

# Efficient Mining of Maximal Sequential Patterns Using Multiple Samples \*

Congnan Luo<sup>†</sup>

Soon M. Chung<sup>‡</sup>

## Abstract

In this paper, we propose a new algorithm, named MSPX, which mines maximal sequential patterns by using multiple samples to effectively exclude infrequent candidates. MSPX begins with a bottom-up search. But at each pass, instead of processing all candidates, it always tries to find most of the infrequent ones effectively by counting only the potentially infrequent candidates against the whole database. After removing verified infrequent candidates, the remaining candidates are used to generate new candidates. Finally, with a top-down search, all the maximal frequent sequences can be identified efficiently. Sampling technique is used at each pass to distinguish potentially infrequent candidates. How to increase the minimum support level for the mining of samples to estimate if candidates could be infrequent is analyzed theoretically. Due to the supersequence frequency based pruning, MSPX reduces much more search space than other algorithms. Unlike the traditional single-sample methods proposed for mining frequent itemsets, MSPX uses multiple samples. Thus, it can avoid or alleviate some problems inherent in the single-sample methods. Our experiments show MSPX has very good performance and better scalability than other algorithms. Moreover, even though MSPX uses sampling, the variance of its performance is very small in multiple runs for the same task.

## 1 Introduction

Mining sequential patterns from large databases is an important problem in data mining. With numerous practical applications, such as consumer market-basket data analysis and web-log analysis, it has become an active research topic. Since it was introduced in [2], many algorithms have been proposed, but most of them are to discover the full set of frequent sequences.

In pure bottom-up, breadth-first search algorithms such as GSP [6] and PSP [5], only subsequence infre-

quency based pruning is used to reduce the number of candidate sequences. So, if a sequence with length  $l$  is frequent, all of its  $2^l$  subsequences must be enumerated first. Thus, if some frequent sequences are long, the overhead of enumerating all of their subsequences is so much that mining the full set of frequent sequences is impractical. An alternative approach is mining only the maximal frequent sequences. A frequent sequence is maximal if none of its supersequences is frequent. Mining only the maximal frequent sequences is efficient because the search space can be reduced a lot by using the supersequence frequency based pruning. In interactive data mining, after mining the set of maximal frequent sequences quickly, we can selectively count the interesting patterns subsumed by this set by scanning the database just once. Moreover, managing and querying a small set of maximal patterns is easy, time-saving and space-saving.

For the association rule mining, many efficient algorithms were proposed to mine maximal frequent itemsets [4]. However, differences between the two kinds of mining make those algorithms very difficult or impossible to be applied for the maximal frequent sequence mining. For example, given a set of items, the search space for mining frequent itemsets is limited, whereas it is unlimited for sequence mining. An item can appear at most once in an itemset but it may appear multiple times in a sequence at different positions.

A critical question for the maximal frequent sequence mining is how to look ahead for longer or maximal frequent sequences at a reasonable cost. In AprioriSome and DynamicSome algorithms [2], the candidates at some passes are directly used to generate longer candidates for the next pass. Actually, it leaves the job of excluding infrequent candidates to the later passes. Since the subsequence infrequency based pruning is not performed at all, a very large number of longer infrequent sequences are generated as candidates. The cost of identifying and removing these infrequent candidates can offset the gain from the supersequence frequency based pruning. On the other hand, in GSP, all the candidates at each pass are counted. In this way, we can exclude all the infrequent candidates at each pass, and hence avoid generating many false candidates. However, we cannot benefit from the supersequence infrequency

---

\*This research was supported in part by Ohio Board of Regents, LexisNexis, NCR, and AFRL/Wright Brothers Institute (WBI).

<sup>†</sup>Dept. of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435, USA.

<sup>‡</sup>Dept. of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435, USA.

based pruning. Based on these observations, a new MSPX algorithm is proposed in this paper, which emphasizes how to effectively exclude most infrequent candidates, so that the performance gain from the supersequence frequency based pruning can be maximized.

MSPX adopts the Apriori candidate generation method [1, 2, 6] and performs a bottom-up, breadth-first search first. But, instead of counting all the candidates on the whole database at each pass, it always tries to find and remove most of the infrequent candidates by counting as few candidates as possible. To achieve this, we count the candidates on a random sample drawn from the database at each pass to estimate which candidates are most potentially infrequent. Then, only the potentially infrequent candidates are verified against the whole database to remove really infrequent ones. The mining process is continued with the survived candidates to the next pass. At the end of the bottom-up phase, a superset of all frequent sequences is obtained. Starting from the border of this superset, a top-down search is performed to pick up maximal frequent sequences efficiently.

To improve the performance of MSPX, additional optimization methods are integrated: A signature technique is used to perform the subsequence infrequency based pruning when the seed set for the new candidate generation is too big to be loaded into memory. A prefix tree structure is developed to count the candidate sequences of different sizes during the database scanning, and it also facilitates the customer sequence trimming. MSPX outperforms GSP considerably, and also shows better scalability than SPAM [3] and SPADE [9]. By using multiple samples, the performance of MSPX is also much more stable than those of single-sample methods.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of sequence mining. Section 3 reviews some related works on sequence mining. Section 4 describes the MSPX algorithm. The experimental results and performance analysis are presented in Section 5. Section 6 contains some conclusions and future work.

## 2 Sequence Mining

Let  $\mathcal{I} = \{i_1, i_2, \dots, i_n\}$  be a set of items. An  $k$ -itemset  $i$  is a set of  $k$  items denoted by  $\{i_{m_1}, i_{m_2}, \dots, i_{m_k}\}$ , where  $1 \leq m_1 < m_2 < \dots < m_k \leq n$ . A sequence  $s$  is an ordered list of itemsets denoted by  $\langle s_1, s_2, \dots, s_k \rangle$ , where each  $s_i$ ,  $1 \leq i \leq k$ , is an itemset. A sequence  $s_a = \langle a_1, a_2, \dots, a_p \rangle$  is contained in another sequence  $s_b = \langle b_1, b_2, \dots, b_q \rangle$  if there exist integers  $1 \leq j_1 < j_2 < \dots < j_p \leq q$  such that  $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \dots, a_p \subseteq b_{j_p}$ . If  $s_a$  is contained in  $s_b$ ,  $s_a$  is a *subsequence* of  $s_b$ , and  $s_b$  is a *supersequence* of  $s_a$ . An item may

appear at most once in an itemset, but it may appear multiple times in different itemsets of a sequence. If there are  $k$  items in a sequence, the length of the sequence is  $k$ , and we call it a  $k$ -sequence. For example, a 3-sequence  $\langle \{A\}, \{B, C\} \rangle$  is a subsequence of a 5-sequence  $\langle \{C\}, \{A, D\}, \{B, C\} \rangle$ . For simplicity, these two sequences can be represented as  $A - BC$  and  $C - AD - BC$ .

Given a database  $\mathcal{D}$  of customer transactions, each transaction consists of a customer-id, transaction-time and an itemset which includes all the items purchased by the customer in that single transaction. All the transactions of a customer can be viewed as a customer sequence, where these transactions are ordered by their transaction times. We denote a customer sequence  $t$  as  $\langle T_1, T_2, \dots, T_m \rangle$ , which means the customer has  $m$  transactions in the database and each transaction  $T_i$ ,  $1 \leq i \leq m$ , contains all the items purchased in that transaction. A customer supports a sequence if the sequence is contained by the customer sequence. The support for a sequence in database  $\mathcal{D}$  is defined as the fraction of total customers who support the sequence. Given a user-specified minimum support, denoted by *minsup*, a sequence is frequent if its support is greater than or equal to *minsup*. The problem of sequence mining is to find all the frequent sequences in the database with respect to a user-specified *minsup*. If a sequence is frequent and none of its supersequences is frequent, then it is a maximal frequent sequence.

Based on the above definitions, two properties are often utilized to speed up the sequence mining: 1) Any supersequence of an infrequent sequence is not frequent, so it can be pruned from the set of candidates. This is called subsequence infrequency based pruning. 2) Any subsequence of a frequent sequence is also frequent, so it can be pruned from the set of candidates. This is called supersequence frequency based pruning.

In [6], the above definition of sequence mining was generalized by incorporating time constraints, sliding time windows, and taxonomy. This generalization makes the sequence mining more complex. For example, a sequence  $A - BC - D - GH$  is frequent does not necessarily mean that its subsequence  $A - BC - GH$  is also frequent, because the subsequence may not satisfy the time constraints. In this research, we consider the nongeneralized sequential pattern discovery.

## 3 Related Work

Mining sequential patterns was introduced in [2] with AprioriAll, AprioriSome and DynamicSome algorithms. Although AprioriSome and DynamicSome try to generate and count long candidate sequences before enumerating all their subsequences, their performance is usu-

ally worse than that of AprioriAll. The reason is that too many false candidates are generated without being pruned by the subsequence infrequency based pruning. The performance gain from the supersequence frequency based pruning is not enough to offset the cost of counting so many false candidates.

GSP [6] was proposed for generalized sequence mining, and it requires multiple passes on the database. At pass  $k$ , the set of candidate  $k$ -sequences are counted on the database and frequent  $k$ -sequences are determined. Then, the candidate  $(k + 1)$ -sequences are generated by joining frequent  $k$ -sequences for the next pass. This process will continue until no candidate is generated. Even though GSP is much faster than AprioriAll, it has a very high overhead of enumerating every single frequent subsequence when there are some long patterns. This is also the main weakness of other Apriori-like algorithms, such as PSP [5]. For PSP, a prefix tree was developed as the internal data structure to organize and count candidates more efficiently. The differences between the PSP's prefix tree and the one developed for our MSPX are: 1) our prefix tree is used to count candidates of different sizes, whereas PSP prefix tree is only used to count the candidates of the same size; 2) to improve the candidate counting, a bit vector is associated with our prefix tree to facilitate the customer sequence trimming; and 3) the supersequence frequency based pruning reduces the size of our prefix tree.

SPADE [9] works on the databases with a vertical id-list format, where a list of (customer-id, transaction-time) pairs are associated with each item, and the candidates are counted by intersecting the id-lists. A lattice-theoretic approach is used to decompose the search space into small pieces so that all working id-lists can be loaded into memory. SPAM [3] uses a vertical bitmap representation of the database for candidate generation and counting. A bitmap is created for each item in the database, where each bit corresponds to a transaction. If transaction  $j$  contains item  $i$ , then bit  $j$  in the bitmap for item  $i$  is set to 1; otherwise, it is set to 0. SPAM also uses a depth-first traversal of the Lexicographic sequence tree and an Apriori-based pruning of candidates.

Both SPADE and SPAM were reported more efficient than GSP. However, their performance may not be scalable in certain cases. For SPADE, if the database is in the horizontal format, where the transactions form the tuples in the database, transforming it to a vertical one requires extra disk space of roughly the same size. This may be a problem in practice if the database is large. Even if the database is in the vertical format, to efficiently count 2-sequences, SPADE proposes transforming it back to the horizontal one on the fly. This

usually requires much time and memory for very large databases and results in a performance degradation.

SPAM is claimed to be a memory-based algorithm. According to our tests, its scalability is much more sensitive to the number of items and the database size than other algorithms. The comparison between MSPX, GSP, SPADE and SPAM is presented in detail in the performance analysis section.

In [8], sampling was evaluated as an efficient way to mine an approximate set of frequent itemsets. In [7], a lowered minsup is used to mine the sample, so that the probability a frequent itemset is missed from the sample result would be small. Then, one more database scan is needed to find the misses and the overestimates.

Our proposed MSPX algorithm also uses sampling to mine maximal frequent sequences from databases. While most of other sampling-based mining algorithms use only one sample in the whole mining, MSPX uses multiple samples, one for each pass in its bottom-up phase.

#### 4 MSPX Algorithm

We present MSPX in this section. First, we use a simple example to explain the basic idea of MSPX. Figure 1 shows how GSP mines a small database with 4 distinct items  $A$ ,  $B$ ,  $C$  and  $D$ . The whole mining includes 4 passes, and each pass corresponds to a level in the lattice in Figure 1. At each pass, all candidate sequences are shown and the infrequent sequences identified are in gray color. In our description,  $C_k$  denotes the set of candidate  $k$ -sequences and  $L_k$  denotes the set of frequent  $k$ -sequences.

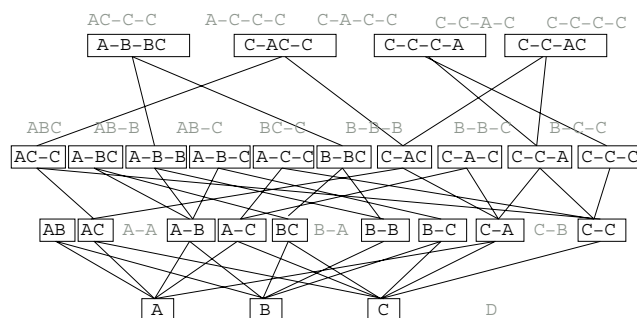


Figure 1: Candidate and Frequent Sequences in GSP

At pass 3 with  $C_3$ , there are three possible strategies to continue the mining: I) As in GSP, we count all 17 candidate 3-sequences, and then generate  $C_4$  from  $L_3$ . The subsequence infrequency based pruning is fully applied because all infrequent 3-sequences are known. II) As in AprioriSome and DynamicSome, we generate  $C_4$  directly from  $C_3$  without counting any candidate 3-sequence. Then, the candidate 4-sequences are counted

earlier than 3-sequences. The purpose of such look-ahead is to apply the supersequence frequency based pruning. If a 4-sequence is identified as frequent, then we can avoid counting its four subsequences of length 3. For the mining task requiring many passes, we can look ahead in this way during several passes or even all passes until the longest candidates are generated. But it has been shown that this method often produces many false candidates and even causes an explosion of the candidate set size [2]. III) Suppose that, with some a priori information, we can divide  $C_3$  into two disjoint subsets, such that the first subset contains most of the infrequent candidates in  $C_3$  while containing as few frequent ones as possible, and the second subset contains almost only the frequent candidates in  $C_3$ . For example, we may get the following two subsets:  $\{ABC, AB-B, B-B-B, AC-C, AB-C, BC-C, A-B-B\}$  and  $\{A-BC, A-B-C, A-C-C, B-BC, B-B-C, B-C-C, C-C-C, C-AC, C-A-C, C-C-A\}$ .

Then, we just count the candidates in the first subset on the whole database. Even though the second subset is not processed yet, we can expect that most of the infrequent candidates in  $C_3$  should have been identified because they are already clustered into the first subset. As a next step, the candidates already identified as frequent and those not counted yet are used to generate the candidates for the next pass. That means, the candidates in the second subset, which are not counted yet, are assumed to be frequent.

While all 17 candidate 3-sequences are counted in strategy I, only 7 candidate 3-sequences are counted in strategy III. In addition, there are 12 candidate 4-sequences generated in strategy III. This number is very close to 9 of strategy I, but much smaller than 25 of strategy II. As shown in this example, strategy III provides a much more effective way to exclude infrequent candidates at pass 3 and look ahead for longer patterns. At the same time, it doesn't impose much extra overhead for excluding longer false candidates in the later passes, because the subsequence infrequency based pruning still can be applied very effectively. Thus, the explosion of the candidate set size reported for strategy II can be avoided. Based on this idea, MSPX employs strategy III at each pass, and eventually obtains the set of potential maximal frequent sequences. Then, an efficient top-down search for real maximal frequent sequences can be performed, starting with this set.

Essentially, strategies I and II can be viewed as two extreme cases of strategy III, where the counted subsets are the whole candidate set and the empty set, respectively. However, both of them ignore the possibility of each candidate to be infrequent. Actually,

strategy III shows that such information can be used to improve the effectiveness of excluding infrequent candidates. A practical question regarding strategy III is how to divide  $C_k$  into such two disjoint subsets as desired. In our research, a sampling technique is developed as a part of the proposed MSPX algorithm.

Our description of MSPX is composed of the following parts: an overview of MSPX; a new signature technique used for subsequence infrequency based pruning; the issue of dividing  $C_k$  into two disjoint subsets; the prefix tree structure and the customer sequence trimming; and finally the comparison of our sampling method with previous ones.

**4.1 Overview of MSPX.** MSPX includes three phases:

**Initial Phase:**  $L_1$  and  $L_2$  are determined. Candidate 3-sequences are generated from  $L_2$ . To count candidate 2-sequences, a two-dimensional array is used. The entry at position  $(i, j)$  in the upper-triangle of the array contains the counts of three candidates  $i - j$ ,  $ij$  and  $j - i$ .

**Bottom-up Phase:** MSPX starts the bottom-up phase from pass 3. At pass  $k$  ( $k \geq 3$ ), a small random sample  $db$  is drawn from the database  $DB$ . We count all the candidates in  $C_k$  on  $db$  for their local supports (i.e., supports in  $db$ ). Then, we choose a support level  $\theta$  as the criterion to divide  $C_k$  into two subsets  $C_k^-$  and  $C_k^+$ . All the candidates with local supports lower than  $\theta$  are put into  $C_k^-$ , and others are put into  $C_k^+$ . The notation  $C_k^-$  ( $C_k^+$ ) means the candidates in  $C_k^-$  ( $C_k^+$ ) are negative (positive) to be frequent. We must try to make  $C_k^-$  contain almost all the infrequent candidates while containing as few frequent ones as possible. How to achieve this will be discussed later. After determining these two subsets, we just count the candidates in  $C_k^-$  on the rest of the database  $DB$ . The really infrequent candidates are removed. A set  $L_k^*$  is constructed by including all the candidates which are already verified frequent and those in  $C_k^+$ , which are not processed yet. Obviously, we have  $L_k \subseteq L_k^*$ . Then,  $L_k^*$  is used as the seed set to generate candidates for the next pass; i.e., the candidates in  $C_{k+1}$  are generated by joining the sequences in  $L_k^*$  as in GSP. At the end of this phase, we obtain a superset of all frequent sequences. The maximal sequences are extracted from this superset to construct  $MFS^*$ , the set of potential maximal frequent sequences. It is guaranteed that all maximal frequent sequences are under the border formed by  $MFS^*$ .

**Top-down Phase:** Starting with  $MFS^*$ , a top-down

search is performed. All the sequences in  $MFS^*$  are counted on  $DB$ . If a  $k$ -sequence ( $k > 3$ ) is infrequent, all of its  $(k - 1)$ -subsequences are considered as candidates for the next pass. For a frequent  $k$ -sequence, we stop splitting it, and put it into the set of maximal frequent sequences,  $MFS$ , if none of its supersequences is already in this set. If a subsequence of the frequent  $k$ -sequence is already in  $MFS$ , this subsequence should be removed from  $MFS$ . For a newly generated candidate  $(k - 1)$ -sequence, if it has any supersequence in  $MFS$ , we remove it from further consideration. We also check if the newly generated  $(k - 1)$ -sequence has any subsequence which is already identified as infrequent. If yes, this candidate must be split again. This top-down process continues until no new candidates are generated.

**4.2 Candidate Generation and Pruning in the Bottom-up Phase.** At pass  $k$  in the bottom-up phase, the candidates are generated in two steps:

**Join Step:** we generate candidate  $(k + 1)$ -sequences by joining  $L_k^*$  with  $L_k^*$ . For any two  $k$ -sequences  $s_1$  and  $s_2$  in  $L_k^*$ , if the subsequence obtained by dropping the first item of  $s_1$  is the same as the subsequence obtained by dropping the last item of  $s_2$ , a new candidate is generated by extending  $s_1$  with the last item of  $s_2$ . The added item starts a new itemset for  $s_1$  if it was a separate itemset in  $s_2$ . Otherwise, it becomes a member of the last itemset in  $s_1$ .

**Prune Step:** The candidate  $(k + 1)$ -sequences with any subsequence of length  $k$  which is not in  $L_k^*$  are removed. If  $L_k^*$  is too large to be loaded into memory totally, we perform the partial subsequence infrequency based pruning as described below.

A weakness of GSP is the way that a large  $L_k$  is processed. When minsup is very small,  $L_k$  could be too large to be loaded into memory totally. For this case, GSP proposed to use a relational merge-join technique to generate candidates. But in this manner, subsequence infrequency based pruning cannot be applied because the whole  $L_k$  is not available in memory and retrieving the relevant portions of  $L_k$  from a disk requires too many swaps. Without subsequence infrequency based pruning, usually the performance of GSP degrades a lot. In MSPX, we adopted a new method to solve this problem. If the  $L_k^*$  at some pass requires too much memory, we assign each  $k$ -sequence in  $L_k^*$  an integer signature which is highly correlated to the content of the sequence. A simple example of generating the signature is shown below, where  $t$  is the

number of itemsets in the sequence;  $m_i$  is the number of items in the  $i$ th itemset;  $I_{ij}$  is the  $j$ th item in the  $i$ th itemset;  $C_i$ ,  $1 \leq i \leq t$ , is the weight imposed on the  $i$ th itemset; and  $C_0$  is the weight imposed on the total number of itemsets.

$$(C_0 * t) + \sum_{i=1}^t (C_i * m_i * \sum_{j=1}^{m_i} I_{ij})$$

All the signatures are sorted and put into an array. Compared with the case of loading the whole  $L_k^*$  into memory, the signature array requires much less space. Thus, we can load working portions of  $L_k^*$  and all the signatures into memory at the same time. When a new candidate  $(k + 1)$ -sequence is generated, the signatures of its  $k$ -subsequences are computed and searched in the signature array. If any one of them is not in the array, the candidate should be removed. Since all the signatures are in memory, subsequence infrequency pruning still can be applied. It is possible that two or more  $k$ -subsequences have the same signature. However, that probability is very low. Our experiments showed that signatures are much more effective than hashing. MSPX performs much better than GSP when the seed set for the candidate generation cannot be loaded into memory totally at some passes. If the memory cannot hold all the candidates, we need to generate them in several parts. For each part, the candidates are counted on the sample. After all parts are processed, the candidates with the local support lower than  $\theta$  are loaded into memory to perform the counting on the rest of the whole database.

**4.3 Dividing the Candidate Set.** At each pass in the bottom-up phase, we need to divide  $C_k$  into two disjoint subsets,  $C_k^-$  and  $C_k^+$ , such that  $C_k^-$  contains most of the infrequent candidates while containing as few frequent ones as possible, and almost all the candidates in  $C_k^+$  are frequent. To achieve this, we collect the local support of each candidate in the sample  $db$ . We set a support level  $\theta$  as the criterion to estimate if a candidate could be frequent or not. If the local support of a candidate is lower than  $\theta$ , it is estimated to be infrequent and put into  $C_k^-$ . Otherwise, we expect it to be frequent and put it into  $C_k^+$ . If an infrequent candidate is misestimated as frequent, we call it an *overestimate*. On the other hand, if a frequent candidate is misestimated as infrequent, it is an *underestimate*.

Obviously, underestimates can affect only the computation overhead of the current pass. In the extreme case, even if we misestimate all frequent candidates by setting  $C_k^-$  as  $C_k$ , it just makes MSPX work like GSP at that pass. But it is hard to predict how much overestimates can affect the whole mining, because the misesti-

mated infrequent candidates in  $C_k^+$  will be directly used to generate new candidates. A certain number of overestimates may easily increase the number of false candidates and thus make the candidate set for the next pass much bigger than the case without those overestimates. This usually results in two consequences: First, the overhead of counting all candidates on the sample in the next pass increases. Second, but more importantly, with much more false candidates, the probability of making further overestimates tends to increase. Therefore, preventing overestimates is important for MSPX because they may make the case complicated and unpredictable.

The easiest way to choose  $\theta$  is setting it to the user-specified minsup. But we found out that, in practice, if there are many candidates whose supports are slightly lower than minsup, a lot of overestimates may occur. Furthermore, since the distribution characteristics of the database to be mined is usually unknown, we do not know if this will happen when the database is mined with respect to a specific minsup. Thus, it is necessary to take some measures to prevent the problem of having too many overestimates. In this research, we increase the user-specified minsup a little for  $\theta$  and explore the relationship between the increment of minsup and the probability of an overestimate. We believe this theoretical analysis can provide a guideline for us in running MSPX. We must keep in mind that if we increase the user-specified minsup too much, a lot of underestimates may happen, and it contradicts our purpose of using the sampling. On the other hand, if we increase minsup too little for  $\theta$ , we may have too many overestimates.

Consider an original database  $DB$  and an arbitrary sequence  $X$ . If the support of  $X$  in  $DB$  is  $P_X$ , then the probability that a customer sequence randomly selected from  $DB$  contains  $X$  is also  $P_X$ . Let's consider a random sample  $db$  with  $m$  customer sequences that are independently drawn from  $DB$  with replacement. The random variable  $T_X$ , which represents the total number of customer sequences containing  $X$  in  $db$ , has a binomial distribution of  $m$  trials with the probability of success  $P_X$ . In general, if  $m$  is greater than 30,  $T_X$  can be approximated by a normal distribution whose mean is  $m * P_X$  and the standard deviation is  $\sqrt{m * P_X * (1 - P_X)}$ .

In MSPX, suppose that we draw a sample  $db$  with  $m$  customer sequences from  $DB$ , and then try to use the point estimator  $P'_X = T_X/m$  to estimate the support of  $X$  in the population of  $DB$ . Then,  $P'_X$  is an unbiased estimator with mean  $m * P_X/m = P_X$  and standard deviation  $\sqrt{m * P_X * (1 - P_X)}/m = \sqrt{P_X * (1 - P_X)}/m$ .

If we assume the support of  $X$  in  $DB$ ,  $P_X$ , is

the user-specified minsup, then  $P'_X$ , which is observed from a sample  $db$ , should be around  $P_X$  with a normal distribution as described above. If we set  $\theta$  to a support level  $P''_X$ ,  $P''_X > P_X$ , the probability that the local support of  $X$  observed in the sample is not lower than  $P''_X$  is  $1 - P_Z$ , where  $Z = (P''_X - P_X)/\sqrt{P_X * (1 - P_X)}/m$ .  $Z$  is often called the  $z$ -score, and  $P_Z$  is the probability at the  $z$ -score value of  $Z$ .

Let's consider the standard deviation of  $P'_X$ ,  $\sqrt{P_X * (1 - P_X)}/m$ . The value of its part  $P_X * (1 - P_X) = -(P_X - 1/2)^2 + 1/4$  is increasing in the  $P_X$  interval of  $[0, 1/2]$ . Since minsup is usually lower than 50%, we can assume the value of  $P_X * (1 - P_X)$  is increasing in the  $P_X$  interval of  $[0, minsup]$ ; that means, the standard deviation of  $P'_X$  is increasing in this  $P_X$  interval. If the support of another sequence  $Y$  in  $DB$  is lower than the minsup  $P_X$  (i.e.,  $Y$  is actually an infrequent sequence), then both the mean and standard deviation of observed  $P'_Y$  should be smaller than those of  $P'_X$ , respectively. Therefore, compared with  $P'_X$ , the distribution curve of  $P'_Y$  is shifted left and shaper. Thus, the probability that the local support of an infrequent sequence  $Y$  observed in the sample is not lower than  $P''_X$  should be smaller than  $1 - P_Z$ . In other words, the probability that  $Y$  is overestimated is smaller than  $1 - P_Z$ .

For a specific mining job, if we specify the upper bound of the probability that an infrequent candidate is overestimated as  $\sigma$  (i.e.,  $1 - P_Z = \sigma$ ), then  $P_Z = 1 - \sigma$ . Let's denote the corresponding  $z$ -score for  $P_Z$  as  $Z_{(1-\sigma)}$ , which can be found from the  $z$ -score table. Then, we can compute  $P''_X$  using the formula (4.1). Actually,  $P''_X$  is the value of  $\theta$  for the upper bound  $\sigma$ .

$$(4.1) \quad P''_X = P_X + Z_{(1-\sigma)} * \sqrt{P_X(1 - P_X)}/m$$

Here,  $P_X = minsup$  and  $m = |db|$ . For example, if we set  $\sigma = 20\%$ , the corresponding  $Z_{(1-\sigma)}$  value is about 0.85. Our experiments show that the value of  $\theta$  (i.e.,  $P''_X$ ) computed using the formula (4.1) often tends to be very conservative. In the above analysis, we just considered a single sample and tried to control the occurrence of overestimates at a very low level for that pass. However, it is not an easy goal to achieve unless we increase minsup very much for  $\theta$ , which may cause too many underestimates.

Let's include the multiple samples into the analysis. In MSPX, if an infrequent sequence  $Y$  is overestimated at a pass, it will be used to generate new candidates. In the next pass, a new sample is drawn. If  $Y$  is not overestimated in the new sample, then all the false candidates grown from  $Y$  are not overestimated either, and hence they are removed. That means, the progressive overestimating based on  $Y$  can be avoided, and the negative effect caused by  $Y$  is limited within

these two passes. Based on this observation, we can relax our policy as follows: Overestimates are allowed at a reasonable level at the current pass, but they are strictly prevented from happening again at the next pass. With this policy, two consecutive samples are used for the analysis. Obviously, if the two samples are drawn independently, the probability that an infrequent candidate is overestimated in both samples cannot be bigger than  $(1 - P_Z)^2$ . Thus, with the same requirement  $\sigma$  on the upper bound of the probability of an overestimate,  $P_Z$  is changed from  $1 - \sigma$  to  $1 - \sqrt{\sigma}$ . Thus, we can rewrite the formula (4.1) as

$$\theta = \text{minsup} + Z_{(1-\sqrt{\sigma})} * \sqrt{\text{minsup} * (1 - \text{minsup}) / |db|} \quad (4.2)$$

Here,  $\sigma = 20\%$  means that the probability an infrequent candidate is overestimated in two consecutive samples is at most 20%. Actually, the upper bound of the probability of an overestimate in the first sample is relaxed to 45%, not the original 20%. The corresponding  $z$ -score  $Z_{(1-\sqrt{\sigma})}$  is about 0.13. Our tests show that the formula (4.2) provides a tighter  $\theta$  value, and MSPX works better with it in practice.

#### 4.4 Efficient Counting of Candidates Using the Prefix Tree and the Customer Sequence Trimming.

During the top-down search for maximal patterns covered by  $MFS^*$ , to reduce the number of passes, we need to count candidates of different sizes at each pass over the database. For that purpose, we developed a new prefix tree structure. Since it is much more efficient than the hash tree, we also use it to count the candidates of the same size during the bottom-up phase.

The following example shows how the prefix tree works. Suppose we have 10 candidates of length 2 or 3. The prefix tree is constructed as shown in Figure 2. Each node is associated with a pointer. If the path from the root to a node represents a candidate, the pointer points to the candidate; otherwise, it is NULL. A node may have two types of children. The ‘‘I-extension’’ child means the item represented by the child node is in the same itemset with the item represented by its parent node. The ‘‘S-extension’’ child means the item represented by the child node starts a new itemset. All the S-extension (I-extension) children of a node are linked together, and only the first child is linked to their parent node by a dashed (solid) line. For example, nodes 4 and 5 are the S-extension children of node 1, and the corresponding paths represent the candidates  $A - A$  and  $A - E$ , respectively. Nodes 6 and 7 are I-extension children, and their paths represent  $AC$  and  $AD$ , respectively.

To speed up the counting, a bit vector is associated

with the prefix tree to facilitate the customer sequence trimming. In this example, we have 8 items in the database:  $A, B, C, D, E, F$ , and  $H$ . Since  $B, F$ , and  $G$  do not appear in any candidate, they should be ignored during counting. Thus, the bit vector is set as  $(10111001)$ , where 1 at the  $i$ -th bit position means item  $i$  appears in the prefix tree. All the bits are initialized to 0, and the corresponding bits are set to 1 as we insert candidates into the prefix tree.

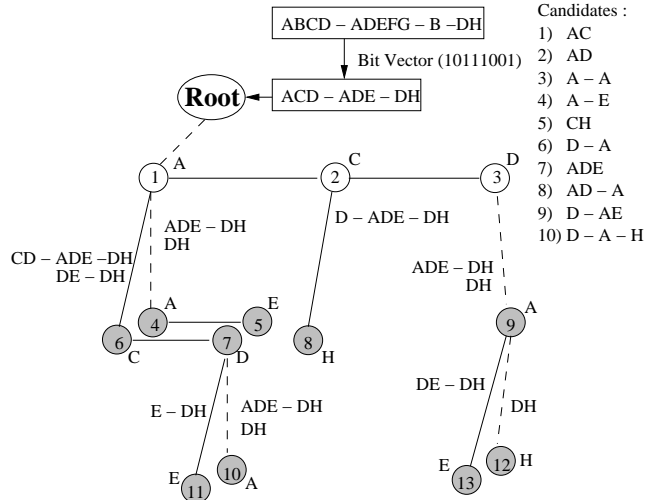


Figure 2: Prefix Tree of MSPX

Given a customer sequence  $s = ABCD - ADEFG - B - DH$ , we trim it to  $s' = ACD - ADE - DH$  using the bit vector first. Then, a recursive method is used to count all the candidates contained in  $s'$ . At the root node, we check each item in  $ACD - ADE - DH$  to see if it is in the root node's S-extension children. The first item of  $s'$  is  $A$ , and it appears as the first S-extension child of the root node. So we recursively call the count function at the root node with two sequence segments. The segment  $CD - ADE - DH$  is used in the call for node 1's I-extension link, while  $ADE - DH$  is for its S-extension link. Then, we can locate the second item of  $s'$ ,  $C$ , at node 2. Since node 2 has no S-extension child, only one recursive call with the segment  $D - ADE - DH$  is made for its I-extension link. The third item of  $s'$ ,  $D$ , is the last item of the first itemset in  $s'$ . Only one call with segment  $ADE - DH$  is made for node 3's S-extension link. The fourth item of  $s'$ ,  $A$ , can be located at node 1 again, and we make two recursive calls. One is for the node 1's I-extension link with  $DE - DH$ , and the other one is for its S-extension link with  $DH$ . Then, we process the remaining items in  $s'$ , one by one, in the same way. Whenever we locate an item at some node, if the pointer associated with the node is not NULL and the count of the corresponding candidate is not

increased yet (for the current customer sequence), it should be increased.

The root node is processed differently from other nodes. At the root node, there is no constraint on which items in the customer sequence should be checked against the root's S-extension link, because the first item of a candidate can appear anywhere in the customer sequence. At other nodes, there are always some constraints. Let's see how to make recursive calls at node 1 along its I-extension link. Recall that we have made two recursive calls at the root node with segments,  $CD - ADE - DH$  and  $DE - DH$ , for node 1's I-extension. Now we process them at node 1. Since the two segments are specified for node 1's I-extension link, we should check the items in their first itemsets,  $CD$  and  $DE$ , against node 1's I-extension link. For  $CD - ADE - DH$ , since  $C$  appears at node 6 which has no child, we stop there by just increasing the count of  $AC$ . Another item,  $D$ , appears at node 7. We increase the count of  $AD$  and make recursive calls for node 7's links. Since  $D$  is the last item of the first itemset in  $CD - ADE - DH$ , only one recursive call with the segment  $ADE - DH$  is made for node 7's S-extension link. For another sequence segment  $DE - DH$  at node 1, two items of the first itemset,  $D$  and  $E$ , are checked.  $D$  is located at node 7. Since the count of  $AD$  is already increased before, we should not increase it again. Two recursive calls are made at node 1 for node 7's links. One is with  $E - DH$  for node 7's I-extension link and the other is with  $DH$  for the S-extension link. We can ignore  $E$  because it is not an I-extension child of node 1. This process will continue until a leaf node is reached or the sequence segment is empty.

**4.5 Sampling in MSPX.** The way of using the sampling in MSPX is unique compared to other previous researches [7, 8]. We use multiple samples in MSPX, instead of using a single one. However, we just collect the local supports of candidates of the same size in the sample, rather than mining the sample completely using a fixed minsup. There are some problems in the previous studies [7, 8]: 1) As only one random sample is used, if the sample does not represent the database well, it will affect the whole mining and degrade the overall performance very much. 2) It is not a surprise that the performance of the algorithms using a single sample may vary considerably from one run to another for the same mining task. 3) When mining a sample, a fixed minsup, either the user-specified minsup or a lowered one, is used. For those candidate sequences whose global supports are slightly higher or lower than the minsup, misjudgement happens frequently because their local supports in the sample often diverge from

their global supports. Then, a lot of effort is needed to identify the overestimates and the underestimates in the sample results. 4) For single-sample methods, when the minsup is very small, simply using this minsup or a lowered one to mine the sample is risky, because  $minsup * |db|$  or  $lowered\_minsup * |db|$  used to filter the candidates during the mining of the sample could be too low. In that case, many overestimates may happen. They not only make the mining of the sample itself very difficult, but also pose a heavy overhead on verifying the sample results. We found this problem is more serious for sequence mining than for mining the frequent itemsets, because the search space of sequence mining is much bigger. Thus, the sampling method used in [7, 8] is challenged when minsup is very small. Using a large sample can relieve these problems to some extent, but it cuts the merit of sampling.

In MSPX, these problems are avoided or alleviated by using multiple samples, one for each pass in the bottom-up phase. If a sample is bad, we may need to count many underestimated candidates in  $C_k^-$ . However, it just affects the current pass. We may also overestimate many infrequent candidates and put them into  $C_k^+$ . Then, they may generate many false candidates and affect the next pass. Fortunately, in the next pass, such false candidates will be categorized into  $C_k^-$  or  $C_k^+$  again based on their local supports in a new sample. Thus, we still have the opportunity to stop the candidate growth caused by the overestimates made in the previous passes. Actually, the probability of choosing bad samples in a row is extremely small if the sample size is reasonable. Due to the joint contribution from multiple samples, the negative effect from one bad sample can be limited within a couple of passes, rather than the whole mining. Our experiments showed that the variance of the execution time of MSPX during 100 runs for each test is very small. The worst case of MSPX was also much better than that of single-sample methods.

In MSPX, we do not set a fixed minsup for the mining of samples. Instead, we simply collect the local supports of the candidates in the sample. Even though we use the local supports to categorize candidates into  $C_K^-$  or  $C_K^+$ , whether to remove a candidate from the search space or not is still based on its global support after it is verified against the whole database. Thus, no frequent sequence will be missed in the bottom-up phase. This enables us to perform an efficient top-down search for all maximal patterns. Moreover, to avoid having too many overestimates, we increase the minsup a little for mining the sample. Thus, we relieve the problem that too many infrequent sequences are overestimated as frequent and placed into  $C_k^+$ .

Initially, we concerned if MSPX would incur a lot of

overhead due to multiple samples when it is compared to the case of using a single sample. In single-sample methods, we draw one single sample but mine it in multiple passes. On the other hand, MSPX draws multiple samples, but each sample is processed in a single pass. Roughly speaking, in both methods, the sampling procedure includes three steps: 1) randomly selecting distinct customer ids for the sample, 2) loading the sample into memory, and 3) processing the sample.

The time required for step 1 is negligible. For step 2, in the traditional single-sample methods, we just need to load the sample into memory once if we have enough memory space. In MSPX, we must do it multiple times because the samples are different at different passes. However, due to the small size of the sample, the loading time is not a dominant factor. Step 3 is the dominant part in the overhead related to the sampling. Even though MSPX uses multiple samples, it does not mine each sample completely. For each sample, it just counts the candidates of the same length for that pass (i.e., candidate  $k$ -sequences for the  $k$ th pass). The single-sample methods mine the sample completely in multiple passes with respect to a minsup.

Let's simplify the situation by assuming that any sample drawn can represent the database well. In the single-sample methods, if GSP is used to mine the sample, the number of passes and the candidate set size at each pass will be similar to those of running GSP on the whole database. In the bottom-up phase of MSPX, since most infrequent candidates can be excluded at each pass as discussed before, the subsequence infrequency based pruning can be performed as effectively as in GSP. Thus, in this phase, the number of passes and the candidate set size at each pass are also similar to the case of running GSP on the whole database. Therefore, the computation costs for step 3 in both single-sample methods and our multi-sample method are actually very close to each other.

The overhead for sampling in both types of methods is actually determined by steps 2 and 3. Our tests showed that MSPX incurs a little more overhead than single-sample methods, but not much. Overall, MSPX still demonstrates its advantage in terms of the average performance, the worst-case performance and the stability in performance. Some relevant test results will be shown in the following performance analysis section.

## 5 Performance Analysis

To compare MSPX with other algorithms, we implemented GSP and obtained the source codes of SPAM and SPADE from their authors' web sites. In addition, we are also interested in the comparison between multi-sample method MSPX and the traditional single-

sample methods. Thus, we implemented a variant of GSP, which will be called GSP-Samp in this paper, by integrating the sampling technique in a traditional way: using GSP to mine a random sample with respect to minsup first, validating sample results to remove false patterns, then performing GSP on the database and using the longest frequent sequences found in the sample to prune candidates at each pass.

All the experiments were performed on a SuSE Linux PC with a 2.6 GHz Pentium processor and 1 Gbytes main memory. For MSPX and GSP-Samp, since sampling technique is probabilistic, we repeated each test 100 times. The average execution time of the 100 runs was reported as the performance result. The default sample size for MSPX and GSP-Samp was fixed as 10% of the test database for all experiments. For MSPX, the support level  $\theta$  for the sample is computed using the formula (4.2). The upper bound of the probability that an infrequent candidate is overestimated at two consecutive passes is set as 20%, i.e.  $\sigma = 0.2$ . Thus, we have  $Z_{(1-\sqrt{\sigma})} = Z_{0.55} = 0.13$ . The databases used in our experiments are synthetically generated as in [2]. The database generation parameters are described in Table 1. For all databases,  $N_S = 5000$  and  $N_I = 25,000$ ; and the names of the databases reflect other parameter values used to generate them.

Table 1: Parameters Used in Database Generation

$D$	Number of customers in the database
$C$	Average number of transactions per customer
$T$	Average number of items per transaction
$S$	Average length of maximal potentially frequent sequences
$I$	Average length of maximal potentially frequent itemsets
$N$	Number of distinct items in the database
$N_S$	Number of maximal potentially frequent sequences
$N_I$	Number of maximal potentially frequent itemsets

**5.1 Performance Comparison.** We ran MSPX, GSP, SPADE and SPAM on databases with medium sizes of about 100 Mbytes. The number of items in these databases is 10,000. The test results on D400K-C10-T5-S10-I2.5-N10K database are presented in Figure 3. In our tests, SPAM could not mine these databases, and its run was terminated by the operating system. Our machine is a 32-bit system, but the user address space is limited to 2 Gbytes. In all these tests, SPAM always required more than 2 Gbytes memory, and hence caused the termination.

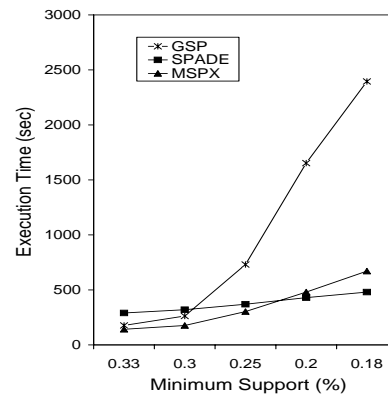
With the optimization components integrated, MSPX performs much better than GSP because it processes fewer candidates in a much more efficient way.

When the minsup is decreased, more and more candidates appear during the mining. In that case, the overhead of GSP in candidate generation, pruning, and especially counting using a huge hash tree increases drastically. For MSPX, this situation is considerably improved by using the supersequence frequency based pruning, the prefix tree structure, and the customer sequence trimming. At each pass in the bottom-up phase of MSPX, only a part of the candidates are selected to be counted on the whole database. As most infrequent sequences were identified early, the situation that too many false candidates are generated did not happen in all the tests. In the top-down phase of MSPX, the search starts with the potential maximal frequent sequences. Once a maximal frequent sequence is found, all of its subsequences are removed from the search space. Thus, the total number of candidates being counted on the whole database is much smaller than that of GSP. Figure 3(b) shows how many candidates with length greater than 2 have been counted on the whole databases in GSP and MSPX.

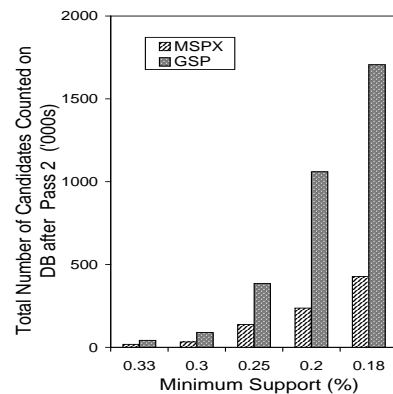
In SPADE, the subsequence infrequency based pruning is performed only partially. Compared with GSP, SPADE is expected to process more candidates. The main advantage of SPADE is the efficient counting of the candidates by intersecting the id-lists. An inefficient part of SPADE is the counting of  $C_2$  for medium and large databases in the vertical format, which degrades the whole performance of SPADE very much. Considering both factors, we can say that if there are not enough number of candidates of length greater than 2 to be counted, SPADE cannot show its efficiency. That is why SPADE is even worse than GSP when minsup is big as shown in Figure 3(a). In these tests, MSPX performed best for large and medium minsup. Only when the minsup is very small, SPADE performed best.

**5.2 Scalability Evaluation.** Both SPADE and SPAM need to store a huge amount of intermediate data to save their computation cost. When the memory space requirement is over the memory size available, CPU utilization drops quickly due to the frequent swapping. Compared with them, MSPX and GSP process the customer sequences one by one, hence only a small memory space is needed to buffer the customer sequences being processed. MSPX can also handle the situation that  $L_k^*$  or  $C_k$  cannot be totally loaded into memory by using the signatures as explained in Section 4. Therefore, MSPX does not require the memory space as much as GSP, SPADE and SPAM.

Many real-life customer market-basket databases have tens of thousands of items and millions of customers, so we evaluated the scalability of the mining al-



(a) Performance



(b) Search Space (MSPX vs. GSP)

Figure 3: Tests on D400K-C10-T5-S10-I2.5-N10K

gorithms in these two aspects. First, we started with a very small database D1K-C10-T5-S10-I2.5 and changed the number of items from 500 to 10,000. The user-specified minsup was 0.5%. To run MSPX on such a small database with only 1000 customers, we selected the whole database as the sample and set  $\theta$  to minsup. Since MSPX does not apply the sampling on such a small database, supersequence frequency based pruning is not performed in mining. Thus, in this case, SPADE and SPAM performed better than MSPX and GSP as long as their memory requirement is satisfied.

As the number of items is increased, SPAM shows its scalability problem. Theoretically, the memory space required to store the whole database into bitmaps in SPAM is  $D * C * N / 8$  bytes. For the id-lists in SPADE, it is about  $D * C * T * 4$  bytes. But we found these values are usually far less than their peak memory space requirement during the mining, because the amount of intermediate data in both algorithms is quite large.

As shown in Figure 4, even though the D1K-C10-T5-S10-I2.5-N8000 database takes only 260 Kbytes, and the theoretical memory space requirement to store the database in SPAM is about  $1000 * 10 * 8000/8$  bytes  $\approx 10$  Mbytes, it could not finish the mining when the minsup was 0.5%, because it required more than 2 Gbytes of memory. Compared with SPAM, SPADE divides search space into small pieces so that only the id-lists being processed need to be loaded into memory. Another advantage of SPADE is that the id-lists become shorter and shorter with the progress in mining, whereas the length of the bitmaps does not change in SPAM. These two differences make SPADE much more space-efficient than SPAM. We also fixed the parameter  $N$  as 1000 and changed the database size from 1K to 100K customer sequences. SPAM could not mine the databases with more than 20K customers due to its memory requirement problem. Our tests showed that SPAM is very sensitive to the number of items and the number of customers, which mainly limits its applicability.

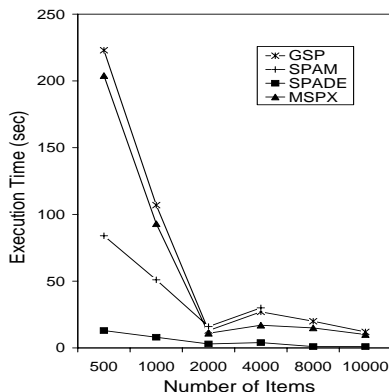


Figure 4: Scalability: Number of Items (on D1K-C10-T5-S10-I2.5, minsup=0.5%)

Second, we investigated how they perform on C10-T5-S10-I2.5-N10K when the user-specified minsup minsup is 0.18%. We fixed the number of items as 10,000 and increased the number of customers from 400,000 to 2,000,000. SPAM could not perform the mining due to its memory requirement problem. For SPADE, we partitioned the test database into multiple chunks for better performance when its size was increased. Otherwise, the counting of  $C_2$  for a large database could be extremely time-consuming. We made each chunk contain 400,000 customers so that its size is only about 100 Mbytes, which is one tenth of our main memory size. Thus, D400K-C10-T5-S10-I2.5-N10K is processed as one chunk, D800K-C10-T5-S10-I2.5-N10K is divided into two chunks, and so on. Figure 5 shows that the scalability of MSPX and GSP are quite linear. But

SPADE cannot maintain a reasonable scalability as the database becomes larger. As the database size is increased, MSPX performs much better than the others.

When the database was relatively small with only 400,000 customers, SPADE performed best — about 20% faster than MSPX. But when the database size is increased from 1600K customers to 2000K customers, there is a sharp performance drop in SPADE, such that it is even slower than GSP. In that case, MSPX is faster than SPADE by a factor of about 8. As discussed before, counting  $C_2$  is a performance bottleneck for SPADE, because the transformation of a large database from the vertical format to the horizontal format takes too much time. When the database is very large, the transformation also requires a large amount of memory and frequent swapping, hence the performance drops drastically. Partitioning the database can relieve this problem to some extent but does not solve it completely. Moreover, for the database with a large number of items and customers, SPADE needs more time to intersect more and longer id-lists.

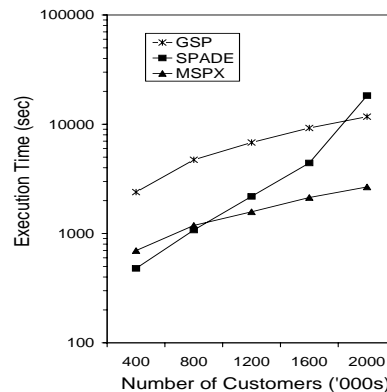


Figure 5: Scalability: Number of Customers (on C10-T5-S10-I2.5-N10K, minsup=0.18%)

Finally, we mined a large database D2000K-C10-T5-S10-I2.5-N10K, which takes about 500 Mbytes, for various minsup. This database was partitioned into 5 chunks for SPADE, and the results are shown in Figure 6. Based on our tests, we found SPADE performs best for small size databases. For medium size databases, MSPX performs better for relatively big minsup while SPADE is faster for small minsup. When the database is large, SPADE's performance drops drastically, and MSPX outperforms SPADE very much.

**5.3 Multi-Sample MSPX vs. Single-Sample GSP-Samp.** As far as we know, all the previous researches on the association rule mining based on the sampling used a single random sample or refined a

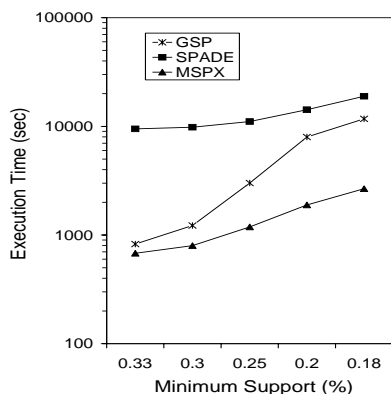


Figure 6: Performance on a Large Database D2000K-C10-T5-S10-I2.5-N10K

single big sample to a smaller one. In MSPX, multiple samples are used. We compared MSPX and GSP-Samp to see if multiple samples can avoid or alleviate the problems inherent in the single-sample methods discussed earlier. To exclude other factors affecting the performance of GSP-Samp, the signature based subsequence infrequency pruning, the prefix tree and the customer sequence trimming techniques were also used for the implementation of GSP-Samp.

Compared with GSP-Samp, MSPX has better average performance. Most importantly, the performance variance of MSPX is much smaller than that of GSP-Samp. The worst performance of MSPX also indicates that even if a few bad samples had been drawn, MSPX could successfully suppress their negative effect. Otherwise, the worst case of MSPX could have been much worse than what we observed, probably similar to the worst case of GSP-Samp. This proves that MSPX is not sensitive to a couple of bad samples because of the contribution of multiple samples.

## 6 Conclusions and Future Work

In this paper, we proposed an algorithm named MSPX, which mines maximal frequent sequences by effectively excluding infrequent candidates. Multiple samples are used in MSPX to avoid or alleviate some problems inherent in the algorithms using only one sample. For MSPX, we explored the relationship between the increment of the user-specified minsup for the sample and the probability of an overestimate. A theoretical guideline is given to increase the minsup for the sample in the context of multiple samples. Our extensive experiments proved that MSPX is a practical and efficient algorithm. Its excellent scalability makes it a very good candidate for mining customer market-basket databases which usually have tens of thousands of items and millions of customer sequences. More importantly, even

though MSPX is a sampling-based algorithm, the variance of its performance during multiple runs for the same mining task is usually very small. Applying the proposed idea of effectively excluding infrequent candidates and the multiple sampling technique to other sequence mining algorithms will be an interesting project.

## References

- [1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th VLDB Conf.*, 1994, pp. 487–499.
- [2] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. of Int'l Conf. on Data Engineering*, 1995, pp. 3–14.
- [3] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential Pattern Mining Using a Bitmap Representation," *Proc. of ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, 2002, pp. 429–435.
- [4] S. M. Chung and C. Luo, "Distributed Mining of Maximal Frequent Itemsets from Databases on a Cluster of Workstations," *Proc. of the 4th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid – CCGrid 2004*, IEEE Computer Society Press, 2004.
- [5] F. Massegli, F. Cathala, and P. Poncelet, "The PSP Approach for Mining Sequential Patterns," *Proc. of European Symp. on Principle of Data Mining and Knowledge Discovery*, 1998, pp. 176–184.
- [6] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proc. of the 5th Int'l Conf. on Extending Database Technology*, 1996, pp. 3–17.
- [7] H. Toivonen, "Sampling Large Databases for Association Rules," *Proc. of the 22nd VLDB Conf.*, 1996, pp. 134–145.
- [8] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara, "Evaluation of Sampling for Data Mining of Association Rules," *Proc. of the 7th Int'l Workshop on Research Issues in Data Engineering*, 1997.
- [9] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning*, 42(1), 2001, pp. 31–60.