Mining for Association Rules without Pruning

M. Arul Arasu P. Sreenivasa Kumar
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai 600036, India.
{arul,psk}@aries.iitm.ernet.in

ABSTRACT

Algorithms for association rule mining first discover all frequent itemsets before producing the association rules. Efficiency of discovering frequent itemsets depends on the candidate generation method. Candidate generation of most of the previous studies, such as Apriori and Pincer Search, adopt Join and Prune procedures. This paper proposes No Pruning Method (NPM), and a new algorithm called All Frequent Itemsets Generation with No Pruning (AFIG_NP). NPM combines the top-down and bottom-up searches of the itemset lattice, and discovers where exactly all the unclassified itemsets are present. AFIG_NP algorithm generates exact number of candidate itemsets without pruning. Furthermore, the proposed candidate generation method itself is a fast one. Extensive experiments have been conducted to study the performance of the proposed algorithm. The proposed algorithm is compared against the Apriori and Pincer Search algorithms. Results show that the proposed algorithm is more efficient and scalable than the other two, for mining both long and short frequent itemsets.

ADVANCES IN DATA MANAGEMENT 2000
Krithi Ramamritham, T.M. Vijayaraman (Editors)
Tata McGraw-Hill Publishing Company Ltd.

1. INTRODUCTION

Association rules have been recognized as an important kind of knowledge that can be extracted from transaction-like data. The problem was initially formulated in the context of market basket analysis [1]. A market basket is a collection of items purchased by a customer in a single transaction. An association rule miner discovers the association among the items such that the presence of a set of items X implies the presence of another disjoint set of items Y. An example of such a rule is "95% of customers who buy Bread and Butter also buy Jam". The following is the mathematical model of the problem [2],

Let $I = \{i_1, i_2, i_3.....i_n\}$ be a set of items. Let D be the set of transactions, where each transaction $T \in D$ is a set of items from I. A set of items is called an itemset. An itemset of cardinality k is called a k-itemset. Items in an itemset are assumed to be sorted in increasing order. A transaction T is said to contain an itemset X, if $X \subseteq T$. The *support of an itemset* X is the percentage of transactions in D that contain X. An association rule is an implication of the form $X \Rightarrow Y$, where $X, Y \subset I$ and $X \cap Y = \phi$. The *support of a rule* $X \Rightarrow Y$ is the support of itemset $X \cup Y$. The *confidence of the rule* $X \Rightarrow Y$ is the ratio of the support of itemset $X \cup Y$ to the support of itemset X. Now the problem is to identify all association rules that have support and confidence not less than user-specified minimum support (MinSup) and minimum confidence (MinConf), respectively.

The discovery of association rules consists of two steps. The first step identifies all frequent itemsets (itemsets whose support is not less than MinSup). The overall performance of mining association rules is determined by this step. The second step generates association rules from these frequent itemsets. This paper addresses the first step.

Motivation: The heuristic to generate the candidate set (set of itemsets that can potentially be frequent) is crucial to the performance of the first step. Most of the previous studies [2, 3, 5, 6, 7] for this task adopt Join and Prune procedures. These procedures perform a large number of membership[1] test computations. In particular, when a database contains large number of long frequent itemsets,

[1] In the generation of C_{k+1} (set of candidate (k+1)-itemsets), Apriori Join takes every pair of elements from L_k (set of frequent k-itemsets) and tests for a match of first (k-1) elements. This equality test is considered as a special case of membership test.

Prune requires more membership tests than Join. Furthermore, the prune operation is an expensive one. Therefore, if an algorithm reduces the number of membership tests, and does not require any pruning (but still produces the right candidate itemsets), the performance can substantially be improved. ⋄

This paper proposes No Pruning Method (NPM), and an algorithm called All Frequent Itemsets Generation with No Pruning (AFIG_NP). NPM constructs a lattice for the itemsets. It combines the top-down and bottom-up searches of the lattice. It discovers where exactly all the unclassified itemsets are present. Based on this method, a procedure called Candidate Itemsets Generation with No Pruning (CIG_NP) is proposed to generate the bottom-up candidate itemsets (C_{k+1}). This procedure is not based on or an extension of any known procedure. CIG_NP builds the candidate itemsets very effectively, and thus reduces the CPU time significantly. It does not require any pruning but still produces the right candidate itemsets as Apriori or DHP. Another procedure called Top-most Border Itemsets Generation (TBIG) is also proposed. It is an extension of MFCS-gen procedure proposed for Pincer-Search [5]. The main aim of TBIG procedure is to avoid redundant support computation of some top-down candidate itemsets, thereby reducing the I/0 overhead considerably. Extensive experiments have been conducted to study the performance of the proposed algorithm. The proposed algorithm is compared against the Apriori and Pincer Search algorithms. Results show that the proposed algorithm is more efficient and scalable than the other two, for mining both long and short frequent itemsets.

The rest of the paper is organized as follows. Section 2 presents the related work, and an overview of two existing algorithms. Section 3 describes the No Pruning Method, AFIG_NP algorithm and a variant algorithm of AFIG_NP. This section also includes two complete examples that illustrate AFIG_NP algorithm, in addition to some partial examples. Section 4 verifies the correctness of the proposed algorithm. Section 5 reports the performance analysis of algorithms. Section 6 concludes the paper.

2. RELATED WORK

Most of the frequent itemset computation algorithms [1, 2, 3, 6, 7] operate in a *bottom-up breadth-first search* manner and is referred as *Bottom-Up approach* [5]. These algorithms start the process with the generation of frequent 1-itemsets and continue till all the maximal length frequent itemsets are generated. A well-known algorithm of this kind is Apriori [2]. Apriori is the first algorithm that uses the property -*if an itemset is infrequent (itemset whose support is less than MinSup), all its supersets will be infrequent*- for pruning the candidate itemsets. Also note that *if an itemset is frequent, all its subsets will be frequent.* DHP [6] enhances Apriori with a hashing scheme. It generates small candidate sets and trims the transaction database effectively. This algorithm collects the upper-bound support for (k+1)-itemsets in pass k. This upper-bound support is used to prune the candidate itemsets. DIC [3] counts the support of candidate itemsets of varying cardinality in the same pass, thereby reducing the number of database scans. It builds a candidate itemset shortly after all its subsets have been determined as frequent, rather than waiting for the current pass to complete. All the above algorithms make multiple passes over the database. Partition algorithm [7] minimizes the I/O cost by making memory-sized partitions of the database. It scans the database only twice. The first scan identifies all local (with respect to the partition) frequent itemsets. The frequent itemsets are computed using Apriori. The second scan identifies all actual (global) frequent itemsets.

The approach that operates in a *top-down breadth-first search* manner is referred as *Top-Down approach* [5]. The algorithms start with n-itemset, where n is the total number of items, and continue in the downward direction to determine the border for infrequent itemsets. The itemsets that are at the border are maximum frequent (none of its superset is frequent and every subset of it is frequent), above the border are infrequent and below the border are frequent. Zaki et al. [9] couples itemset clustering with lattice traversal model to generate maximum frequent itemsets.

The third approach is to combine the *Top-Down* and *Bottom-Up* approaches. An algorithm of this kind Pincer Search [5] makes multiple passes over the database and generates Maximum Frequent Set (MFS, a set containing maximum frequent itemsets). Another algorithm that follows this approach was reported by Zaki et al. [8, 9]. It couples itemset clustering with lattice traversal model to generate MFS. Next two subsections briefly explain Apriori and Pincer Search algorithms.

Recently, Han et al.[4] has proposed an entirely different approach to discover the frequent itemsets. This method scans the original database exactly twice. The first pass discovers all the frequent items. In the second pass, after reading every transaction, the frequent items corresponding to this transaction are incrementally stored into a tree called FP-tree. Then, it develops a method called FP-growth to discover the frequent itemsets.

2.1 Apriori Algorithm

Apriori algorithm [2] works as follows. Let C_k denote the set of candidate k-itemsets and L_k denote the set of frequent k-itemsets. The process starts with $k = 1$ and $C_1 = \{\{i\} \mid i \in I\}$. The following steps are repeated till C_k becomes empty.

- Find the support for every itemset in C_k.
- Compute $L_k = \{X \mid X \in C_k \text{ and } Support(X) \geq MinSup\}$.
- Generate C_{k+1} from L_k using Apriori-Join and Prune procedures. Apriori-Join joins every two elements in L_k if they have the same $(k-1)$-prefix. Prune removes a (k+1)-itemset generated in Join step, if any of its k-element subset is not in L_k.
- Increment the value of k by 1.

2.2 Pincer Search Algorithm

Pincer search [5] combines the *Bottom-Up and Top-Down approaches* to generate MFS. It introduces a top-down candidate set called Maximum Frequent Candidate Set ($MFCS$). $MFCS$ is a set of itemsets satisfying two conditions,

1. Frequent $\subseteq \{Y \mid Y \in 2^X \text{ and } X \in MFCS\}$
2. Infrequent $\cap \{Y \mid Y \in 2^X \text{ and } X \in MFCS\} = \phi$

where Frequent and Infrequent stand respectively for the set of all frequent and infrequent itemsets identified so far.

The algorithm works as follows. Let S_k be the set of infrequent k-itemsets, i.e, $S_k = \{X \mid X \in C_k \ and \ Support(X) < MinSup\}$ and MFS be the set of maximal frequent itemsets, i.e, $MFS = \{Y \mid Y \in MFCS \ and \ Support(Y) \geq qMinSup\}$. The process starts with $k = 1$, $MFCS = \{\{I\}\}$ and $C_1 = \{\{i\}\ \mid i \in I\}$. In each pass, the support for itemsets in $MFCS$ and C_k is computed. L_k, S_k and MFS are immediately identified from C_k and $MFCS$. Now the top-down candidate set, $MFCS$ is updated in such a way that no element in $MFCS$ contains any infrequent itemset identified in the bottom-up approach, i.e, if $X \in MFCS$ and $Y \in S_k$, then $X \not\supseteq Y$. Since this algorithm aims to generate all maximum frequent itemsets, it deletes all k-itemsets that are subsets of new maximum frequent itemset from L_k. This is to avoid the generation of candidate $(k+1)$-itemset that is a subset of some maximum frequent itemset. Bottom-up candidate set, C_{k+1} is generated from L_k, $MFCS$ *and* MFS using Apriori-Join, Pincer Recovery and Prune. Apriori-Join is same as the Apriori-Join explained in the previous subsection. Pincer Recovery recovers two groups of missing candidate (k+1)-itemsets. The candidate itemsets are missed due to the removal of some of the frequent itemsets from L_k after their supersets are identified to be maximum frequent. The missing itemsets are recovered in the following ways,

- Restore all k-itemsets that are subsets of every maximum frequent itemset.
  Perform join with the L_k elements, i.e, a restored k-itemset and a L_k element are joined.

- Restore all k-itemsets that are subsets of every maximum frequent itemset.
  Perform join if both the itemsets are not in the same maximum frequent itemset.
  Note: Otherwise the new (k+1)-itemset will be frequent.

After the recovery, the prune procedure removes an itemset from C_{k+1} if it is not a subset of any $MFCS$ itemset.

3. NO PRUNING METHOD (NPM)

This section explains the working principle of NPM. NPM combines the top-down and the bottom-up approaches, and comes up with the following features,

1. Discovers where exactly all the unclassified itemsets (itemsets identified neither as frequent nor as infrequent) are present.

2. Generates the candidate set C_{k+1} efficiently (with no pruning).

3. Avoids redundant support computation of top-down candidate itemsets.

3.1 Discovering Unclassified Itemsets

Consider a lattice of all possible itemsets from the set of items I. An itemset A in the lattice is covered by an itemset B, if A is a proper subset of B. NPM combines the top-down and the bottom-up searches of the lattice. In each direction, the algorithm maintains one candidate set - $TDCS$(Top-Down Candidate Set) for the top-down direction and C_k for the bottom-up direction. The information gathered in one direction is used to generate the candidate itemsets in the other direction.

Each pass of the algorithm computes the support for itemsets in $TDCS$ and C_k. The infrequent itemset information obtained in the bottom-up direction is propagated to the upward direction of the lattice, and all its supersets will be considered as infrequent itemsets. The new $TDCS$ is generated based on this information. Similarly, if an itemset is frequent (maximum frequent) in the top-down direction, the information is propagated to the downward direction of the lattice so that their subsets will not be considered for the candidate itemsets generation. These subsets are directly put into the set of frequent itemsets, and their support is computed.

Before explaining how the top down and bottom up candidate itemsets are generated, we define the required sets.

Maximal Itemset: An itemset X in a collection C of itemsets is called *maximal itemset* if none of its subset is also present in the collection.

Maximum Frequent Itemset: An itemset X is maximum frequent if all its subsets are frequent and all its supersets are infrequent.

Maximum Frequent Set (MFS): A set is said to be MFS if all its elements are maximim frequent itemsets.

Top Down Infrequent Set (TDIS): TDIS is a set of top down infrequent itemsets for which none of their subset is identified as infrequent.

Top Down Candidate Set (TDCS): TDCS is a set of maximal unclassified itemsets.

Top Down Border Set (M): The union of TDIS, MFS and TDCS sets is said to be top down border set.

C_{k+1}: C_{k+1} is defined to be the difference of the set of all (k+1)-subsets of every element in $TDCS \cup TDIS$ and the set of all (k+1)-subsets of every maximum frequent itemset.

Now, we shall explain how the top down and the bottom up candidate itemsets are generated. First, we have to identify the top most border for the search space of frequent itemsets. The top most border set denoted by M contains all maximal itemsets from the collection of itemsets that are not identified as infrequent in the *bottom-up direction.*

Remarks: Strictly speaking, the top-most border should contain the set of maximum itemsets from the itemsets that are not identified as infrequent, in either top-down or bottom-up directions. If we do so, the run-time efficiency is reduced heavily during the initial stages. Hence the top-down infrequent itemsets continue to be in the border set till the bottom-up direction identifies them as infrequent.

Example: Consider $TDCS = \{\{1, 2, 3, ..., 2000\}\}$ and $C_1 = \{\{1\}, \{2\}....., \{2000\}\}$. The pass 1 identifies $L_1 = C_1$ and $\{1, 2, 3, ..., 2000\}$ as infrequent. Now, our algorithm takes all the k-element subsets of every itemset in M, for the generation of C_k. From $\{1, 2, 3, ..., 2000\}$, all 2-itemsets ($2000C_2 = 1999000$ itemsets) are generated only once. On the other hand, assume all the 1999-subsets of $\{1, 2, 3, ..., 2000\}$ as the top most border itemsets (M), since $\{1, 2, 3, ..., 2000\}$ is infrequent. The number of such 1999-itemsets are 2000. Still, the same candidate set is generated, i.e, number of different itemsets generated are the

same. But the number of itemsets that have been actually generated are 2000 x 1999C_2 (1999000 x 1998). This is 1998 times bigger than the former method. ⋄

Every element in the set M is such that none of its proper subset is infrequent. All the itemsets in M that are neither frequent nor infrequent, constitute the top-down candidate itemsets ($TDCS$).

The bottom-up candidate itemsets are generated from the itemsets that are in the sub-lattices. These sub-lattices are induced by the itemsets in M. The sublattice induced by an itemset X of M consistes of all itemsets that are subsets of X including X. Note that if a sublattice contains an infrequent itemset, then it will be the top element of the sublattice. Any (k+1)-itemset that is a *proper subset* of some itemset in M is either frequent or unclassified itemset. If this (k+1)-itemset is covered by any maximum frequent itemset, it is a frequent itemset. Otherwise it is an unclassified itemset. The set of these unclassified (k+1)-subsets should constitute C_{k+1}. This explains the definition of C_{k+1} given above. Note that C_{k+1} does not include the (k+1)-subsets of maximum frequent itemsets (identified so far). They are considered as identified frequent itemsets and directly put into L_{k+1}. This is the reason for difference operation. The procedure proposed in the next subsection handles this difference operation efficiently.

Example The Figure 1 shows the lattice of itemsets, from the set of items $\{1, 2, 3, 4, 5\}$. The relation cover is not represented using normal lines to maintain the clarity. Assume that the algorithm is at pass 2. In pass 1, algorithm identifies $\{1\}, \{2\}, \{3\}, \{4\}$ and $\{5\}$ as frequent itemsets and $\{12345\}$ as infrequent itemset. Since none of the subsets of $\{12345\}$ is identified as infrequent in the bottom-up direction, all the 2-subsets of $\{12345\}$ is considered as candidate 2-itemsets. In pass 2, $\{15\}$ is identified as infrequent in bottom-up direction. It is represented by circle. Since $\{15\}$ is infrequent, all its supersets are taken as infrequent. This is shown in the figure. The itemsets denoted by the box represent the frequent itemsets. The itemsets marked by the cross indicate that they are identified as infrequent in both the directions. The unmarked itemsets are the unclassified itemsets. No itemset is identified as maximum frequent itemset, so far.

Now, the algorithm has to construct the top-most border candidate itemsets. It identifies $\{1234\}$ and $\{2345\}$ as new top-most border itemsets (M). All the unclassified 3-itemsets are considered as candidate 3-itemsets. They are nothing but the union of all the 3-subset of every itemset in M. Note that no maximum frequent itemset is identified so far. ⋄

3.2 Efficient Generation of Candidate Itemsets

This subsection proposes a new procedure called CIG_NP (Candidate Itemsets Generation with No Pruning), for bottom-up candidate generation. CIG_NP is completely a new procedure, that is not based on or an extension of any known procedure used in algorithms such as Apriori or Pincer-Search. This procedure does not require any pruning. The candidate itemsets are generated directly from the topmost border itemsets.

The topmost border itemsets M is divided into three sets, 1) MFS - the set containing the maximum frequent itemsets 2) $TDIS$ - Top-Down Infrequent Set (the set containing infrequent itemsets from M) 3) $TDCS$ - the set containing the top-down candidate itemsets. Every itemset in M is such that no proper subset of it is infrequent.

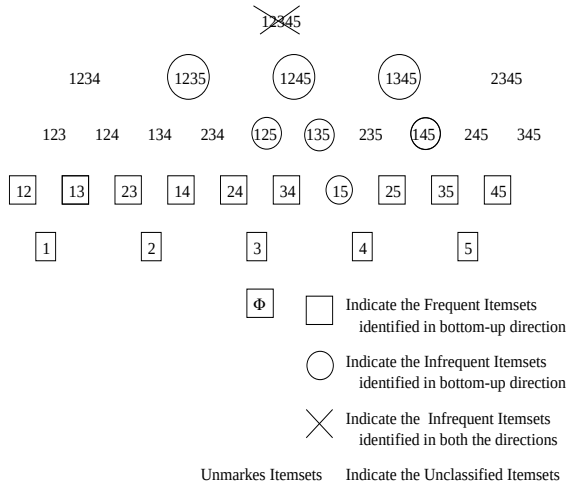


Figure 1: The lattice, after pass 2 counting phase. M will be $\{\{1234\}, \{2345\}\}$. C_3 = Union of all the 3-subsets of every itemset in M. No maximum frequent itemset is identified so far.

Consider the itemsets in the sub-lattices that are induced by the itemsets in M. A (k+1)-itemset in the sub-lattices is put into C_{k+1}, if the itemset is not covered by any maximum frequent itemset. If a (k+1)-itemset is covered by any maximum frequent itemset, then it is directly put into L_{k+1}. The remaining elements for L_{k+1} are added, after the support for C_{k+1} is computed.

Let A denote the set of all (k+1)-subsets of every itemset in ($TDIS \cup TDCS$) and B denote the set of all (k+1)-subsets (proper) of every itemset in MFS. Then, C_{k+1} is defined as $A - B$. $L_{k+1} = B \cup$ {Frequest Itemsets in C_{k+1}}.

The procedure works as follows. It takes a L_k element (say X) and tests if it is a subset of any itemset (say Y) in MFS. If X is a subset of Y, the index (say j) where match for k^{th} element of X occurs in Y, is found out. Then, the procedure adds all the elements in Y that are having indices greater than j, as singleton itemsets, into a set[2] P_1. The same test is done for all the itemsets in MFS, and the resulting elements are added into P_1 in a similar manner.

The same method is repeated for the same L_k element with all the itemsets in ($TDIS \cup TDCS$), instead of MFS. The resulting singleton elements are added into a set Q_1. Let R_1 be $Q_1 - P_1$. Now X is extended with every element in R_1, and the extended itemset is added into the set C_{k+1}. Also, X is extended with every element in P_1, and the extended itemset is added into the set L_{k+1}. The entire procedure is repeated for all the elements in L_k.

Example : Let $TDIS = \{\{1\ 2\ 3\ 4\ 5\ 6\ 9\}, \{3\ 4\ 5\ 6\ 8\}\}$, $MFS = \{\{2\ 3\ 4\ 5\ 6\ 7\ 9\}\}$ and $TDCS = \phi$.
Consider an itemset {3 4 5}, which is an element in L_3.

[2]The procedure CIG_NP maintains 3 temporary variables representing the sets P_1, Q_1 and R_1. Each set contains set of singleton itemsets.

Now, {3 4 5} is tested against the maximum frequent itemset {2 3 4 5 6 7 9}, the match occurs at index 4. The items that are having indices greater than 4 are added into P_1. P_1 becomes $\{\{6\},\{7\},\{9\}\}$. {3 4 5} is also tested against all the itemsets in $TDIS$. Q_1 becomes $\{\{6\},\{9\}\}$, after {3 4 5} is tested against {1 2 3 4 5 6 9}. Finally, {3 4 5} is tested against {3 4 5 6 8}, Q_1 becomes $Q_1 \cup \{\{6\},\{8\}\}$ $= \{\{6\},\{8\},\{9\}\}$. There is no further change in Q_1, since $TDCS = \phi$. Now, $P_1 = \{\{6\},\{7\},\{9\}\}$ and $Q_1 = \{\{6\},\{8\},\{9\}\}$. R_1 will be $\{\{8\}\}$. Every element in R_1 is extended (merged) with {3 4 5} to produce some candidate itemsets, $C_{k+1} = C_{k+1} \cup \{\{3\ 4\ 5\ 8\}\}$. The elements in P_1 are extended with {3 4 5} to produce some frequent itemsets, $L_{k+1} = L_{k+1} \cup \{\{3\ 4\ 5\ 6\},\{3\ 4\ 5\ 7\},\{3\ 4\ 5\ 9\}\}$.

```
Procedure : Candidate Itemsets Generation with No Pruning (CIG_NP)
Input : TDCS, TDIS, MFS, L_k        Output : C_{k+1}, L_{k+1}
1. P_1 = φ, Q_1 = φ     /* singleton itemsets */
2. for all itemsets l ∈ L_k
3.       for all itemsets m ∈ MFS and | m |> k + 1
4.            if l ⊂ m     /* Suppose m.item_j = l.item_k */
5.                 for i = j+1 to | m |
6.                      P_1 = P_1 ∪ {m.item_i}
7.       for all itemsets m ∈ TDCS ∪ TDIS and | m |> k + 1
8.            if  l ⊂ m     /* Suppose m.item_j = l.item_k */
9.                 for i = j+1 to | m |
10.                     Q_1 = Q_1 ∪ {m.item_i}
11.      R_1 = Q_1 − P_1
12.      for i = 1 to | R_1 |
13.           C_{k+1} = C_{k+1} ∪ {l ∪ {R_1.item_i}}
14.      for i = 1 to | P_1 |
15.           L_{k+1} = L_{k+1} ∪ {l ∪ {P_1.item_i}}
END.
```

3.3 Frequent Itemset Recovery with No Pruning (FIR_NP)

Once C_{k+1} becomes empty[3], there may be some MFS itemsets whose size is greater than (k+1). The algorithm has to find the support for the subsets (size $\geq$ k+1) of these maximum frequent itemsets. The FIR_NP procedure is used to generate only L_{k+1}, from MFS.

```
Procedure : Frequent Itemsets Recovery with No Pruning (FIR_NP)
Input : L_k, MFS      Output : L_{k+1}
1. for all itemsets f ∈ L_k
2.       for all itemsets m ∈ MFS and | m |> k + 1
3.            if f ⊂ m /* Suppose m.item_j = f.item_k */
4.                 for i = j+1 to | m |
5.                      P_1 = P_1 ∪ { m.item_i}
6.       for i = 1 to | P_1 |
7.            L_{k+1} = L_{k+1} ∪ {f ∪ {P_1.item_i}}
END.
```

3.4 Avoiding Redundant Support Computation of Itemsets

This subsection proposes a new procedure for top most border itemset generation(TBIG). This procedure is an extension of MFCS-gen procedure proposed in Pincer-Search [5]. MFCS-gen procedure leads to redundant support computation of certain itemsets. We identify the situations where the redundant computation takes place, and propose modifications to avoid the unnecessary computation, thus improving the performance significantly. Intuitively speaking, the redundant computation takes place while dealing with the itemsets that are closer to or at the convergence point (the point where the bottom-up and top-down approaches meet). Since all the maximal length frequent itemsets are not of same length, redundant generation takes place at different points in time. Hence, they are classified with respect to the current pass k.

The topmost border itemsets generation procedure works as follows. Consider an itemset (say $Q \in M$) of size m whose subset (say $R \in S_k$) is infrequent in the bottom-up direction. Now the itemset Q is replaced by k or lesser than k different itemsets that are proper subsets of Q. Each such subset (say X) is of size m-1, has all the items except one of the items in R and no proper superset of X is present in M. This action corresponds to lines 1-9 of the algorithm. The operation of lines 1-9 is as same as MFCS-gen algorithm in Pincer-Search [5]. Lines 10-16 are new, and avoid the redundant support computation of itemsets.

At this point of the algorithm (after line 9), the set M contains some new as well as old border itemsets. The old border itemsets in M are either frequent or infrequent itemsets. The new border itemsets in M are either frequent or unclassified itemsets, since the new border itemsets are the maximum itemsets from the collection of itemsets that are not identified as infrequent. See the Figure 1. Now, consider any new border itemset (say P) of size k. Since the algorithm has completed the counting phase of pass k, it has the information about all the itemsets of size k. Each k-itemset is classified either as frequent or as infrequent itemset. Since P is a new border itemset and of size k, P will be a frequent itemset. Since P is a border itemset, it is a maximum frequent itemset. Hence, every such k-itemset is put into MFS. The similar argument also applies for new border itemsets of size (k-1). These itemsets are also put into MFS. The remaining new border itemsets are put into $TDCS$. This change is incorporated in lines 10 to 12 of the TBIG procedure. In Pincer-Search[5], the support for these k and (k-1) itemsets will be computed in the next pass, and they will be added to MFS.

Consider an infrequent itemset (say Q) of size (k+1) in M. By the definition of M, none of the subsets of Q is infrequent in the bottom-up direction. Since the algorithm has completed the counting phase of pass k, all the proper subsets of Q are frequent (identified in bottom-up process). Since Q is infrequent and all its subsets are frequent, every k-subset of Q is a maximum frequent itemset, provided the k-itemset is not covered by any other itemset in M. Hence, Q is replaced by all such k-subsets of Q. The similar argument also applies for the infrequent itemsets of size k in M. Hence, all such k-itemsets are replaced by their (k-1)-subsets. All these (k-1)-subsets are maximum frequent itemsets. The change is incorporated in lines 13 and 16 of the TBIG algorithm. In Pincer-Search, Q will be considered by C_{k+1} and support will be computed in the next pass [(k+1)]. At pass (k+1), all the subsets of Q will be added into $TDCS$ and counted in pass (k+2), for the second time. Many a times this redundant computation alone leads to two more passes for Pincer-Search.

[3] At this point, all the maximum frequent itemsets have been found out. And $TDCS = TDIS = \phi$

Example : Let $M = \{\{1\ 2\ 3\ 4\ 5\}\}$ and a S_2 element be $\{1\ 5\}$. Now, $\{1\ 2\ 3\ 4\ 5\}$ is replaced by two itemsets that are of size 4 in such a way that $\{1\ 5\}$ is not a subset in any of these two itemsets. These itemsets are $\{1\ 2\ 3\ 4\}$ and $\{2\ 3\ 4\ 5\}$.

Example : Let $M = \{\{4\ 5\ 6\ 7\ 8\ 9\}\}$ and a S_3 element be $\{4\ 5\ 7\}$. Now, $\{4\ 5\ 6\ 7\ 8\ 9\}$ is replaced by three itemsets that are of size 5 in such a way that $\{4\ 5\ 7\}$ is not a subset in any of these three itemsets. These itemsets are $\{5\ 6\ 7\ 8\ 9\}$, $\{4\ 6\ 7\ 8\ 9\}$ and $\{4\ 5\ 6\ 8\ 9\}$.

Procedure : Topmost Border Itemsets Generation (TBIG)
Input : $TDIS$, MFS **and Infrequent Set** S_k
Output : $TDIS$, MFS **and** $TDCS$.
1. $NewItemsets = \phi$ **New border Itemsets**
2. **for all itemsets** $s \in S_k$
3. **for all itemsets** $m \in (TDIS \cup NewItemsets)$
4. **if** $s \subseteq m$
5. **if** $(m \in TDIS)$ $TDIS = TDIS - \{m\}$
6. **else** $NewItemsets = NewItemsets - \{m\}$
7. **for all items** $e \in s$
8. **if** $m - \{e\}$ **is not a subset of any itemset in** M
9. $NewItemsets = NewItemsets \cup \{m - \{e\}\}$
10. $MFS = MFS \cup \{X \mid X \in NewItemsets\ and\ |X| = k\ or\ k-1\}$
11. **NewITemsets = NewItemsets - {X|X∈ NewItemsets and | X | = k or k-1 }**
12. $TDCS = NewItemsets$
13. $MFS = MFS \cup$ **{k-subsets of every (k+1)-itemset in** $TDIS$ **if k-subset is a maximum itemset for M }**
14. $TDIS = TDIS - \{X \mid X \in TDIS\ and\ |X| = k+1\}$
15. $MFS = MFS \cup$ **{(k-1)-subsets of every k-itemset in** $TDIS$ **if (k-1)-subset is a maximum itemset in M}**
16. $TDIS = TDIS - \{X \mid X \in TDIS\ and\ |X| = k\}$
END

3.5 All Frequent Itemsets Generation with No Pruning (AFIG_NP)

This subsection presents the complete algorithm of NPM model that combines the bottom-up and top-down approaches. The algorithm starts with $TDCS = \{\{I\}\}$ $C_1 = \{\{i\} \mid i \in I\}$ and $L_1 = \phi$. Each pass computes the support for these sets. The sets MFS and L_k are updated according to the support information. New itemsets are generated for $TDCS$, whenever some itemset is identified in the bottom-up direction (C_k). At each pass, top most border itemsets (M) are updated, and the bottom-up candidate itemsets (C_{k+1}) are generated. Along with C_{k+1}, some of the (k+1)-frequent itemsets that are subsets of the maximum frequent itemsets, are also generated, and the support is computed for them in the counting phase. This process ends when the bottom-up candidate set becomes empty. Then, the frequent itemsets that are subsets of itemsets in MFS, are generated and the support is computed for them. The algorithm terminates, when all the frequent itemsets and their support have been computed.

Algorithm : All Frequent Itemsets Generation with No Pruning (AFIG_NP)
1. $k = 1$, $C_1 = \{\{i\} \mid i \in I\}$
2. $TDCS = \{I\}$, $MFS = \phi$, $TDIS = \phi$
3. **while** $C_k \neq \phi$
4. **Read the database and compute the support for itemsets in** C_k, L_k **and** $TDCS$
5. $L_k = L_k \cup \{X \mid X \in C_k\ and\ Support(X) \geq MinSup\}$
6. $S_k = \{X \mid X \in C_k\ and\ Support(X) < MinSup\}$
7. $MFS = MFS \cup \{X \mid X \in TDCS\ and\ Support(X) \geq MinSup\}$
8. $TDIS = TDIS \cup \{X \mid X \in TDCS\ and\ Support(X) < MinSup\}$
9. **Call TBIG Procedure**
10. **Call CIG_NP Procedure**
11. **k = k + 1**
12. **endwhile**
13. **while** $(L_k \neq \phi)$
14. **Read the database and compute the support for itemsets in** L_k.
15. **Call FIR_NP procedure**
16. **k = k + 1**
17. **endwhile**
END.

Note that the major problem with most of the top-down and the combined top-down and bottom-up approaches is that the support is not computed for *all* the frequent itemsets. They aim to produce the MFS. This results in overhead during the rule generation, since the support needs to be computed for some frequent itemsets whose support is not computed before the rule generation. But, AFIG_NP algorithm computes the support for all the frequent itemsets, like Apriori does. During rule generation, the support computation is not performed for any itemset.

3.6 Complete Examples

In this subsection, we give two examples. The first example illustrates our algorithm, in the absence of maximum frequent itemsets and the second example illustrates it, in the presence of maximum frequent itemsets. These two examples use the same database but different MinSup values. The following is the database that we take, to illustrate our examples.

1	1 2 3 4 5
2	1 2 3 4
3	1 2 3 4
4	2 3 4 5

Example: Consider Table 1 and assume that MinSup is 2 transactions, i.e, 50%.

Step 1:
k = 1, $C_1 = \{\{1\},\ \{2\},\ \{3\},\ \{4\},\ \{5\}\}$
Step 2:
$TDCS = \{\{1\ 2\ 3\ 4\ 5\}\}$
$MFS = \phi$
$TDIS = \phi$

Pass 1

Step 5:
$L_1 = \{\{1\},\ \{2\},\ \{3\},\ \{4\},\ \{5\}\}$
Step 6:
$S_k = \phi$
Step 8:
$TDIS = \{\{1\ 2\ 3\ 4\ 5\}\}$
Step 10:
$C_2 = \{\{1\ 2\},\ \{1\ 3\},\ \{1\ 4\},\ \{1\ 5\},$
$\{2\ 3\},\ \{2\ 4\},\ \{2\ 5\},$
$\{3\ 4\},\ \{3\ 5\},\ \{4\ 5\}\}$

Pass 2

Step 5:
$L_2 = \{\{1\ 2\},\ \{1\ 3\},\ \{1\ 4\},$
$\{2\ 3\},\ \{2\ 4\},\ \{2\ 5\},$
$\{3\ 4\},\ \{3\ 5\},\ \{4\ 5\}\}$
Step 6:
$S_k = \{\{1\ 5\}\}$
Step 9:
$TDCS = \{\{1\ 2\ 3\ 4\},\ \{2\ 3\ 4\ 5\}\}$
Step 10:
$C_3 = \{\{1\ 2\ 3\},\ \{1\ 2\ 4\ \},\ \{1\ 3\ 4\},$
$\{2\ 3\ 4\},\ \{2\ 3\ 5\},\ \{2\ 4\ 5\},\{3\ 4\ 5\}\}$

Pass 3

Step 5:
$L_3 = \{\{1\ 2\ 3\},\ \{1\ 2\ 4\ \},\ \{1\ 3\ 4\},$
$\{2\ 3\ 4\},\ \{2\ 3\ 5\},\ \{2\ 4\ 5\},\{3\ 4\ 5\}\}$
$\{2\ 3\},\ \{2\ 4\},\ \{2\ 5\},$
$\{3\ 4\},\ \{3\ 5\},\ \{4\ 5\}\}$
Step 6:
$S_k = \phi$
Step 7:
$MFS = \{\{1\ 2\ 3\ 4\},\ \{2\ 3\ 4\ 5\}\}$
Step 10:
$C_4 = \phi$

Example: Consider Table 1 and assume that MinSup is 3 transactions, i.e, 75%.

Step 1:
k = 1, $C_1 = \{\{1\},\ \{2\},\ \{3\},\ \{4\},\ \{5\}\}$
Step 2:
$TDCS = \{\{1\ 2\ 3\ 4\ 5\}\}$
$MFS = \phi$
$TDIS = \phi$

Pass 1

Step 5:
$L_1 = \{\{1\},\ \{2\},\ \{3\},\ \{4\}\}$
Step 6:
$S_k = \{5\}$
Step 8:
$TDIS = \{\{1\ 2\ 3\ 4\ 5\}\}$
Step 10:
$TDCS = \phi\ TDIS = \{\{1\ 2\ 3\ 4\}\}$
Step 10:
$C_2 = \{\{1\ 2\},\ \{1\ 3\},\ \{1\ 4\},$
$\{2\ 3\},\ \{2\ 4\},\ \{3\ 4\}\}$

Pass 2

Step 5:
$L_2 = \{\{1\ 2\},\ \{1\ 3\},\ \{1\ 4\},$
$\{2\ 3\},\ \{2\ 4\},\ \{3\ 4\}\}$
Step 7:
$MFS = \{\{1\ 2\ 3\ 4\}\}$
Step 9:
$TDCS = \phi$
Step 10:
$L_3 = \{\{1\ 2\ 3\},\ \{1\ 2\ 4\ \},\ \{1\ 3\ 4\},$
$\{2\ 3\ 4\}\}$
$C_3 = \phi$

Pass 3

Step 14:
Read the database.
Step 15:
$L_4 = \phi$

3.7 Variation : Maximum Frequent Itemsets Generation with No Pruning (MFIG_NP)

The AFIG_NP algorithm is modified to give an algorithm called MFIG_NP that generates the maximum frequent itemsets. In addition to generating all the maximum frequent itemsets, MFIG_NP also generates some frequent itemsets that are not maximum frequent itemsets. These frequent itemsets are generated in the bottom-up direction. The lines from 1 to 12 of the algorithm AFIG_NP, and the lines from 1 to 13 of CIG_NP procedure, will perform this function. Lines 14 and 15 of CIG_NP are not needed. These two lines consider the subsets of maximum frequent itemsets. There is no need for FIR_NP procedure, that recovers the frequent itemsets that are the subsets of the maximum frequent itemsets. Note that Pincer-Search aims to generate maximum frequent itemsets. Hence, for the purpose of fair comparison, we have implemented MFIG_NP algorithm also.

4. CORRECTNESS OF THE ALGORITHM

Lemma 1: Any itemset that is not a subset of some itemset in M is infrequent.
Proof: Let $M = \{\{X_1\}, \{X_2\}, ...\{X_r\}\}$. The set of all possible subsets of itemsets in M is, $P_1 = \{Y \mid Y \in 2^X \textit{ and } X \in M\}$. The set P_1 gets affected only when TBIG procedure is called. This procedure replaces each itemset in M that contains $S \in S_k$ with different itemsets in such a way that none will contain S. The number of such different itemsets is always k, provided any of their superset is not present in M already. After S is considered by TBIG procedure, $New\ M = \{\{Y_1\}, \{Y_2\}, ...\{Y_t\}\}$. All possible subsets of itemsets in $New\ M$ is $P_2 = P_1 - \{Q \mid Q \in P_1 \textit{ and } Q \supseteq S\}$. Processing all S_k elements has the similar effect. Hence the lemma. $\diamond$

Lemma 2: C_{k+1} generated at pass k is such that C_{k+1} = Difference of the set of all $(k+1)$-subsets (proper) of every itemset in $(TDIS \cup TDCS)$ and the set of all itemsets that are subsets of any maximum frequent itemset.
Proof : C_{k+1} is generated by applying CIG_NP procedure. It is clear from the construction of CIG_NP that C_{k+1} satisfies the above assertion. Hence the lemma. $\diamond$

Theorem 1: All itemsets in C_{k+1} are unclassified and all unclassified itemsets of size (k+1) are in $C_{k+1} \cup TDCS$ itemsets of size (k+1), $k \geq 0$.
Proof:
Part 1: All itemsets in C_{k+1} are unclassified
Consider an itemset $X \in C_{k+1}$. If X is a frequent itemset, then either X is a maximum frequent itemset or a subset of maximum frequent itemset. Due to Lemma 2, X can not be in C_{k+1}. If X is a infrequent itemset, then X can not be in C_{k+1} due to the property of M that all the itemsets in M are such that none of its proper subset is infrequent. Therefore, X is an unclassified itemset.

Part 2: All unclassified itemsets of size (k+1) are in $C_{k+1} \cup TDCS$ itemsets of size (k+1).

According to Lemma 1, an itemset that is not a subset of some itemset in M, is infrequent. Hence, every unclassified itemset is a subset of some itemset in M. By Lemma 2, it is assured that C_{k+1} does not contain any frequent or infrequent itemsets identified so far. Hence the theorem. $\diamond$

5. PERFORMANCE EVALUATION

This section reports the experimental results. The experiments were performed on a 450 MHZ Pentium III PC with 64MB main memory and an attached 4.5GB disk, running on Linux environment. The algorithms Apriori[2], Pincer-Search[5], AFIG_NP and MFIG_NP are implemented in C++. The important data structures used in all the algorithms are Hash Tree and Hash Table that are described in [2]. We use a two dimensional array of counters for Pass 2. The use of two dimensional array, rather than Hash Tree for Pass 2, has a major performance impact [5, 6]. The synthetic datasets are generated by the program designed by IBM Quest project using the procedure given in [2]. Datasets with varying parameters - in size, average length of transaction, average size of frequent itemsets and the number of itemsets- are given as input to the programs. The algorithms are compared in four aspects,

1. **Number of candidates generated, and membership tests performed :** Number of candidate itemsets includes the top-down (if any) as well as the bottom-up candidate itemsets. In case of Apriori and Pincer-Search, it is the number of candidate itemsets that have been generated before pruning takes place. But, the membership tests include the pruning phase also.

2. **Time Taken for the candidate generation process (T_{cg}) :** T_{cg} = Total time taken for the generation of top-down (if any) and bottom-up candidate itemsets.

3. **Execution Time :** Run time (total execution time) is used as the measure. Run time measure is better than CPU time measure, since CPU time considers only the cost of the CPU resources.

4. **Scalability :** The scalability experiments consider the number of items (50 to 10000), average length of the transaction (10 to 60) and the number of transactions (1 million to 10 million).

5.1 Number of Candidates generated, and membership tests performed

Experiments show that Apriori and Pincer algorithms generate a lot of unnecessary candidate itemsets before pruning. In initial passes, they generate more than twice the number of necessary candidates. Generally the problem is very acute in candidate 3-itemset generation. This is due to the fact that in pass 2, the algorithm generates more frequent and infrequent itemsets than any other passes which leads to generation of more number of unnecessary candidate itemsets. Pass1 of the Apriori always generates one candidate less than AFIG_NP. Pass2 of the Apriori generates equal (if $MFS = \{\{I\}\}$), or one candidate less than AFIG_NP. Pincer and MFIG_NP generate equal number of candidates in pass 1 and 2. Table 1 compares the number of candidates generated in other passes. The first row indicates the pass number. Table 2 compares the number of membership tests performed in the entire run of the algorithm. As the length and the number of maximal frequent itemsets increase, the number of membership tests performed by the procedure Prune increases heavily, where as a very less number of membership computations is performed by the proposed algorithms. Hence, the proposed algorithms can meet the scale-up properties well. Please note that, Table 1 compares Apriori with AFIG_NP (since both generate all frequent itemsets), and Pincer-Search with MFIG_NP (since both generate maximum frequent itemsets). However, Pincer-Search takes more time and performs more number of membership tests than AFIG_NP. Recall that AFIG_NP algorithm is restricted to generate maximum frequent itemsets, and named as MFIG_NP algorithm.

Pass No	AFIG_NP	Apriori	MFIG_NP	Pincer
3	31931	97521	31931	97534
4	22958	62416	22958	62525
5	8731	19138	8690	17739
6	3563	4936	3110	4576
7	2132	3294	1555	1605
8	1033	2334	597	781
9	213	398	51	209
10	85	165	10	48
11	32	52	0	27
12	7	21	0	18
13	0	17	0	0

Table 1: No.of candidates :: T20.I15.D400k

Dataset	AFIG_NP	Apriori	MFIG_NP	Pincer
T40.I35.D200k	42169	618936	30296	300469
T40.I30.D200k	25686	504789	19686	446876
T40.I25.D200k	19389	258700	16986	214734
T40.I20.D200k	6493	82361	5812	74461

Table 2: Membership Test Comparison

5.2 Time Analysis of Candidate Generation

All the algorithms are compared for decreasing values of MinSup on four different datasets, and the results are presented in Figure 2 and 3. As the MinSup value decreases, the number and the size of frequent itemsets increases. Apriori and Pincer-Search algorithms thus have to do more membership tests. This results in sudden raise in curve for these two algorithms, where as in AFIG_NP and MFIG_NP the curve raises gradually. Note that the candidate generation time of the algorithms is not directly proportional to the number of membership tests performed by these algorithms, due to the unit time for individual membership test. The unit time varies heavily even within the pass itself for AFIG_NP, MFIG_NP and Pincer-Search algorithms, since they consid-

er the itemsets of different cardinalities (in M) in the same pass.

5.3 Execution Time

Execution times of all the algorithms are compared for decreasing values of MinSup on four different datasets, and the results are presented in Figure 4 and 5. As the MinSup value decreases, the performance gap between the proposed algorithms and the other two algorithms increase. This shows that the proposed algorithms are scalable even for small support values, and hence, can mine the long frequent itemsets efficiently. Note that even though the number of itemsets that are compared against the database is slightly more than Apriori (since Apriori has no top-down process), it does not affect the performance because of the time that is saved in candidate generation process.

5.4 Scale-up Experiments

The scale-up experiments are done with respect to the number of items (50 to 10000), average length of transactions (10 to 60) and the number of transactions (1 Million to 10 Million). As the number of items decreases considerably and/or the average length of transaction increases, the average support for an itemset increases. This results in increasing the number of frequent itemsets. In situations like these, the Apriori and Pincer-Search algorithms perform rather poorly; their curves raise suddenly, whereas the curve for proposed algorithms raises gradually, as can be seen in Figures 5 and 6. The execution times in Figure 5 are normalized with respect to the times for 1000 items in the first graph, and 10000 items in the second graph, and the execution times in Figure 6 are normalized with respect to the times for the average length of transactions 10. The proposed algorithms also scale well with respect to the number of transactions. All the algorithms scale linearly. Refer the Figure 7. But the curves for the proposed algorithms have the least slope. In Figure 7, the execution times are normalized with respect to the times for 1 million transactions.

5.5 Comparison With FP-tree Based Method

Recently, J.Han et al. has proposed a method based on FP-tree. This method scans the database exactly twice. During the second pass, it builds a FP-tree. FP-tree effectively compresses and stores the database. Then, it develops FP-growth method to discover the frequent itemsets.

In their experiments, it was observed that the size of the FP-tree is usually small and can fit in memory even for very large size datasets. While I/O is the biggest overhead in other studies such as Apriori, Pincer-Search and our method, this method seems to reduce the I/O overhead to a greater extent, without causing any other side effects. Hence, we believe that this method can possibly outperform our method. Due to the lack of time, we could not do the implementation. In near future, we plan to implement this method and perform a comparative study against our method.

6. CONCLUSIONS

This paper has closely examined the Apriori and Pincer Search algorithms for mining association rules, and showed that the candidate generation process of these algorithms can greatly be improved. The proposed method discovers where exactly all the unclassified candidate itemsets are present. AFIG_NP algorithm that follows this model, builds the candidate itemsets effectively, and requires no pruning. Furthermore, the algorithm avoids the redundant support computation. Extensive experiments have been conducted, and the results showed that the proposed algorithm is more efficient and scalable than the other two, for mining both long and short frequent itemsets.

Acknowledgments We thank K.Arthi and K.Srinathan for their comments and suggestions.

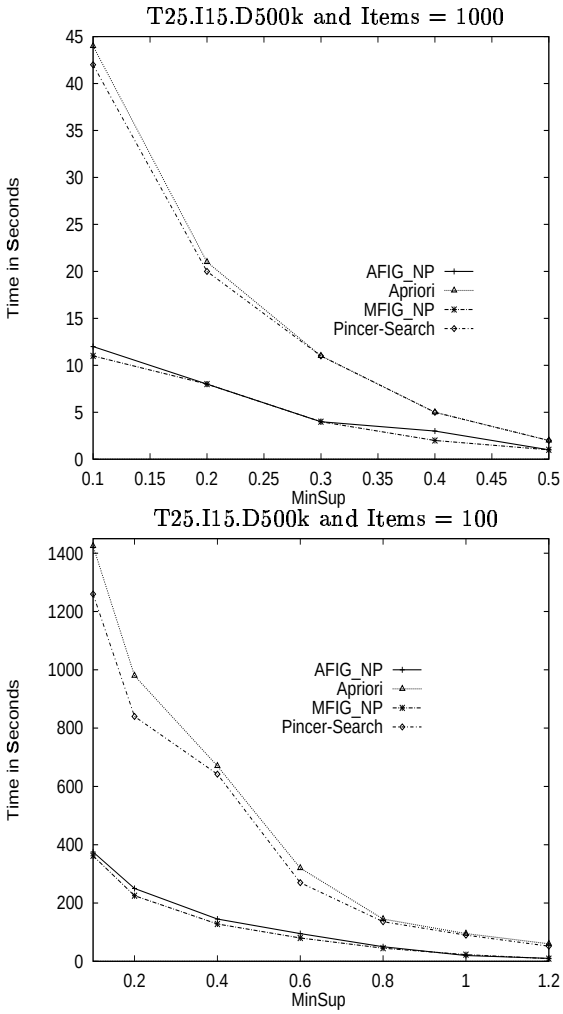


Figure 2: Time Analysis of Candidate Generation process

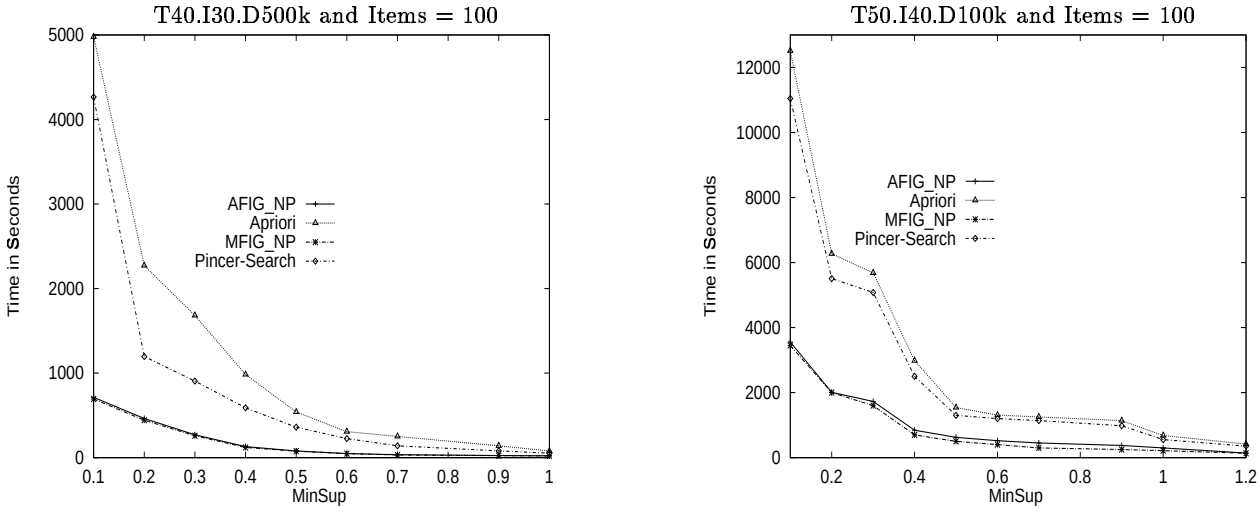


Figure 3: Time Analysis of Candidate Generation process

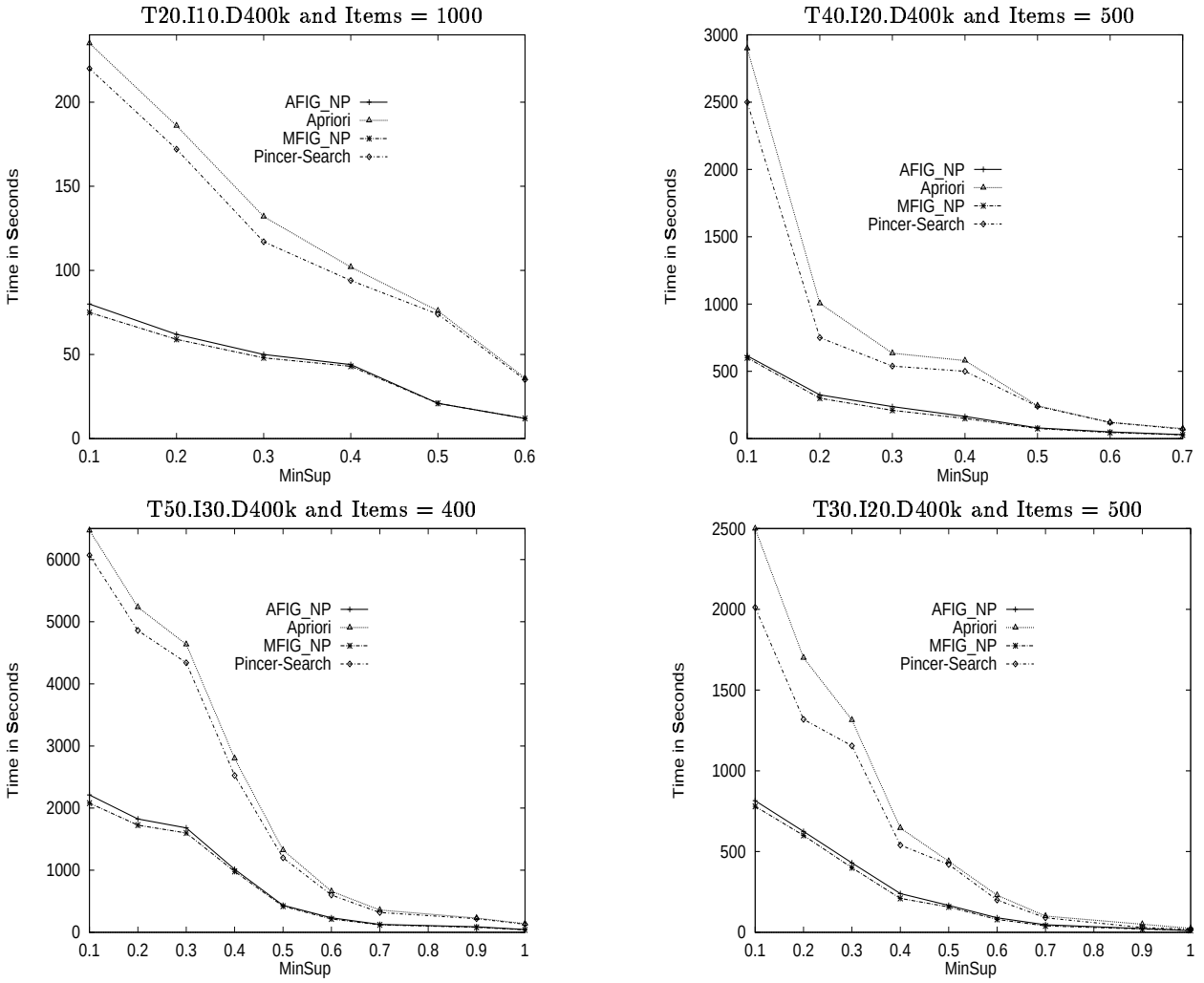
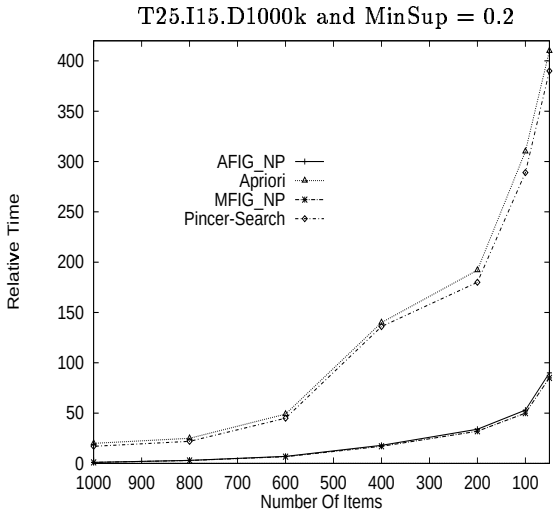


Figure 4: Execution Time



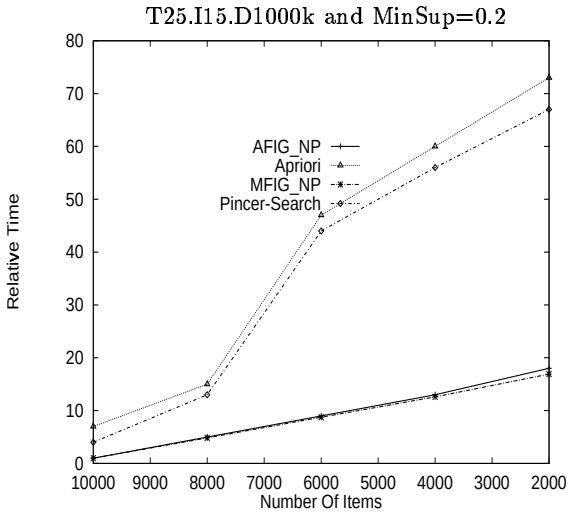
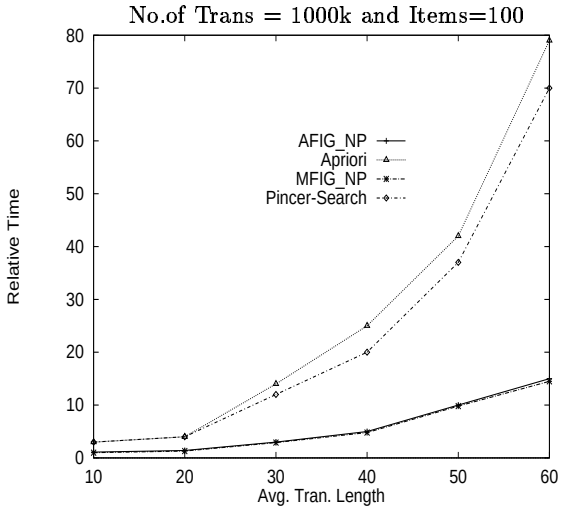


Figure 5: Scale-up Experiments : No. of Items



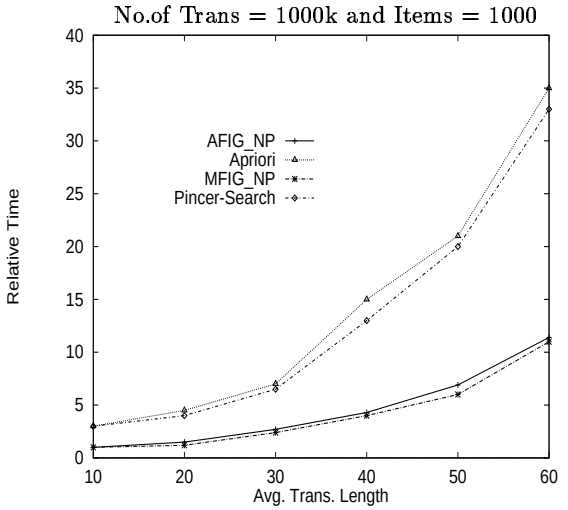
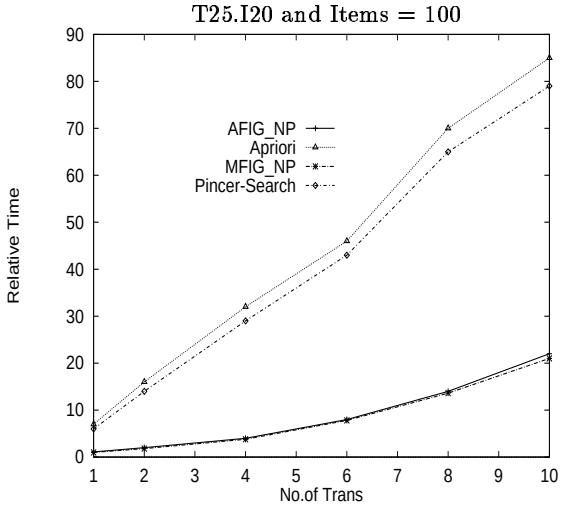


Figure 6: Scale-up Experiments : Average Length of Transactions. MinSup=0.2



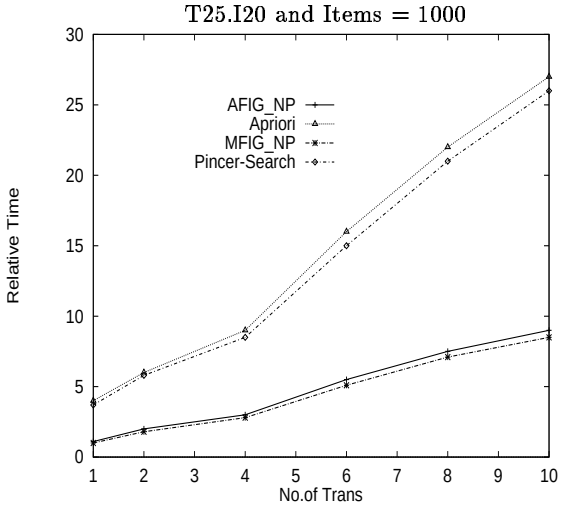


Figure 7: Scale-up Experiments : No. of Transactions (in Millions). MinSup=0.2

7. REFERENCES

[1] R.Agrawal, T.Imielinski and A.Swami. Mining Association Rules between Sets of Itemsets in Large Databases. *In proceedings of ACM SIGMOD* , Pages 207-216, May 1993.

[2] R.Agrawal and R.Srikanth. Fast Algorithms for Mining Association Rules. *In proceedings of Very Large Data Bases*, Pages 487 - 499, Sep. 1994.

[3] S.Brin, R.Motwani, J.D.Ullman, and S.Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. *In proceedings of ACM SIGMOD*, Pages 255 - 264, May 1997.

[4] J.Han, J.Pei and Y.Yin.Ullman Mining Frequent Patterns without Candidate Generation ket Basket. *In proceedings of ACM SIGMOD*, Pages 255 - 264, May 2000.

[5] D.I.Lin and Z.M.Kedem. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set, *In proceedings of Extending DataBase Technology*, Pages 105 - 119, March 1998.

[6] J.S.Park, M.S.Chen, P.S.Yu. An Effective Hash-based Algorithm for Mining Association Rules. *In proceedings of ACM SIGMOD*, Pages 175 - 186, May 1995.

[7] A.Savasere, E.Omiecinski and S.Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. *In proceedings of Very Large Data Bases*, Pages 432 - 444, Sep. 1995.

[8] M.J.Zaki, S.Parthasarathy, M.Ogihara and W.Li. New Algorithms for Fast Discovery of Association Rule Mining. *In proceedings of ACM SIGKDD,* Pages 283 - 286, Aug. 1997.

[9] M.J.Zaki, S.Parthasarathy, M.Ogihara and W.Li. New Algorithms for Fast Discovery of Association Rule Mining. *Technical Report 651, Computer Science Department, The University of Rochester, Rochster* , July 1997.