# A New Approach for the Discovery of Frequent

Rosa Meo

Politecnico di Torino
Dip. Automatica e Informatica, c.so Duca degli Abruzzi 24 - 10129 Torino, Italy
rosimeo@polito.it

**Abstract.** The discovery of the most recurrent association rules, in a large database of sales transactions requires that the sets of items bought together by a sufficiently large population of customers are identified. This is a critical task, since the number of generated itemsets grows exponentially with the total number of items. Most of the algorithms start identifying the sets with the lowest cardinality, and subsequently, increase it progressively. Our approach is different, since the sets to be considered at a time are determined by the items in the sets. The main advantage is a significant reduction of the CPU time required to update data structures in main memory. This paper presents an algorithm that requires only one pass on the database, presents linear scale-up property with the dimensions of the database and, as shown by the experiments, performs better than other classical algorithms.

## 1 Introduction

Association rules are a powerful and intuitive conceptual tool to represent the phenomena that are recurrent in data. The discovery of association rules has several applications in the analysis of business data, such as the basket data of supermarkets, failures in telecommunications networks, medical test results, health insurance, and many others.

In a database of transactions, an association rule $\mathcal{X} \Rightarrow \mathcal{Y}$ associates two sets of data (also called sets of items) which are found together in a transaction. Its utility is to show which kind of items are frequently correlated in customers' purchases. The statistical frequency of an association rule (or more generally of a set of items) is called *support* and is the percentage of transactions of the database in which all the items in the association rule are present.

The number of association rules that may be extracted from a very large database is exponentially large with the number of the items. The feasibility of the problem requires a threshold for the support is provided by the analyst in order to discover only the most frequent association rules. Nevertheless, even if the problem is stated like this, it still may be critical. The most important step of the algorithms that extract association rules is to identify all the sets of items $\mathcal{S}$ whose support is higher than the threshold (called *large* itemsets). The computation of the effective support of a set of items requires that in reading the

Mukesh Mohania and A Min Tjoa (Eds.): DaWaK'99, LNCS 1676, pp. 193-202, 1999

database a counter is allocated in main memory to keep track of the number of transactions that contain the set. If the number of the examined sets is not kept low enough while the reading of the database is performed, the total number of counters may be too large to fit in main memory, or too much effort is wasted to keep the support of sets that eventually reveal to be lower then the threshold.

The algorithms that have already been proposed [1, 2, 3, 4, 5, 6] solve this problem iteratively. They keep only a subset of the collection of sets in main memory at a time. In particular, in each iteration, the cardinality of the sets whose support is being computed is fixed. After the support of each of them is known, the *pruning* phase is executed, to get rid of those sets whose support is lower than the threshold. In the next iteration the support of the sets with increased cardinality is determined. These sets (called the *candidate* sets) are identified from the large itemsets found in the previous iteration. These algorithms execute the pruning phase once for each iteration, and perform as many iterations as the cardinality of the longest itemsets with sufficient support. Notice, also, that some of these algorithms [1, 3, 4, 6] perform a reading pass on the database for each iteration. This reading pass determines the number of I/O operations performed in each iteration which are the most expensive from the viewpoint of the execution time.

In this paper we propose a new approach for the identification of all the large itemsets. This approach is based on the observation that the collection of the sets that is maintained in the main memory in each iteration can be arbitrarily chosen. The itemsets are ordered lexicographically. The first iteration keeps the itemsets that start with the last item in the lexicographical order, while the subsequent iterations keep the itemsets that start with the other items, in the decreasing order. The purpose is to improve the efficiency in the generation of the candidate itemsets and in the reduction of the number of accesses to main memory required to update their support.

A new algorithm based on this approach is proposed. The algorithm is called `Seq` for the fact that in the first step, instead of building itemsets, builds sequences of items. `Seq` reduces I/O execution times because it makes a single reading pass on the database. Moreover, we will show with our experiments that `Seq` is specifically oriented to databases of very large dimensions and searches of very high resolution, where the minimum support is defined at very low levels. Seq reduces also CPU execution times because it requires only two accesses to main memory in the generation of a candidate itemset and when updating its support counter, regardless of the itemset length; moreover it executes the pruning phase once for each item, instead of once for each value of itemset cardinality (thousands times more in real databases!).

The paper is organized as follows. Section 2 introduces some preliminary definitions and presents the algorithm. Section 3 provides an evaluation of its properties. Finally, Section 4 shows the results of some experiments, while Section 5 will draw the conclusions.

## 2 Algorithm Seq

### 2.1 Preliminary definitions

1. The database is organized in transactions. Each transaction is represented with the items lexicographically ordered. We indicate with `T[i]` the item of transaction `T` in the position `i` (i starts from 0).
2. We call the set of all the possible items in the database the *Alphabet*. We consider it lexicographically ordered and indicate with $<$ the ordering operator and with $>$ the operator of opposite ordering.
3. Given a transaction T of length L, the sequence of all items extracted from T starting at position j is denoted as $\mathtt{Seq}_{\mathtt{T}}^{\mathtt{j}}$ ($0 \leq j <$L) and defined as follows: $\mathtt{Seq}_{\mathtt{T}}^{\mathtt{j}}$=⟨T[j] T[j+1] T[j+2]$\cdots$T[L-1]⟩
   T[j], the starting item in the sequence $\mathtt{Seq}_{\mathtt{T}}^{\mathtt{j}}$, is called the *leader* of the sequence, whereas T[L-1], the last one, is called the *terminal* item.
   The number of sequences in a transaction T is equal to the length of T. For example, the transaction T=$\langle ABCD\rangle$ has four sequences:
   $\mathtt{Seq}_{\mathtt{T}}^{0}$=$\langle ABCD\rangle$, $\mathtt{Seq}_{\mathtt{T}}^{1}$=$\langle BCD\rangle$, $\mathtt{Seq}_{\mathtt{T}}^{2}$=$\langle CD\rangle$ and $\mathtt{Seq}_{\mathtt{T}}^{3}$=$\langle D\rangle$.
4. An ordered set, and thus also a sequence $\mathcal{S}$=$\mathtt{Seq}_{\mathtt{T}}^{\mathtt{j}}$, is stored in the main memory in a tree. The first item in the sequence (`Leader`{S}=T[j]) is saved on the root node of the tree; the second one (T[j+1]) on a son node of the first one, and so on. For example, the sequence $\langle ABCD\rangle$ would be saved on a tree with `A` on the root node, `B` on a son node of `A` and so on. A given tree is used to store all the sequences, having the same starting item. Figure 1 shows the tree with the two sequences $\langle ABCD\rangle$ and $\langle ABD\rangle$. The tree in the above example, is denoted as $\mathcal{T}_{\mathtt{A}}^{<}$ since `A` is the item in the root node and $<$ is the operator of ordering of the items. Viceversa, we denote with $\mathcal{T}^{>}$ a

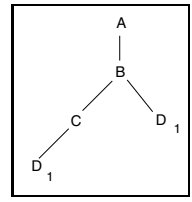


**Fig. 1.** $\mathcal{T}_{\mathtt{A}}^{<}$ tree with the two sequences $\langle ABCD\rangle$ and $\langle ABD\rangle$.

tree in which $>$ is used as operator of ordering to store the items in the tree.
5. A counter is associated to the terminal node of each sequence. It keeps the number of transactions in which the sequence (composed of the items stored from the root node to the terminal node) occurs in. Observe also that the counter of a sequence not necessarily is stored in a leaf node of the tree: this is the case of the sequence $\langle ABC\rangle$, substring of the sequence $\langle ABCD\rangle$.

### 2.2 Description of the Algorithm Seq

Algorithm Seq works in two steps.

**First Step.** The database is read. For each transaction it finds all the sequences and stores them in the main memory in $\mathcal{T}^<$ trees. For example, the transaction $T_1=\langle ABCD\rangle$ generates four sequences $\langle ABCD\rangle$, $\langle BCD\rangle$, $\langle CD\rangle$ and $\langle D\rangle$ that are stored in the four trees $\mathcal{T}_A^<$, $\mathcal{T}_B^<$, $\mathcal{T}_C^<$ and $\mathcal{T}_D^<$. If a sequence is found for the first time, the counter associated to the terminal node is set to 1; otherwise, it is incremented by 1. Figure 2 shows the trees $\mathcal{T}^<$ for a very simple database.

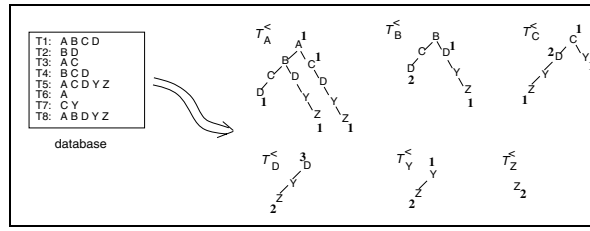


**Fig. 2.** The trees after the first step is completed.

At the end of the first step only the support of the sequences is known. However, the support of the itemsets can be determined from the support of the sequences as stated by the following theorem whose proof will be omitted.

*Theorem.* The support of an itemset $\mathcal{I}$, with item X as its first item, can be obtained as the sum of the counters of all the sequences of $\mathcal{T}_X^<$ containing $\mathcal{I}$. □

**Second Step.** It has the purpose to determine the support of the itemsets from the support of the sequences according to the previous theorem. The trees generated in the first step ($\mathcal{T}^<$) are used in order to produce a second set of trees ($\mathcal{T}^>$) in which the itemsets with the relative support counters are represented. Each tree $\mathcal{T}^<$ is read, starting from the tree $\mathcal{T}^<$ of the last item in the Alphabet and proceeding with the trees $\mathcal{T}^<$ of the other items in the decreasing order. The sequences of each tree $\mathcal{T}^<$ are taken with their counters, and from each of them *the subsets containing the Leader of the sequences* are determined and stored in a $\mathcal{T}^>$ tree (the other subsets, not containing the Leader, are determined while reading the other trees). These subsets are the itemsets. The counter associated to the terminal node of the itemset is incremented by the value of the counter of the sequence originating it. In this way, at the end of the reading of a generic tree $\mathcal{T}^<$, the counters of the itemsets originated from the sequences of that tree contain the correct value necessary to determine their support. If the support of an itemset is not sufficient, the itemset is deleted from its $\mathcal{T}^>$ tree.

*Sequence Subset Determination* In order to understand the technique used to produce the itemsets from the sequences, consider the reading of the tree $\mathcal{T}_{\mathtt{D}}^{<}$ of Figure 2, and in particular the sequence $\langle DYZ \rangle$ (see Figure 3).

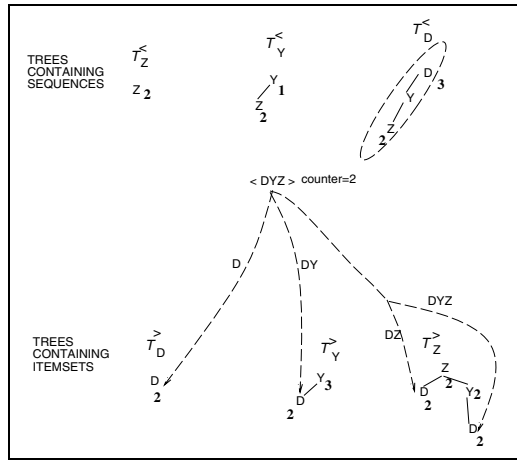


**Fig. 3.** The creation of the itemsets from the sequence $\langle DYZ \rangle$.

At this time, the trees $\mathcal{T}_{\mathtt{Y}}^{<}$ and $\mathcal{T}_{\mathtt{Z}}^{<}$ have been already read; in main memory there are the itemsets originated from the sequences that start with `Y` and `Z`. The generation of the itemsets of the sequence $\langle DYZ \rangle$ consists in the addition of the item `D`, Leader of the sequence, to all the subsets of the remaining portion of the sequence ($\langle YZ \rangle$). These latter subsets are the empty set and {Z},{Y}, {YZ}. The empty set corresponds to the determination of the itemset {D}: the root node of $\mathcal{T}_{\mathtt{D}}^{>}$ is created only if the reading of the database in the first step has proved that item D has a sufficient support. As regards the other subsets, if their support is not sufficient, they are not found in $\mathcal{T}_{\mathtt{Z}}^{>}$ and $\mathcal{T}_{\mathtt{Y}}^{>}$ trees respectively, and no work is wasted considering their supersets. In the positive case, a leaf node containing the Leader `D` is added to the subset in the appropriate tree $\mathcal{T}^{>}$. The algorithm that creates the sequence subsets is reported.

```
procedure create_subsets (sequence S, counter c, list prune_list)
          X=Leader{S};
          list current_sets, previous_sets;
          for all items I ∈ S from last one to first one (I≠ X) do
              if exists T_I^> then
                 leaf = add_leaf(T_I^>.root_node, X, c, prune_list);
                 add T_I^>.root_node to current_sets;
              end if
              for all nodes P in previous_sets do
```

```
                if exists a child node N of P with item I then
                  leaf = add_leaf(N, X, c, prune_list);
                  add N to current_sets;
                end if
              end for
              swap current_sets into previous_sets;
          end for
end procedure
procedure add_leaf (node F, item X, counter c, list prune_list)
          if exists a child node N of F with item X then
            N.count = N.count + c;
          else
              allocate a new node N child of F with item X;
              N.count = c;
              add N to prune_list;
          end if
end procedure
```

### 2.3 Sequences Advantages

`Seq` receives several benefits with the representation in terms of sequences:

1. During the reading of the database, when not enough information is known to eliminate many itemsets, only the sequences are maintained in the main memory: the support of the sequences is a sort of "summary" of the support of the itemsets and enables the saving of many counters in main memory.
2. `Seq` saves CPU execution time because this latter one is not determined only by the total number of candidate itemsets kept in main memory but also by the number of accesses in main memory that each of them requires. `Seq` needs two accesses in main memory when it generates a new itemset and when it updates its support (see the code in Section 2.2), independently of the itemset length. On the contrary, for the generation of an itemset of length `k`, `Apriori` requires `k` accesses. Then, `Apriori` checks for the presence of `k` subsets of length `(k-1)` that requires `k(k-1)` accesses. Finally, when it updates the support of an itemset of length `k` it performs `k` accesses.
3. `Seq` executes the pruning phase very frequently (once for each item with sufficient support). Instead, traditionally, the pruning phase is run once for each level of itemset cardinality, that is about three orders of magnitude less.

### 2.4 Implementation of `Seq`

When very large databases are used, the number of sequences represented in the trees $\mathcal{T}^<$ might be too large to fit in the main memory. We had to change the algorithm to allow the buffer management. The algorithm swaps the content of the trees to disk in the first step and read it in the second one. Each tree

is swapped to a separate file: $\mathcal{T}_{\mathtt{A}}^{<}$ is swapped to file $\mathcal{F}_{\mathtt{A}}$, $\mathcal{T}_{\mathtt{B}}^{<}$ to $\mathcal{F}_{\mathtt{B}}$, and so on. The sequences are written to disk in a compressed form. For example: once that the first sequence of $\mathcal{T}_{\mathtt{A}}^{<}$, $\langle ABCD \rangle$ is written, the second one, $\langle ABDYZ \rangle$ can be represented substituting the prefix items common to the two consecutive sequences ($\langle AB \rangle$) with the prefix length (2). Especially if the transactions have many common sequences this technique saves a great amount of information.

## 3 Evaluation of Algorithm Seq

Three basic parameters characterize a given data mining problem: the total number N of the transactions, the average length L of the transactions and the number I of different items. Let us analyze their influence on the computational work of the presented algorithm and the access times to mass storage.

**Computational work** Computational work involved in step 1, i.e. in the construction of the trees containing sequences, is relatively small and is proportional to the product $L * N$, that is to the size of database. The evaluation of the computational work in step 2, that is in the generation of the trees of the subsets of items, grows exponentially with L and linearly N. However, L is limited and is characteristic of a specific application. Besides, computational time needed to construct the trees of the itemsets is relatively small with respect to the times necessary to read the database and to store and retrieve the trees from disk.

The execution time of Seq is nearly independent of the value of the minimum support. This point is rather important. Indeed, the concept of minimum support has been introduced to reduce the computational time, but it reduces the statistical significance of the search. Above all, if a certain value of the minimum support is reasonable for the itemsets of length equal to 1, it might be enormous for itemsets of length 2 or more. So, the new algorithm might be adopted in searches characterized by very small values of resolution.

**Access time to mass storage** Let $\mathrm{T}_R$ be the time spent to read the database, that is the lower bound of any algorithm. However, $\mathrm{T}_R$ must be increased by the time $\mathrm{T}_F$ spent to save and retrieve $\mathcal{T}^{<}$ trees. Thus the upper bound of the total volume of data transferred with the mass storage amounts to the size of the database plus two times the sizes of the files of sequences. For very long databases this upper bound is slightly larger than the database size.

Two contrasting factors influence $\mathrm{T}_F$ with respect to $\mathrm{T}_R$. Indeed, the volume of data in the $\mathcal{T}^{<}$ trees would be larger than the whole database (because to any transaction there correspond more sequences) for those databases in which transactions have very few common items. This is the case of synthetic databases of the experiments of Section 4. On the other side, repeated transactions require the same information amount of a single transaction: therefore $\mathrm{T}_F < \mathrm{T}_R$ in case transactions are frequently repeated or have many common items. $\mathrm{T}_F$ can be considerably reduced if information represented by the trees of sequences is compressed. In our implementation we have adopted a very simple technique of compression but it would be possible to reduce $\mathrm{T}_F$ with more sophisticated compression techniques even in the case of no repeated transactions.

## 4 Experiments

We have run our implementation of algorithm `Seq` using a PC Pentium II, with a 233 Mhz clock, 128 MB RAM and Debian Linux as operating system. We have worked on the same class of synthetic databases [1] that has been taken as benchmark by most of previous algorithms. Broadly, each transaction of these databases has been generated by addition of some items, extracted in a casually fashion, to a large itemset. For this "semi-casual" content, we believe that this database is not very suitable for an efficient execution of `Seq`: in most of the cases the total dimension of the files containing sequences gets comparable with the database dimension and execution time spent in I/O is not mainly determined by the database reading pass. The results of the experiments are shown in Figure 4.

In the left column of Figure 4 we compare the execution times of `Seq` with `Apriori`, one of the best algorithms. The three experiments analyzed refer to the three classes of databases characterized by increasing transaction length (5, 10, 15). For each database class, we show the execution times for different databases in which the average length of the large itemsets gets the values 2, 4, 6, 8, 12. `Seq` works better than `Apriori` in all the experiments, but the best gains occur when the average itemsets length is comparable with the transaction length. Observe also how `Seq` execution time is almost constant with respect to the average itemsets length, whereas `Apriori` increases the execution times because of the increasing number of reading passes on the database. Notice that, as the value of the minimum support decreases, `Seq` gets much better. This behavior is due to the fact that a certain number of itemsets with higher cardinality values result with sufficient support. In these conditions, `Apriori` must increase the number of reading passes on the database whereas the number of I/O operations performed by `Seq` remains almost constant. Furthermore, even when the number of I/O operations performed by `Seq` is comparable with `Apriori`, the execution times of `Seq` are still lower. Therefore, we have compared the number of candidate itemsets in main memory in order to ascertain whether this one was a favorable factor to `Seq` that could determine a lower CPU processing time. We have noticed that there is not a significant difference between the two algorithms. So, we have concluded that the computational work performed by `Seq` is lower than `Apriori` because of the lower number of accesses to main memory required in the generation and update of candidate itemsets, as already stated in Section 2.3.

The experiments on the scale-up properties with respect to the dimension of the database are shown in the first experiment of the second column. In this experiment the minimum support is fixed to 0.5, but analogous results are given with lower values. We adopted databases with 100, 200, 300 and 400 thousands of transactions. The linear behavior is still verified by both `Seq` and `Apriori` but with very different slopes! These different increases are due again to the number of I/O operations. `Seq` reads once the database, writes once the files with the sequences and then reads a certain number of them. However, the size of the files increase of a little fraction of the database size as this latter one increases. On the contrary, the number of I/O operations performed by `Apriori` are still determined by the repeated reading passes on the database.

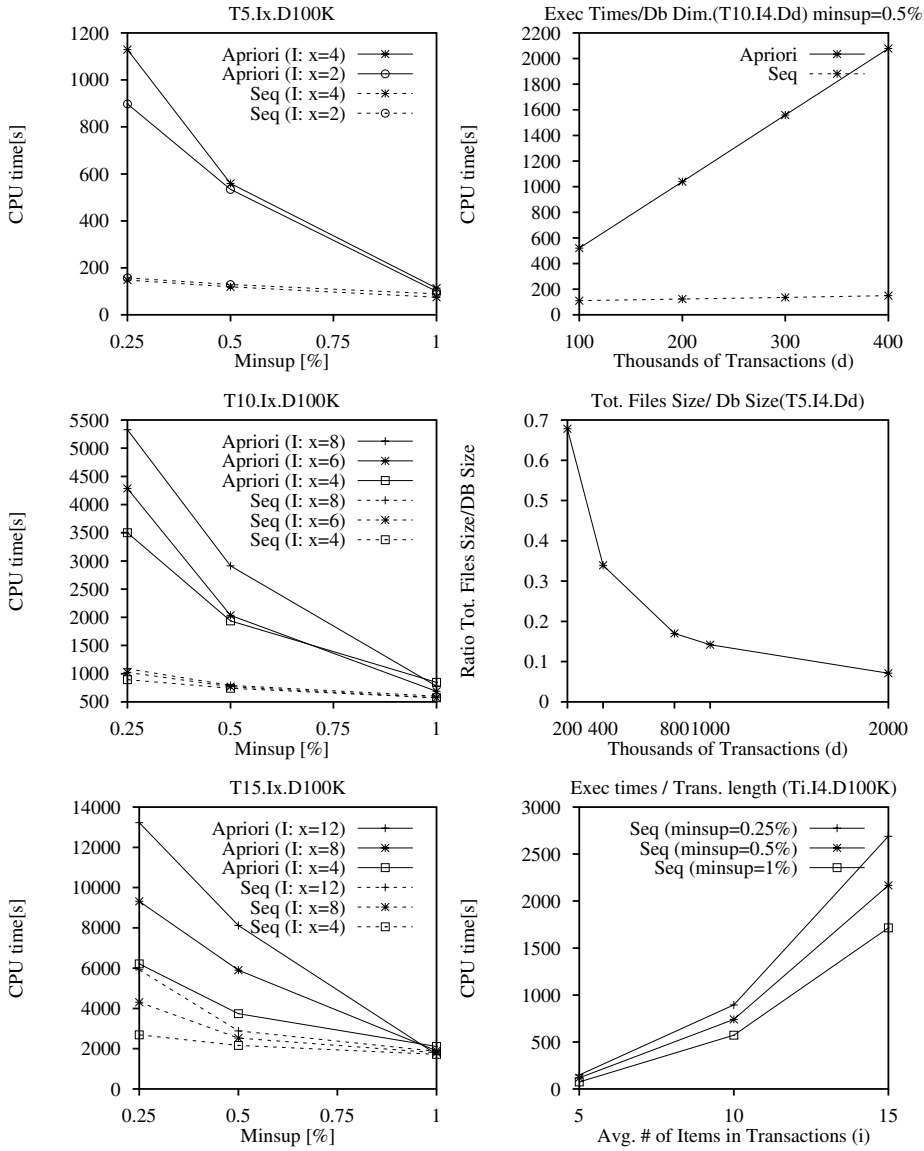


**Fig. 4.** Experiments results with synthetic databases.

The next experiment shows the ratio between the total storage occupation of the files containing the sequences and the database size with respect to different databases having the same statistics but different dimensions. This experiment confirms our previous evaluations. You can notice that for very large databases (2 millions of transactions) the size of the files containing the sequences is only a little fraction (7%) of the database size. In these cases, the I/O operations on the files do not influence the total time spent in performing I/O because this one is mainly determined by the reading pass of the database.

The remaining experiment shows the variation of the execution times of `Seq` with the average transaction length that confirms that the execution time of `Seq` is exponential with respect to the length of the transaction.

## 5 Conclusions and Future Work

A new technique for the discovery of frequent itemset have been presented. It is specifically oriented to databases of very large dimensions and searches of very high resolution. The algorithm `Seq` based on this approach is characterized by an increased processing efficiency, since its execution times are better than `Apriori`, one of the best algorithms of current literature. `Seq` needs only one pass on the database and has a linear behavior, almost constant, with the dimension of the database. Moreover our experiments have shown that `Seq` execution time is nearly constant with respect to the maximum cardinality of the itemsets.

`Max-Miner` [7] and `Pincer-Search` [6], new algorithms presented while this work was in the implementation phase, perform better than `Apriori` on databases with very long itemsets. Further work will compare these algorithms with `Seq`.

## References

1. R.Agrawal, H.Mannila, R.Srikant, H.Toivonen and A.I.Verkamo. Fast discovery of association rules. In *Knowledge Discovery in Databases*, AAAI/MIT Press, 1995.
2. A.Savasere, E.Omiecinski, and S.Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995.
3. J.S.Park, M.Shen, and P.S.Yu. An effective hash based algorithm for mining association rules. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, 1995.
4. S.Brin, R.Motwani, J.D.Ullman, and S.Tsur. Dynamic itemsets counting and implication rules for market basket data. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, vol.26,2 of SIGMOD Record, p.255-264, New York, 1997.
5. H.Toivonen. Sampling large databases for association rules. In *Proc. of the 22nd VLDB Conference*, Bombay, India, 1996.
6. D.I.Lim and Z.M.Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *Proc. of the EDBT'98 Conference*, Valencia, Spain, 1998.
7. R.J.Bayardo. Efficiently mining long patterns from databases. In *Proc. of the SIGMOD'98 Conference*, Seattle, WA, USA, 1998.