = 2;
  While (Lk-1 <> empty)
  Begin
    Ck = Candidate_Generation(Lk-1)
    For each transaction T in TDB
    Begin
      For each itemset c in Ck
      Begin
        If T includes c then
        Begin
          count[c] = count[c] + 1;
        End;
      End;
    End;
    Lk = Prune_Candidate(Ck, minsup);
    k = k + 1
  End;
  Output L1, L2,…, Lk;
```

(a)

```
Candidate_Generation(Lk-1)
Inputs:
  Lk-1: Frequent Itemset, // Supp(Itemset) >= minsupp
                // and size of every itemset = k -1
Outputs:
  Ck: Set of Candidate Itemsets with Size k
Steps:
  For each itemset m in Lk-1
  Begin
    For each itemset n in Lk-1
    Begin
      If (m <> n) then  // itself not joined
      Begin
        l = m join n;
        If sizeof (l) = k then
        Begin
          Ck = Ck  ∪ l; // join
        End;
      End;
    End;
  End;
```

(b)

```
Prune_Candidate(Ck, minsup)
Inputs:
  Ck: Set of Candidate Itemsets with Size k
  minsup: Minimum Support
```

```
Output:
  $L_k$: Frequent Itemset, //Supp(Item)  >= minsup and
                        //size of every itemset = k
Steps:
  For each itemset c in $C_k$
  Begin
    If (support(c) >= minsup) then
    Begin
      $L_k = L_k \cup c$;
    End;
  End;
  return $L_k$;
```

(c)

Figure 2. The apriori algorithm

Consider the Apriori algorithm with the example shown in Figure 1 and also assume that minimum support is 3 transactions. Figure 3 shows the application of the Apriori algorithm.

Transactions	Itemset
T1	{B, C, D, E}
T2	{B, C, D}
T3	{A, B, D}
T4	{A, B, C, D, E}
T5	{A, B, C}
T6	{B, E}

Pass 1

Candidate 1-Itemsets	Support
{A}	3
{B}	6
{C}	4
{D}	4
{E}	3

⇒

1-itemsets	Support
{A}	3
{B}	6
{C}	4
{D}	4
{E}	3

(a)

1-itemsets	Support
{A}	3
{B}	6
{C}	4
{D}	4
{E}	3

Pass 2

Candidate 2-Itemsets	Support
{A, B}	3
{A, C}	2
{A, D}	2
{A, E}	1
{B, C}	4
{B, D}	4
{B, E}	3
{C, D}	3
{C, E}	2

⇒

2-itemsets	Support
{A, B}	3
{B, C}	4
{B, D}	4
{B, E}	3
{C, D}	3

(b)

2-itemsets	Support
{A, B}	3
{B, C}	4
{B, D}	4
{B, E}	3
{C, D}	3

Pass 3

Candidate 3-Itemsets	Support
{A, B, C}	2
{A, B, D}	1
{A, B, E}	1
{B, C, D}	3
{B, C, E}	2
{C, D, E}	2

⇒

3-itemsets	Support
{B, C, D}	3

(c)

3-itemsets	Support
{B, C, D}	3

Pass 4

Candidate 4-Itemsets	Support
{}	

(d)

Figure 3. A sample application of the apriori algorithm

In Figure 3(a), the five 1-itemsets are created from the transaction database in which all the 1-itemsets have the support greater than the minimum support 3. Figure 3(b) presents five frequent 2-itemsets. Figure 3(c) presents one frequent 3-itemsets. However, there are no 4-itemsets in pass 4 created because no candidate 4-itemsets can be created. Thus, the process stops.

2.2 A fast apriori implementation

Bodon conducted a literature survey on frequent itemsets mining [5]. Bodon discusses the efficiency of frequent itemset mining algorithms that depends on the generation of candidates, the use of data structures, and the details of implementations. Bodon [3] claims that the implementation details are always neglected in existing algorithms. Thus, Bodon presents a fast Apriori implementation which provides more efficiency than the other implementations focusing on the way candidates generated and the data structures used.

The original Apriori algorithm was implemented using hash tree [1]. Bodon [3, 4] implemented the Apriori algorithm using the trie structure. A trie is a rooted directed tree. The depth of the root is zero. There are edges between two nodes and each edge is labeled by a letter. In the context of frequent itemsets mining, the letter represents an item in $I = \{i_1, i_2, i_3, \ldots, i_m\}$. Figure 4 presents a trie that stores all the candidate itemsets from I = {A, B, C, D, E} where 'A' represents bread, 'B' represents milk, 'C' represents cheese, 'D' represents cookie, and 'E' represents egg. Thus, a candidate k-itemset such as $C_k = \{i_1 < i_2 < i_3 < \ldots < i_k\}$ can be generated by composing items from $I = \{i_1, i_2, i_3, \ldots, i_m\}$. In Figure 4, for example, ten candidate 2-itemsets in C_2 can be generated from the trie including {A, B}, {A, C}, {A, D}, {A, E}, {B, C}, {B, D}, {B, E}, {C, D}, {C, E}, and {D, E}.

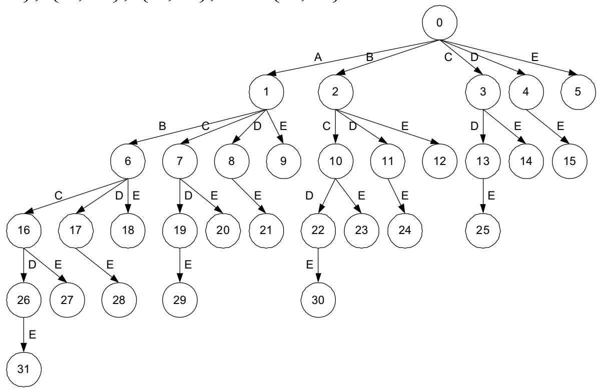


Figure 4. A trie containing itemsets

Using a trie for storing candidate itemsets has the advantages [3] including efficient candidate generation, efficient association rules production, simple implementation of the data structure, and immediate generation of negative border for online association rules [11] and sampling based algorithms [12].

3. Our Implementation for parallel computing

The Apriori algorithm has been revised in several ways. One revision of the Apriori algorithm is to partition a transaction database into disjoint partitions TDB_1, TDB_2, TDB_3, …, TDB_n. Partitioning a transaction database may improve the performance of frequent itemsets mining by fitting each partition into limited main memory for quick access and allowing incremental generation of frequent itemsets.

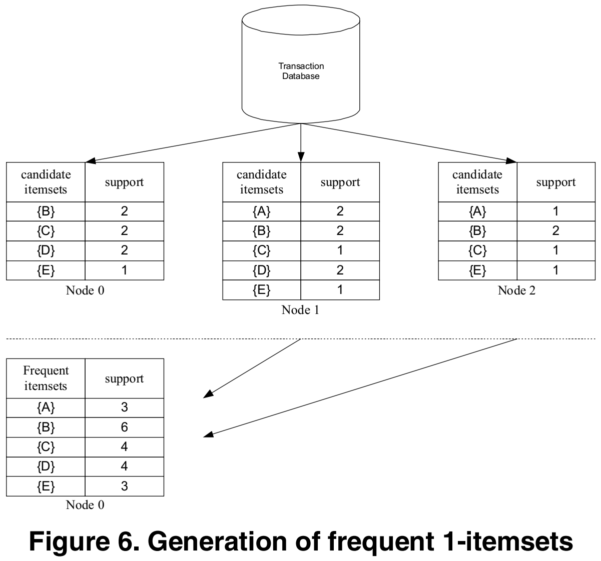
Our implementation is a partition based Apriori algorithm that partitions a transaction database into N partitions and then distributes the N partitions to N nodes where each node computes its local candidate k-itemsets from its partition. As each node finishes its local candidate k-itemsets, it sends its local candidate k-itemsets to node 0. Node 0 then computes the sum of all candidate k-itemsets and prunes the candidate k-itemsets to the frequent k-itemsets. Figure 5 shows our implementation for parallel computing. Assume there are N nodes in a parallel computer.

- Parallel Scan the Transaction Database (TDB) to get the frequent 1-itemsets, L_1
  - Divide the TDB file into N parts;
  - Processor Pi reads TDB[i] to generate the local candidate 1-temsets from its partition;
  - P_i passes its local candidate 1-itemsets to P_0;
  - P_0 calculates the counts of all local candidate 1-itemsets and prune it to generate the frequent 1-itemsets, L_1;
- Parallel scan TDB to get the frequent 2-itemsets, L_2
  - P_i scans TDB to get the local candidate 2-itemsets;
  - P_i passes its local candidate 2-itemsets to P_0;
  - P_0 calculates the counts of all local candidate 2-itemsets and prunes it to get L_2;

```
• K = 3
      – While ($L_{k-1}$ <> empty)
        Begin
            – Build Candidate Tree Trie[i]
              for $C_k$ on Processor $P_i$ based
              on $L_{k-1}$;
            – Scan TDB[i] on Processor $P_i$
              to update the counts of
              Trie[i] for $C_k$;
            – Prune the Trie[i] for $C_k$ to
              get $L_k$;
            – k = k + 1;
        End;
```

Figure 5.Our implementation for parallel computing

To compute frequent 1-itemsets, each node counts its local 1-itemsets from its partition and than passes the results to node 0. As node 0 receives all the results from the other N-1 nodes, it creates the global frequent 1-itemsets, L_1. Let us use the same example shown in Figure 3 to illustrate the generation of 1-itemsets using 3 nodes. The transaction database is evenly divided into 3 partitions. Node 0 will get the first two transactions T_1 and T_2. Node 1 will get T_3 and T_4. Node 2 will have the last two transactions T_3 and T_4. To simplify the implementation of the data structure, we use a vector to store candidate 1-itemsets in our implementation. Figure 6 demonstrates the count distribution process.



candidate itemsets	support
{B}	2
{C}	2
{D}	2
{E}	1

Node 0

candidate itemsets	support
{A}	2
{B}	2
{C}	1
{D}	2
{E}	1

Node 1

candidate itemsets	support
{A}	1
{B}	2
{C}	1
{E}	1

Node 2

Frequent itemsets	support
{A}	3
{B}	6
{C}	4
{D}	4
{E}	3

Node 0

Figure 6. Generation of frequent 1-itemsets

The same process is applied to find the candidate 2-itemsets. Each node computes its local candidate 2-itemsets based on the dataset distributed to it. The results are sent back to node 0 for the generation of the global frequent 2-itemsets. In our implementation, same as Bodon's implementation, we did not use the trie data structure for storing 1-itemsets and 2-itemsets. Instead, one simple vector is used to store candidate 1-itemesets and a two-dimensional array is used to store candidate 2-itemsets.

To generate frequent k-itemsets where k ≥ 3, a trie data structure is used to store candidate k-itemsets. Each node builds up its own local trie incrementally for its local candidate k-itemsets based on L_{k-1}. The count of each itemset is performed on the trie. Each node then prunes its own local candidate k-itemsets to get the frequent k-itemsets. The global frequent k-itemsets are then achieved by collecting all the local frequent k-itemsets from each node. The process is repeated until no more candidate itemsets are generated. Figure 7 depicts a trie for storing candidate k-itemsets among 3 nodes. Although there are many ways to partition a transaction database into 3 partitions, in our implementation, we chose to let each node handle their partitions independent to each other. Thus, the transactions containing an item A are distributed to node 0 and the transactions containing an item B are distributed to node 1. The rest of transactions are then assigned to node 2.

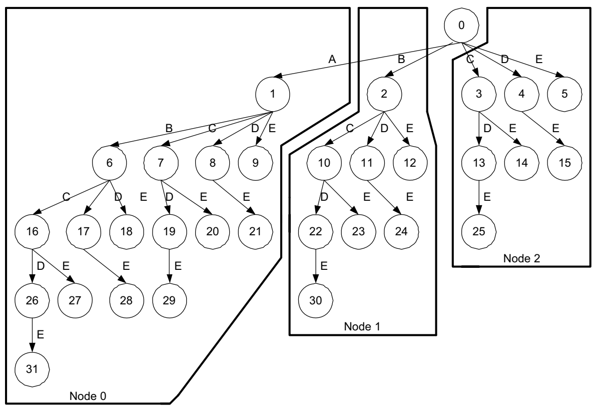


Figure 7. Trie distribution for generating candidate k-itemsets

Our implementation is basically a partition based Apriori algorithm. The data structure used to store candidate itemsets and their counts is a trie. The trie data structure provides an effective technique to store, access, and count itemsets. A trie is incrementally created dependent on the counts of candidate itemsets. If a candidate itemset is infrequent, the path with that itemset will not be further expanded in the trie. Thus a node in the trie only contains the frequent itemsets with the corresponding counts. Using the example shown in

Figure 1, Figure 8 shows a trie for storing frequent itemsets of minimum support 3.

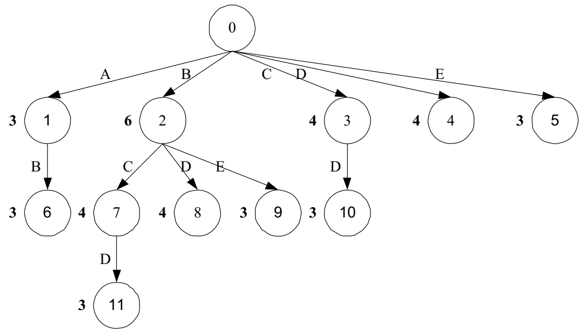


Figure 8. Trie tree for 1-itemsets and 2-itemsets

By traversing the trie tree, we get the following itemsets with the minimum support 3 transactions: {A}, {B}, {C}, {D}, {E}, {A, B}, {B, C}, {B, D}, {B, E}, {C, D}, and {B, C, D} with the support 3, 6, 4, 4, 3, 3, 4, 4, 3, 3, and 3, respectively. In other words, given a transaction database as shown in Figure 1, our implementation will incrementally build a trie tree to store all the frequent itemsets with their counts.

4. Evaluation of the implementation

The implementation was tested on the SGI® Altix® system [9], namely Cobalt that consists of 1,024 Intel® Itanium® 2 processors running the Linux® operating system, 3 terabytes of globally accessible memory, and 370 terabytes of SGI® InfiniteStorage that serves as the Center's shared file system. The system is a symmetric multiprocessing system where the file system is shared by other high-performance computing resources within the National Center for Supercomputing Applications (NCSA). The SGI® InfiniteStorage filesystem CXFS™ is depicted in Figure 9.

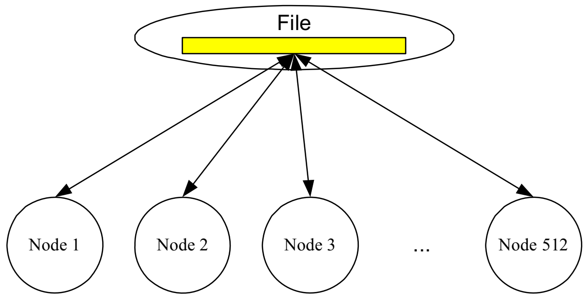


Figure 9. The CXFS file sharing system

The following two tables present the running wall time (in seconds) of the implementation on the 2 databases: T10I4D100K and T10I4D50K. T10 indicates the average transaction size of the database is 10 items. I4 indicates the average itemset size is 4. D100K indicates 100,000 transactions contained in the transaction database, T10I4D100K. The minimum support of the itemsets is 0.005. Figure 9 compares the performance of the implementation on these two datasets.

Table 1. T10I4D100K database

Nodes (Processors)	Wall Time (Secs)
1	1322
2	995
4	1017
8	998

Table 2. T10I4D50K database

Nodes (Processors)	Wall Time (Secs)
1	1066
2	550
4	515
8	551

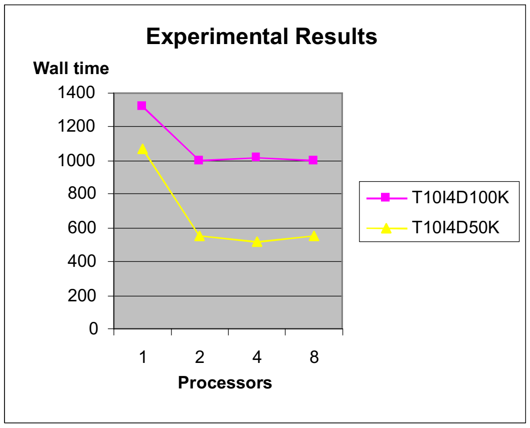


Figure 9. Running time comparisons

Figure 9 shows the performance of the implementation running on the system with 1, 2, 4, and 8 nodes. Apparently, the implementation running on one node has the worst performance compared to the performance of the implementation running on multiple nodes. Unfortunately, the implementation running on more nodes does not guarantee better performance than fewer nodes. The reason is that the system that the implementation was tested on is a symmetric multiprocessing system and the file system is shared by multiple nodes. This creates a bottleneck for multiple nodes to read in the transactions from a transaction database. Thus, running on multiple nodes in this environment does not help speed up the process for finding frequent itemsets.

In addition, the way we partition a transaction database ignores load-balancing issues. For example, in Figure 7, node 0 apparently has a heavier load than the other two nodes. Thus, unbalanced tries may be created by the nodes using our implementation. If all the transactions in Figure 7 contain an item A, then only node 0 has work to do. Node 2 and Node 3 will be idle. How to effectively partition a transaction database so every node will have equal load is our future work.

5. Summary

Frequent itemsets mining is one of the most important areas of data mining. Existing implementations of the Apriori based algorithms focus on the way candidate itemsets generated, the optimization of data structures for storing itemsets, and the implementation details. Bodon presented an implementation that solved frequent itemsets mining problem in most cases faster than other well-known implementations [3]. In this paper, we revised the Bodon's implementation for parallel computing. The performance of the implementation running on a symmetric multiprocessing computer was presented.

6. References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Database," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Vol. 22, Issue 2, 1993, pp. 207-216.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 487-499.

[3] F. Bodon, "A Fast Apriori Implementation," In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, Vol. 90 of CEUR Workshop Proceedings, 2003.

[4] F. Bodon, "Surprising Results of Trie-based FIM Algorithm," In B. Goethals, M. J. Zaki, and R. Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, Vol. 90 of CEUR Workshop Proceedings, 2004.

[5] F. Bodon, *A Survey on Frequent Itemset Mining*, Technical Report, Budapest University of Technology and Economic, 2006.

[6] M. S. Chen, J. Han, and P. S. Yu, "Data Mining: An Overview from a Database Perspective," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, 1996, pp. 866-883.

[7] B. Goethals and M. J. Zaki, "Advances in Frequent Itemset Mining Implementations: Introduction to FIMI03," In Goethals, B. and Zaki, M. J., editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, Vol. 90 of CEUR Workshop Proceedings, 2003.

[8] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 1-12.

[9] NCSA Computational Resources, Retrieved May 14, 2006 from *http://http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware*.

[10] G. Piatetski-Shapiro, *Discovery, Analysis, and Presentation of Strong Rules*. In Knowledge Discovery in Databases, Piatetski-Shapiro, G., editor, AAAI/MIT Press, 1991, pp. 229-248.

[11] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka, "An efficient Algorithm for the Incremental Updation of Association Rules in Large Databases," *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, 1997, pp. 263-266.

[12] H. Toivonen, "Sampling Large Databases for Association Rules," *Proceedings of the 22nd International Conference on Very Large Databases*, 1996, pp. 134-145.