Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Systems 29 (2004) 385–404

www.elsevier.com/locate/infosys

Incremental update on sequential patterns in large databases by implicit merging and efficient counting ☆

Ming-Yen Lin, Suh-Yin Lee*

Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan 30050, China

Received 6 February 2002; received in revised form 20 December 2002; accepted 2 April 2003

Abstract

Current approaches for sequential pattern mining usually assume that the mining is performed in a static sequence database. However, databases are not static due to update so that the discovered patterns might become invalid and new patterns could be created. In addition to higher complexity, the maintenance of sequential patterns is more challenging than that of association rules owing to sequence merging. Sequence merging, which is unique in sequence databases, requires the appended new sequences to be merged with the existing ones if their customer ids are the same. Re-mining of the whole database appears to be inevitable since the information collected in previous discovery will be corrupted by sequence merging. Instead of re-mining, the proposed *IncSP* (*Inc*remental *S*equential *P*attern Update) algorithm solves the maintenance problem through effective implicit merging and efficient separate counting over appended sequences. Patterns found previously are incrementally updated rather than re-mined from scratch. Moreover, the technique of early candidate pruning further speeds up the discovery of new patterns. Empirical evaluation using comprehensive synthetic data shows that *IncSP* is fast and scalable.

Keywords: Data mining; Sequential patterns; Incremental update; Sequence discovery; Sequence merging

1. Introduction

Sequential pattern discovery, which finds frequent temporal patterns in databases, is an important issue in data mining originated from retailing databases with broad applications [1–8]. The discovery problem is difficult considering the numerous combinations of potential sequences, not to mention the re-mining required when databases are updated or changed. Therefore, it is essential to investigate efficient algorithms for sequential pattern mining and effective approaches for sequential pattern updating.

A sequential pattern is a relatively frequent sequence of transactions, where each transaction is a set of items (called *itemset*). For example, one might purchase a PC and then purchase a printer later. After some time, he or she could possibly buy some printing software and a scanner. If there exists a sufficient number of customers in the transactional database who have the purchasing sequence of PC, printer, printing software and scanner, then such a frequent sequence is a sequential pattern. In general, each customer

☆ Recommended by Prof. Nick Koudas.

*Corresponding author.
E-mail address: sylee@csie.nctu.edu.tw (S.-Y. Lee).

0306-4379/03/$ - see front matter
doi:10.1016/S0306-4379(03)00036-X

record in the transactional database is an itemset associated with the transaction *time* and a *customer-id* [1]. Records having the same customer-id are sorted by ascending transaction time into a *data sequence* before mining. The objective of the discovery is to find out all sequential patterns from these data sequences.

A *sequential pattern* is a sequence having *support* greater than or equal to a minimum threshold, called the *minimum support*. The *support* of a sequence is the percentage of data sequences containing the sequence. Note that the support calculation is different in the mining of association rules [9–12] and sequential patterns [1,4,7,13]. The former is transaction-based, while the latter is sequence-based. Suppose that a customer has two transactions buying the same item. In association discovery, the customer "contributes" to the support count of that item by two, whereas it counts only once in the support counting in sequential pattern mining.

The discovery of sequential patterns is more difficult than association discovery because the patterns are formed not only by combinations of items but also by permutations of itemsets. For example, given 50 possible items in a sequence database, the number of potential patterns is $50 \times 50 + C(50,2)$ regarding two items, and $50 \times 50 \times 50 + 50 \times C(50,2) \times 2 + C(50,3)$ regarding three items (formed by 1-1-1, 1-2, 2-1, and 3),…, etc. Most current approaches assume that the sequence database is static and focus on speeding up the time-consuming mining process. In practice, databases are not static and are usually appended with new data sequences, conducted by either existing or new customers. The appending might invalidate some existing patterns whose supports become insufficient with respect to the currently updated database, or might create some new patterns due to the increased supports. Hence, we need an effective approach for keeping patterns up-to-dated.

However, not much work has been done on the maintenance of sequential patterns in large databases. Many algorithms deal with the mining of association rules [9,10,12], the mining of sequential patterns [1,3,7,8,14,15], and parallel mining of sequential patterns [6]. Some algorithms discover frequent episodes in a single long sequence [16]. Nevertheless, when there are changes in the database, all these approaches have to re-mine the whole updated database. The re-mining demands more time than the previous mining process since the appending increases the size of the database.

Although there are some incremental techniques for updating association rules [11,17], few research has been done on the updating of sequential patterns, which is quite different. Association discovery is transaction-based; thus, none of the new transactions appended is related to the old transactions in the original database. Sequential pattern mining is sequence-based; thus, the two data sequences, one in the newly appended database and the other in the original database, must be merged into a data sequence if their customer-ids are the same. However, the *sequence merging* will corrupt previous support count information so that either FUP or FUP2 [17] algorithm could not be directly extended for the maintenance of sequential patterns.

One work dealing with incremental sequence mining for vertical database is the *I*ncremental *S*equence *M*ining (*ISM*) algorithm [5]. Sequence databases of vertical layout comprise a list of (*cid*, *timestamp*) pairs for each of all the items. In order to update the supports and enumerate frequent sequences, ISM maintains "maximally frequent sequences" and "minimally infrequent sequences" (called *negative border*). However, the problem with ISM is that the size of negative border (i.e. the number of potentially frequent sequences) might be too large to be processed in memory. Besides, the size of extra space for transformed vertical lists might be several times the size of the original sequence database.

This paper presents an efficient incremental updating algorithm for up-to-date maintenance of sequential patterns after a nontrivial number of data sequences are appended to the sequence database. Assume that the minimum support keeps the same. Instead of re-mining the whole database for pattern discovery, the proposed algorithm utilizes the knowledge of previously computed frequent sequences. We merge data sequences implicitly, generate fewer but more promising candidates, and separately count supports with respect to the original database and the

newly appended database. The supports of old patterns are updated by merging new data sequences implicitly into the original database. Since the data sequences of old customers are processed already, efficient counting over the data sequences of new customers further optimizes the pattern updating process.

The rest of the paper is organized as follows. Section 2 describes the problem of sequential pattern mining and addresses the issue of incremental update. In Section 3, we review some previous algorithms of sequence mining. Section 4 presents our proposed approach for the updating of sequential patterns after databases are changed. Comparative results of the experiments by comprehensive synthetic data sets are depicted in Section 5. Section 6 concludes this paper.

2. Problem formulation

In Section 2.1, we formally describe the problem of sequential pattern mining and the terminology used in this paper. The issue of incremental update is presented in Section 2.2. Section 2.3 demonstrates the changes of sequential patterns due to database update.

2.1. Sequential pattern mining

A *sequence* s, denoted by $\langle e_1 e_2 \ldots e_n \rangle$, is an ordered set of n elements where each *element* e_i is an itemset. An itemset, denoted by $(x_1, x_2, \ldots, x_q)$, is a nonempty set of q items, where each *item* x_j is represented by a literal. Without loss of generality, we assume the items in an element are in lexicographic order. The *size* of sequence s, written as $|s|$, is the total number of items in all the elements in s. Sequence s is a *k-sequence* if $|s| = k$. For example, $\langle (8)(2)(1) \rangle$, $\langle (1,2)(1) \rangle$, and $\langle (3)(5,9) \rangle$ are all 3-sequences. A sequence $s = \langle e_1 e_2 \ldots e_n \rangle$ is a *subsequence* of another sequence $s' = \langle e'_1 e'_2 \ldots e'_m \rangle$ if there exist $1 \leqslant i_1 < i_2 < \ldots < i_n \leqslant m$ such that $e_1 \subseteq e'_{i1}, e_2 \subseteq e'_{i2}, \cdots$, and $e_n \subseteq e'_{in}$. Sequence s' *contains* sequence s if s is a subsequence of s'. For example, $\langle (2)(1,5) \rangle$ is a subsequence of $\langle (\mathbf{2},4)(3)(\mathbf{1},3,\mathbf{5}) \rangle$.

Each sequence in the sequence database DB is referred to as a data sequence. Each data sequence is associated with a customer-id (abbreviated as *cid*). The number of data sequences in DB is denoted by $|DB|$. The *support* of sequence s, denoted by $s.sup$, is the number of data sequences containing s divided by the total number of data sequences in DB. The *minsup* is the user specified minimum support threshold. A sequence s is a *frequent sequence*, or called sequential pattern, if $s.sup \geqslant minsup$. Given the *minsup* and the sequence database DB, the problem of sequential pattern mining is to discover *the set of all sequential patterns*, denoted by S^{DB}.

2.2. Incremental update of sequential patterns

In practice, the sequence database will be updated with new transactions after the pattern mining process. Possible updating includes transaction appending, deletions, and modifications. With respect to the same *minsup*, the incremental update problem aims to find out the new set of all sequential patterns after database updating without re-mining the whole database. First, we describe the issue of incremental updating by taking the transaction appending as an illustrating example. Transaction modification can be accomplished by transaction deletion and appending.

The *original database* DB is appended with a few data sequences after some time. The *increment database* db is referred to as the set of these newly appended data sequences. The *cids* of the data sequences in db may already exist in DB. The whole database combining all the data sequences from the original database DB and the increment database db is referred to as the *updated database* UD. Let the *support count* of a sequence s in DB be s^{DB}_{count}. A sequence s is a frequent sequence in UD if $s^{UD}_{count} \geqslant minsup \times |UD|$, where s^{UD}_{count} is the support count of s in UD. Although UD is the union of DB and db, $|UD|$ is not necessarily equal to $|DB|$ plus $|db|$. If there are $|old|$ *cids* appearing both in DB and db, then the number of 'new' customers is $|new| = |db| - |old|$. Thus $|UD| = |DB| + |db| - |old|$ due to sequence merging. When all *cid*s in db are different from

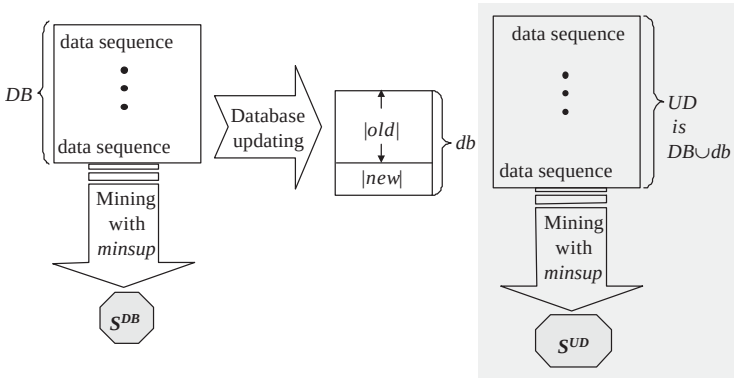
those in DB, $|old|$ (the number of 'old' customers) is zero. On the contrary, $|old|$ equals $|db|$ in case all *cid*s in db exist in DB. Let s^{db}_{count} be the increase in support count of sequence s due to db. Whether sequence s in UD is frequent or not depends on s^{UD}_{count}, with respect to the same *minsup* and $|UD|$.

Most approaches re-execute mining algorithms over all data sequences in UD to obtain s^{UD}_{count} and discover S^{UD}, as shown in Fig. 1a. However, we can effectively calculate s^{UD}_{count} utilizing the support count of each sequential pattern s in S^{DB}. Fig. 1b shows that we discover S^{UD} through incremental update on S^{DB} after implicit merging. Table 1 summarizes the notations used in this paper.

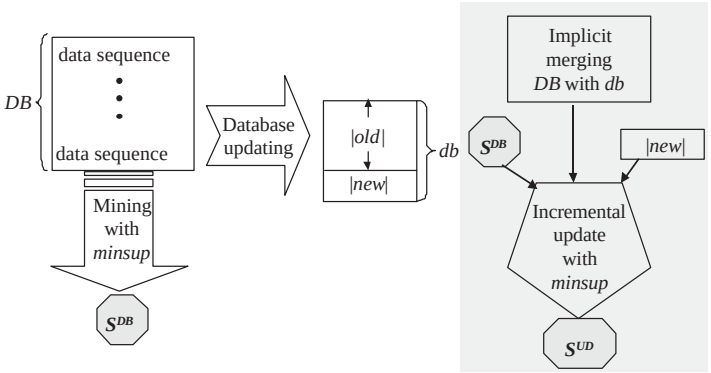
2.3. Changes of sequential patterns due to database update

Consider an example database DB with 6 data sequences as shown in Fig. 2. Assume that *minsup* = 33%, i.e., minimum support count being 2. The sequential patterns in DB are $\langle(1)\rangle$, $\langle(2)\rangle$, $\langle(3)\rangle$, $\langle(4)\rangle$, $\langle(1,2)\rangle$, $\langle(1)(4)\rangle$, $\langle(2)(2)\rangle$, and $\langle(3)(1)\rangle$. Note that $\langle(6)\rangle$, though appeared twice in the same data sequence C6, is not frequent because its support count is one.

Fig. 3a shows the data sequences in the increment database db after some updates from new customers only. The updated database UD is shown in Fig. 3b. Corresponding to the nine data sequences and with the same *minsup*, the support



(a) Obtain S^{UD} by re-executing mining algorithm on UD



(b) Obtain S^{UD} by incremental updating with S^{DB}

Fig. 1. Incremental update versus re-mining.

Table 1
Notations used

$x_1, x_2, \ldots, x_q$	Items
$(x_1, x_2, \ldots, x_q)$	A q-itemset, each x_i is an item
$s = \langle e_1 e_2 \ldots e_n \rangle$	A sequence with n element
$e_1, e_2, \ldots, e_n$	Elements (of a sequence). Each e_i is an itemset
minsup	The minimum support specified by the user
UD	The updated database
DB	The original database
db	The increment database
$\|UD\|, \|DB\|, \|db\|$	The total number of data sequences in *UD*, *DB*, and *db*, respectively
$\|old\|$	The total number of data sequences of 'old' customers in *db*
$\|new\|$	The total number of data sequences of 'new' customers in *db*
S^{DB}, S^{UD}	The set of all sequential patterns in *DB* and *UD*, respectively
s^{DB}_{count}, s^{UD}_{count}	The support counts of candidate sequence s in *DB* and *UD*, respectively
s^{db}_{count}	The increase in support count of candidate sequence s due to *db*
S_k	The set of all frequent k-sequences, see Section 3.1
X_k	The set of all candidate k-sequences, see Section 3.1
$X\kappa'$	The reduced set of candidate k-sequences, see Section 4
S^{DB}_k	The set of frequent k-sequences in *DB*, see Section 4.2
$X\kappa_{(DB)}$	The set of candidates in $X\kappa$ that are also in S^{DB}_k, see Section 4
$X\kappa'_{(DB)}$	$X\kappa'_{(DB)} = X_k - X\kappa_{(DB)}$, see Section 4
$ds^{UD}, ds^{DB}, ds^{db}$	A data sequence in *UD*, *DB*, and *db*, respectively, see Section 4.1
$ds^{DB} \cup ds^{db}$	An implicitly merged data sequence, see Section 4.1
UD_{DB}	Data sequences in *UD* whose *cids* appearing in *DB* only, see Appendix
UD_{db}	Data sequences in *UD* whose cids appearing in *db* only, see Appendix
UD_{Dd}	Data sequences in *UD* whose *cids* are in both *DB* and *db*, see Appendix

Cid	Data Sequence (ds^{DB})
C1	<(1)(4)>
C2	<(2)(3,5)(1,2)>
C3	<(1,2)(2,4)>
C4	<(4)(3)(1)>
C5	<(1)>
C6	<(6)(2,6,7)>

Fig. 2. The original database *DB* example, $|DB| = 6$.

Cid	Data Sequence (ds^{db})
C7	<(2,4)>
C8	<(2,4)(5)>
C9	<(1,2)(5)(2,6)>

(a)

Cid	Data Sequence (ds^{UD})
C1	<(1)(4)>
C2	<(2)(3,5)(1,2)>
C3	<(1,2)(2,4)>
C4	<(4)(3)(1)>
C5	<(1)>
C6	<(6)(2,6,7) >
C7	<(2,4)>
C8	<(2,4)(5)>
C9	<(1,2)(5)(2,6)>

(b)

Fig. 3. Data sequences in the increment database and the updated database (a) *db* with new customers only (b) the updated database *UD*.

count of a frequent sequence must be three or larger. The support counts of previous sequential patterns $\langle(3)\rangle$, $\langle(1)(4)\rangle$, and $\langle(3)(1)\rangle$ are less than three, and are no longer frequent due to the database updates. While $\langle(5)\rangle$, $\langle(2)(5)\rangle$, and $\langle(2,4)\rangle$ become new patterns because they have minimum supports now.

In the cases of updates when the new sequences are from old customers, i.e., the *cid*s of the new sequences appear in the original database. These data sequences must be appended to the old data sequences of the same customers in *DB*. Assume that two customers, *cid* = C4 and *cid* = C8, bought item '8' afterward. The data sequences for *cid* = C4 and *cid* = C8 now become $\langle(4)(3)(1)(8)\rangle$ and $\langle(2,4)(5)(8)\rangle$, respectively. Fig. 4 shows the example of an increment database having data sequences from both old and new customers. In this example, $|old| = 4$, $|new| = 3$, and $|db| = 7$ where records in shadow are old customers. Fig. 5 presents the resulting data sequences in *UD*. After invalidating the patterns $\langle(5)\rangle$, $\langle(2)(2)\rangle$, $\langle(2)(5)\rangle$, and $\langle(1,2)\rangle$, the up-to-date

Cid	Data Sequence (ds^{db})
C2	<(4)>
C4	<(8)>
C5	<(1,4)>
C8	<(8)>
C10	<(2,4,6,8)>
C11	<(1)(7)>
C12	<(2,6)(7)>

Fig. 4. Data sequences of old and new customers in *db*.

Cid	Data Sequence (ds^{UD})
C1	<(1)(4)>
C2	<(2)(3,5)(1,2)(4)>
C3	<(1,2)(2,4)>
C4	<(4)(3)(1)(8)>
C5	<(1)(1,4)>
C6	<(6)(2,6,7)>
C7	<(2,4)>
C8	<(2,4)(5)(8)>
C9	<(1,2)(5)(2,6)>
C10	<(2,4,6,8)>
C11	<(1)(7)>
C12	<(2,6)(7)>

Fig. 5. Merged data sequences in the updated database *UD*.

sequential patterns are $\langle(1)\rangle$, $\langle(2)\rangle$, $\langle(4)\rangle$, $\langle(6)\rangle$, $\langle(2,4)\rangle$, $\langle(2,6)\rangle$ and $\langle(1)(4)\rangle$, for the given *minsup* 33%.

3. Related work

In Section 3.1, we review some algorithms for discovering sequential patterns. Section 3.2 presents related approaches for incremental pattern updating.

3.1. Algorithms for discovering sequential patterns

The *Apriori* algorithm discovers association rules [9], while the *AprioriAll* algorithm deals with the problem of sequential pattern mining [1]. *AprioriAll* splits sequential pattern mining into three phases, itemset phase, transformation phase, and sequence phase. The itemset phase uses *Apriori* to find all frequent itemsets. The database is transformed, with each transaction being replaced by the set of all frequent itemsets contained in the transaction, in the transformation phase. In the third phase, *AprioriAll* makes multiple passes over the database to generate candidates and to count the supports of candidates. In subsequent work, the same authors proposed the *G*eneralized *S*equential *P*attern (*GSP*) algorithm that outperforms *AprioriAll* [7]. Both algorithms use the similar techniques for candidate generation and support counting, as described in the following.

The GSP algorithm makes multiple passes over the database and finds out frequent k-sequences at kth database scanning. In each pass, every data sequence is examined to update the support counts of the candidates contained in this sequence. Initially, each item is a candidate 1-sequence for the first pass. Frequent 1-sequences are determined after checking all the data sequences in the database. In succeeding passes, frequent $(k-1)$-sequences are self-joined to generate candidate k-sequences. Again, the supports of these candidate sequences are counted by examining all data sequences, and then those candidates having minimum supports become frequent sequences. This process terminates when there is no candidate sequence any more. In the following, we further depict two essential sub-processes in GSP, the candidate generation and the support counting.

Candidate generation: Let S_k denote the set of all frequent k-sequences, and X_k denote the set of all candidate k-sequences. GSP generates X_k by two steps. The first step joins S_{k-1} with S_{k-1} and obtains a superset of the final X_k. Those candidates in the superset having any $(k-1)$-subsequence which is not in S_{k-1} are deleted in the second step. In the first step, a $(k-1)$-sequence $s1 = \langle e_1 e_2 \ldots e_{n-1} e_n \rangle$ is joined with another $(k-1)$-sequence $s2 = \langle e'_1 e'_2 \ldots \boldsymbol{e'_n} \rangle$ if $\overline{s1} = \overline{s2}$, where $\overline{s1}$ is the $(k-2)$-sequence of $s1$ dropping the first item of e_1 and $\overline{s2}$ is the $(k-2)$-sequence of $s2$ dropping the last item of $\boldsymbol{e'_n}$. The generated candidate k-sequence $s3$ is $\langle e_1 e_2 \ldots e_{n-1} e_n \boldsymbol{e'_n} \rangle$ if $\boldsymbol{e'_n}$ is a 1-itemset. Otherwise, $s3$ is $\langle e_1 e_2 \ldots e_{n-1} e_n \boldsymbol{e'_n} \rangle$. For example, the candidate 5-sequence $\langle(1,2)(3,5)(\mathbf{6})\rangle$ is generated by joining $\langle(1,2)(3,5)\rangle$ with $\langle(2)(3,5)(\mathbf{6})\rangle$, and the candidate $\langle(1,2)(3,5)(\mathbf{6})\rangle$ is generated by joining $\langle(1,2)(3,5)\rangle$ with $\langle(2)(3,5,\mathbf{6})\rangle$. In addition, the X_k produced from this procedure is a superset of S_k as proved in [7]. That is, $X_k \supseteq S_k$.

Support counting: GSP adopts a hash-tree structure [9,7] for storing candidates to reduce the number of candidates that need to be checked for each data sequence. Candidates would be placed in the same leaf if their leading items, starting from the first item, were hashed to the same node. The next item is used for hashing when an interior node, instead of a leaf node, is reached [7]. The candidates required for checking against a data sequence are located in leaves reached by applying the hashing procedure on each item of the data sequence [7]. The support of the candidate is incremented by one if it is contained in the data sequence.

In addition, the *SPADE* (Sequential PAttern Discovery using Equivalence classes) algorithm finds out sequential patterns using vertical database layout and join-operations [8]. Vertical database layout transforms customer sequences into items' id-lists. The id-list of an item is a list of (*cid*, *timestamp*) pairs indicating the occurring timestamps of the item in that customer-id. The list pairs are joined to form a sequence lattice, in which SPADE searches and discovers the patterns [8].

Recently, the *FreeSpan* (*Fre*qu*e*nt pattern-projected *S*equential *Patte*r*n* Mining) algorithm was proposed to mine sequential patterns by a database projection technique [3]. *FreeSpan* first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller intermediate databases. Finally, all sequential patterns are found by recursively growing subsequence fragments in each database. Based on the similar projection technique, the authors proposed the *PrefixSpan* (*Prefix*-projected *S*equential *pa*ttern mining) algorithm [14].

Nevertheless, all these algorithms have to re-mine the database after the database is appended with new data sequences. Next, we introduce some approaches for updating patterns without re-mining.

3.2. Approaches for incremental pattern updating

A work for incremental sequential pattern updating was proposed in [18]. The approach uses a dynamic suffix tree structure for incremental mining in a single long sequence. However, the focus of research here is on multiple sequences of itemsets, instead of a single long sequence of items.

Based on the SPADE algorithm, the ISM algorithm was proposed for incremental sequence mining [5]. An *Increment Sequence Lattice* consisting of both frequent sequences and the nearly frequent ones (called *negative border*) is built to prune the search space for potential new patterns. However, the ISM might encounter memory problem if the number of the potentially frequent patterns is too large [5]. Besides, computation is required to transform the sequence database into vertical layout, which also requires additional storage several times the original database.

In order to avoid re-mining from scratch with respect to database updates with both old and new customers, we propose a pattern updating approach that incrementally mines sequential patterns by utilizing the discovered knowledge. Section 4 gives the details of the proposed algorithm.

4. The proposed algorithm

In sequence mining, frequent patterns are those candidates whose supports are greater than or equal to the minimum support. In order to obtain the supports, every data sequence in the database is examined, and the support of each candidate contained in that data sequence is incremented by one. For pattern updating after database update, the database DB was already mined and the supports of the frequent patterns with respect to DB are known. Intuitively, the number of data sequences need to be examined in current updating with database UD seems to be $|UD|$. However, we can utilize the prior knowledge to improve the overall updating efficiency. Therefore, we propose the *IncSP* (*Inc*remental *S*equential *P*attern Update) algorithm to speed up the incremental updating problem. Fig. 6 depicts the architecture of a single pass in the IncSP algorithm. In brief, IncSP incrementally updates and discovers the sequential patterns through effective implicit

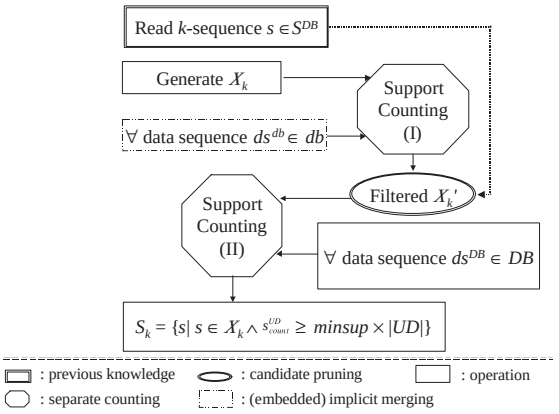


Fig. 6. The architecture of the kth pass in IncSP.

merging, early candidate pruning, and efficient separate counting.

The data sequence of a customer in *DB* and the sequence with same *cid* in *db* must be merged into the customer's data sequence in *UD*. If all such sequences are merged explicitly, we have to re-mine and re-count the supports of the candidates contained in the resultant customer sequences from scratch. Hence, IncSP deals with the required sequence merging implicitly for incremental pattern updating, which is described in Section 4.1.

IncSP further speeds up the support counting by partitioning the candidates into two sets. The candidates with respect to *DB* which were also frequent patterns before updating are placed into set $\boldsymbol{X\kappa}_{(DB)}$, and the remaining candidates are placed into set $\boldsymbol{X\kappa}_{(DB)}$. After the partitioning, the supports of the candidates in $\boldsymbol{X\kappa}_{(DB)}$ can be incremented and updated simply by scanning over the increment database *db*. During the same scanning, we also calculate the increment supports of the candidates in $\boldsymbol{X\kappa'}_{(DB)}$ with respect to *db*. Since the supports of the candidates in $\boldsymbol{X\kappa'}_{(DB)}$ are not available (only the supports of frequent patterns in *DB* are kept in prior mining over *DB*), we need to compute their supports against the data sequences in *DB*. The number of candidates need to be checked is reduced to the size of set $\boldsymbol{X\kappa'}_{(DB)}$ instead of the full set X_k. Thus, IncSP divides the counting procedure into separate processes to efficiently count the supports of candidates with respect to *DB* and *db*. We show that the support of a candidate is the sum of the two support counts after the two counting processes in Lemma 1 (in Section 4.2).

Moreover, some candidates in $\boldsymbol{X\kappa'}_{(DB)}$ can be pruned earlier before the actual counting over the data sequences in *DB*. By partitioning the set of candidates into $\boldsymbol{X\kappa}_{(DB)}$ and $\boldsymbol{X\kappa'}_{(DB)}$, we know that all the candidates in $\boldsymbol{X\kappa'}_{(DB)}$ are not frequent patterns with respect to *DB*. If the support of a candidate in $\boldsymbol{X\kappa'}_{(DB)}$ with respect to *db* is smaller than the proportion $minsup \times (|UD| - |DB|)$, the candidate cannot possibly become a frequent pattern in *UD*. Such unqualifying candidates are pruned and only the more promising candidates go through the actual support counting over *DB*. Lemma 2 (in Section 4.2) shows this property. This early pruning further reduces the number of candidates required to be counted against the data sequences in *DB*. The reduced set of candidates is referred to as $\boldsymbol{X\kappa'}$.

In essence, IncSP generates candidates and examines data sequences to determine frequent patterns in multiple passes. As shown in Fig. 6, IncSP reduces the size of X_k into $\boldsymbol{X\kappa'}$ and updates the supports of patterns in S^{DB} by simply checking the increment database *db*, which is usually smaller than the original database *DB*. In addition, the *separate counting* technique enables IncSP to accumulate candidates' supports quickly because only the *new* candidates, whose supports are unavailable from S^{DB}, need to be checked against *DB*. The complete IncSP algorithm and the separate counting are described in Section 4.2. Section 4.3 further illustrates other updating operations such as modifications and deletions.

4.1. Implicit merging of data sequences with same cids

For the discovery of sequential patterns, transactions coming from the same customer, either in *DB* or in *db*, are parts of the unique data sequence corresponding to that customer in *UD*. Given a customer having one data sequence in *DB* and another sequence in *db*, the proper data sequence for the customer (in *UD*) is the merged sequence of the two. Since the transaction times in *db* are later

than those in *DB*, the merging appends the data sequences in *db* to the sequences in *DB*. Nevertheless, such "explicit merging" might invalidate S^{DB} because the data sequence of the customer becomes a longer sequence. Some patterns in S^{DB}, which are not contained in the data sequence before merging, might become contained in the now longer data sequence so that the support counts of these patterns become larger. In order to effectively keep the patterns in S^{DB} up-to-date, IncSP implicitly merges data sequences of the same customers and delays the actual action of merging until pattern updating completes.

Assume that an explicit merging must merge ds^{DB} with ds^{db} into ds^{UD}, where ds^{DB}, ds^{db}, and ds^{UD} represent the data sequences in *DB*, *db*, and *UD*, respectively. In each pass, the mining process needs to count the supports of candidate sequences against ds^{UD}. The "implicit merging" in IncSP employs ds^{DB} and ds^{db} as if ds^{UD} is produced during mining process. We will describe how "implicit merging" updates the supports of sequential patterns in S^{DB}, and how "implicit merging" counts the supports of candidates contained in the implicitly merged data sequence, represented by $ds^{DB} \cup ds^{db}$.

The "implicit merging" updates the supports of sequential patterns in S^{DB} according to ds^{DB} and ds^{db}. This updating involves only the newly generated (candidate) k-sequences in the kth pass, which are contained in ds^{UD} but not in ds^{DB}, since ds^{DB} had already engaged in the discovery of S^{DB}. We refer to these candidate k-sequences as the *new k-sequences*. As indicated in Fig. 6, when ds^{db} is checked in Support Counting (I), only the supports of such *new k-sequences* must be counted. If this *new k-sequence* is also a sequential pattern in S^{DB}, we update the support count of the sequence in S^{DB}. Otherwise, supports of *new k-sequences* which are not in S^{DB}, being initialized to zero before counting, are incremented by one for this data sequence ($ds^{DB} \cup ds^{db}$). In this way, IncSP correctly maintains S^{DB} with the *new k-sequences* and counts supports with respect to ds^{db} during Support Counting (I).

Example 1. Implicit merging for support updating in pass-1. Take customers in Fig. 5 for example, the *DB* is shown in Fig. 3b and the *db* is shown in Fig. 4. The customer with *cid* = C2 has the two sequences, $ds^{DB} = \langle(2)(3,5)(1,2)\rangle$ and $ds^{db} = \langle(4)\rangle$. During pass 1, $\langle(4)\rangle^{DB}_{count}$ is increased by one due to the implicit merging with ds^{db} and ds^{DB} (of C2). Note that implicit merging for the customer with *cid* = C5 whose $ds^{DB} = \langle(1)\rangle$ and $ds^{db} = \langle(1,4)\rangle$ contains only the *new 1*-sequence $\langle(4)\rangle$ because $\langle(1)\rangle$ was already counted when we examined ds^{DB} to produce S^{DB}. Eventually, the support count $\langle(4)\rangle^{DB}_{count}$ is increased by two considering the two implicitly merged sequences of C2 and C5. Similarly, the support count of candidate $\langle(8)\rangle^{DB}_{count}$ is two after the implicit merging on customer sequences whose *cid*s = C4 and C8.

4.2. The IncSP algorithm

The implicit merging technique preserves the correctness of supports of the patterns and enables IncSP to count the supports in *DB* and *db* separately for pattern updating. Fig. 7 lists the proposed IncSP algorithm and Fig. 8 depicts the two separate sub-processes of support counting in the IncSP algorithm. Through separate counting, we do not have to check the full candidate set X_k against all data sequences from *db* and *DB*. Only the (usually) smaller *db* must take all the candidates in X_k into consideration for support updating. Furthermore, we can prune previous patterns and leave fewer but more promising candidates in $\boldsymbol{X\kappa'}$ before applying the data sequences in *DB* for support counting.

The IncSP algorithm generates candidates and computes the supports for pattern updating in multiple passes. In each pass, we initialize the two support counts of each candidate in *UD* to zero, and read the support count of each frequent k-sequence s in *DB* to s^{DB}_{count}. We then accumulate the increases in support count of candidates with respect to the sequences in *db* by Support Counting (I). Before Support Counting (II) starts, candidates which are frequent in *DB* but cannot be frequent in *UD* according to Lemma 4 are filtered out. The full candidate set X_k is reduced into the set $\boldsymbol{X\kappa'}$. Next, the Support Counting (II) calculates

1) /* Initially, each item is a candidate *1*-sequence */
2) X_1 = set of candidate 1-sequences;
3) $k = 1$; /* Start from pass 1 */
4) repeat /* Find frequent k-sequences in the k-th pass */
5) for each $s \in X_k$ do $s^{DB}_{count} = s^{db}_{count} = 0$; /* Initialize counters */
6) Read S^{DB}_k ; /* S^{DB}_k = {frequent k-sequences in DB} */
7) Check sequences in db by **Support Counting (I)** ; /* See Fig. 8 */
8) /* Prune candidates: (1) counted in S^{DB}_k (2) insufficient "new" counts */
9) $X_k' = X_k - \{s | s \in S^{DB}_k\} - \{s | \ s^{db}_{count} < minsup \times (|UD| - |DB|)\}$;
10) Check sequences in DB by **Support Counting (II)** ;/* See Fig. 8 */
11) /* Frequent k-sequences in UD found */
12) $S_k = \{s | \ s \in X_k \wedge \ s^{DB}_{count} + s^{db}_{count} \geq minsup \times |UD|\}$;
13) $k = k + 1$;
14) Generate C_k with S_{k-1} ; /* Generate candidates for next pass */
15) until no more candidates
16) Answer $S^{UD} = \cup_k S_k$;

Fig. 7. Algorithm IncSP.

Support Counting (I):
/* Updating "old" supports and counting candidates against data sequences in db */
1) for each data sequence ds^{db} in db do
2) if cid of ds^{db} is not found in DB then /* ds^{db} is a new customer's sequence */
3) /* Increment s^{db}_{count} by 1 if s is contained in ds^{db} */
4) for each $s \in X_k \wedge s \subseteq ds^{db}$ do s^{db}_{count}++ ;
5) endif
6) if cid of ds^{db} is found in DB then /* ds^{db} should be appended to ds^{DB} */
7) for each $s \in X_k \wedge s \subseteq (ds^{DB} \cup ds^{db})$ do /* Implicit merging and counting */
8) /* Increment s^{db}_{count} by 1 if s is contained in ds^{db} but not in ds^{DB} */
9) if $s \not\subset ds^{DB}$ then s^{db}_{count}++ ;
10) endfor
11) endif
12) endfor

Support Counting (II):
/* Counting "new" candidates against data sequences in DB */
1) for each data sequence ds^{DB} in DB do
2) /* Increment s^{DB}_{count} by 1 if s is contained in ds^{DB} */
3) for each $s \in X_k' \wedge s \subseteq ds^{DB}$ do s^{DB}_{count}++ ; /* X_k' is the reduced candidate set */
4) endfor

Fig. 8. The separate counting procedure.

the support counts of these promising candidates with respect to the sequences in DB. As indicated in Lemma 1, the support count of any candidate k-sequence is the sum of the two counts obtained after the two counting processes. Consequently, we can discover the set of frequent k-sequences S_k by validating the sum of the two counts of every candidate. The S_k is used to generate the complete candidate set for the next pass, employing the similar candidate generation procedure in GSP.

The above process is iterated until no more candidates.

We need to show that IncSP updates the supports and discovers frequent patterns correctly. Several properties used in the IncSP algorithm are described as follows. The details of the proof of the lemmas are included in Appendix.

Lemma 1. *The support count of any candidate k-sequence s in UD is equal to* $s^{DB}_{count} + s^{db}_{count}$.

Lemma 2. *A candidate sequence s, which is not frequent in DB, is a frequent sequence in UD only if* $s^{db}_{count} \geqslant minsup \times (|UD| - |DB|)$.

Lemma 3. *The separate counting procedure* (*in* Fig. 8) *completely counts the supports of candidate k-sequences against all data sequences in UD.*

Lemma 4. *The candidates required for checking against the data sequences in DB in Support Counting (II) is the set* $\boldsymbol{X\kappa'}$, *where* $\boldsymbol{X\kappa'} = X_k - \{s|s \in S^{DB}_k\} - \{s|s^{db}_{count} < minsup \times (|UD| - |DB|)\}$.

Theorem 1. *IncSP updates the supports and discovers frequent patterns correctly.*

Proof. In IncSP, we use the candidate generation procedure analogous to GSP to produce the complete set of candidates in X_k. By Lemma 3, the separate counting procedure completely counts the supports of candidate k-sequences against all data sequences in UD. Lemma 1 determines frequent patterns in UD and the updated supports. Therefore, IncSP correctly maintains sequential patterns. □

Example 2. Sequential pattern updating using IncSP. The data sequences in the original database DB is shown in Fig. 3b. The *minsup* is 33%. S^{DB} is listed in Table 2. The increment database db is shown in Fig. 4. IncSP discovers S^{UD} as follows.

Pass 1:

(1) Generate candidates for pass *1*, $X_1 = \{\langle(1)\rangle, \langle(2)\rangle, \ldots, \langle(8)\rangle\}$.
(2) Initialize the two counts of each candidate in X_1 to zero, and read S^{DB}_1.
(3) After Support Counting (I), the increases in support count are listed in Part (b) of Table 2. Note that for customer with *cid* = C5, the increase in support count of $\langle(1)\rangle$ is not changed. Now $|UD| = 12$ and $|DB| = 9$. Since

Table 2
Sequences and support counts for Example 2

Part (a): S^{DB}		Part (b): Pass *1*		Part (c): Pass *2*		Part (d): S^{UD}	
s^{DB}_{count}		Support counting (I) s^{db}_{count}		Support counting (I) s^{db}_{count}		s^{UD}_{count}	
$\langle(1)\rangle$	6	$\langle(1)\rangle$	1	$\langle(1)(1)\rangle$	1	$\langle(1)\rangle$	7
$\langle(2)\rangle$	6	$\langle(2)\rangle$	2	$\langle(1)(4)\rangle$	2	$\langle(2)\rangle$	8
$\langle(4)\rangle$	5	$\langle(4)\rangle$	3	$\langle(2)(4)\rangle$	1	$\langle(4)\rangle$	8
$\langle(5)\rangle$	3	$\langle(6)\rangle$	2	$\langle(2,4)\rangle$	1	$\langle(6)\rangle$	4
$\langle(2)(2)\rangle$	3	$\langle(7)\rangle$	2	$\langle(2,6)\rangle$	2	$\langle(1)(4)\rangle$	4
$\langle(2)(5)\rangle$	3	$\langle(8)\rangle$	3	$\langle(4,6)\rangle$	1	$\langle(2,4)\rangle$	4
$\langle(1,2)\rangle$	3	$\langle(3)\rangle$	0	$\langle(1,4)\rangle$	1	$\langle(2,6)\rangle$	4
$\langle(2,4)\rangle$	3	$\langle(5)\rangle$	0	Others	0		
		Support counting (II) s^{DB}_{count}		Support counting (II) s^{DB}_{count}			
		$\langle(6)\rangle$	2	$\langle(1)(4)\rangle$	2		
		$\langle(7)\rangle$	1	$\langle(2)(4)\rangle$	1		
		$\langle(8)\rangle$	0	$\langle(2,6)\rangle$	2		
				$\langle(1)(1)\rangle$	0		
				$\langle(1,4)\rangle$	0		
				$\langle(4,6)\rangle$	0		

$S_1^{DB} = \{\langle(1)\rangle, \langle(2)\rangle, \langle(4)\rangle, \langle(5)\rangle\}$ and the increase in support count of $\langle(3)\rangle$ are less than $33\% \times (|UD| - |DB|)$, the reduced set X_1' is $\{\langle(6)\rangle, \langle(7)\rangle, \langle(8)\rangle\}$.

(4) After Support Counting (II), the s_{count}^{DB} of $\langle(6)\rangle$ and $\langle(7)\rangle$ are 2 and 1, respectively. The minimum support count is 4 in *UD*. IncSP obtains the updated frequent *1*-sequences, which are $\langle(1)\rangle$, $\langle(2)\rangle$, $\langle(4)\rangle$, and $\langle(6)\rangle$. Total 22 candidate *2*-sequences are generated with the four frequent *1*-sequences.

Pass 2:

(5) We read S_2^{DB} after initializing the two support counts of all candidate *2*-sequences. Note that the s_{count}^{DB} of $\langle(2)(5)\rangle$ is useless because $\langle(2)(5)\rangle$ is not a candidate in *UD* in this pass.

(6) We list the result of Support Counting (I) in Part (c) of Table 2. The increases in support count of some candidates, such as $\langle(1,6)\rangle$ or $\langle(4)(6)\rangle$, are all zero and are not listed.

(7) Again, we compute the X_2' so that the candidates need to be checked against the data sequences in *DB* are $\langle(1)(1)\rangle$, $\langle(1)(4)\rangle$, $\langle(1,4)\rangle$, $\langle(2)(4)\rangle$, $\langle(2,6)\rangle$, and $\langle(4,6)\rangle$. We filter out 16 candidates (13 candidates with insufficient "support increases" and 3 candidates in S_2^{DB}) before Support Counting (II) starts.

(8) The s_{count}^{DB} of $\langle(1)(4)\rangle$, $\langle(2)(4)\rangle$, and $\langle(2,6)\rangle$ are 2, 1, and 2, respectively, after Support Counting (II). IncSP then sums up the counts (s_{count}^{DB} and s_{count}^{db}) to obtain the updated frequent *2*-sequences. Finally, IncSP terminates since no candidate *3*-sequence is generated. Part (d) of Table 2 lists the sequential patterns and their support counts in *UD*.

In comparison with GSP, IncSP updates supports of sequential patterns in S^{DB} by scanning data sequences in *db* only. New sequential patterns, which are not in *DB*, are generated from fewer candidate sequences comparing with previous methods. The support increases of new candidates are checked in advance and leave the most promising candidates for Support Counting (II) against data sequences in *DB*. Every candidate in the reduced set is then checked against *DB* to see if it is frequent in *UD*. On the contrary, GSP takes every candidate and counts over all data sequences in the updated database. Consequently, IncSP is much faster than GSP as shown in the experimental results.

4.3. Pattern maintenance on transaction deletion and modification

Common operations on constantly updated databases include not only appending, but also deletions and modifications. Deleting transactions from a data sequence changes the sequence; thereby changing the supports of patterns contained in this sequence. The supports of the discovered patterns might decrease but no new patterns would occur. We check patterns in S^{DB} against these data sequences. Assume that a data sequence *ds* is changed to *ds'* due to deletion. The *ds'* is an empty sequence when all transactions in *ds* are deleted. If a frequent sequence *s* is contained in *ds* but not in *ds'*, s_{count}^{DB} is decreased by one. The resulting sequential patterns in the updated database are those patterns still having minimum supports.

A transaction modification can be accomplished by deleting the old transaction and then inserting the new transaction. In IncSP, we delete the original data sequence from the original database, create a new sequence comprising the substituted transaction(s), and then append the new sequence to the increment database.

5. Performance comparisons and experimental results

In order to assess the performance of the IncSP algorithm, we conducted comprehensive experiments using an 866 MHz Pentium-III PC with 1024 MB memory. In these experiments, the databases are composed of synthetic data. The method used to generate these data is described in Section 5.1. Section 5.2 compares the performance and resource consumption of algorithms GSP, ISM and IncSP. Results of scale-up experiments are presented in Section 5.3. Section 5.4 discusses the memory requirements of these algorithms.

5.1. Synthetic data generation

Updating the original database DB with the increment database db was modeled by generating the update database UD, then partitioning UD into DB and db. Synthetic transactions covering various data characteristics were generated by the well-known method in [1]. Since all sequences were generated from the same statistical patterns, it might model real updates very well.

At first, total $|UD|$ data sequences were created as UD. Three parameters are used to partition UD for simulating different updating scenarios. Parameter R_{inc}, called *increment ratio*, decides the size of db. Total $|db| = |UD| \times R_{inc}$ sequences were randomly picked from UD into db. The remaining $|UD| - |db|$ sequences would be placed in DB. The *comeback ratio* R_{cb} determines the number of "old" customers in db. Total $|old| = |db| \times R_{cb}$ sequences were randomly chosen from these $|db|$ sequences as "old" customer sequences, which were to be split further. The splitting of a data sequence is to simulate that some transactions were conducted formerly (thus in DB), while the remaining transactions were newly appended. The splitting was controlled by the third parameter R_f, the *former ratio*. If a sequence with total $|ds^{UD}|$ transactions was to split, we placed the leading $|ds^{DB}| = |ds^{UD}| \times R_f$ transactions in DB and the remaining $|ds^{UD}| - |ds^{DB}|$ transactions in db. For example, a UD with $R_{inc} = 20\%$, $R_{cb} = 30\%$, and $R_f = 40\%$ means that 20% of sequences in UD come from db, 30% of the sequences in db have cids occurring in DB, and that for each "old" customer, 40% of his/her transactions were conducted before current pattern updating. (Note: The calculation is integer-based with 'ceiling' function. E.g. $|ds^{UD}| = 4$, $|ds^{DB}| = \lceil 4 * 40\% \rceil = 2$.)

We now review the details of data sequence generation, as described in [1]. In the modeled retailing environment, each customer purchases a sequence of itemsets. Such a sequence is referred to as a *potentially frequent sequence* (*PFS*). Still, some customers might buy only some of the items from a PFS. A customer's data sequence may contain more than one PFS. The PFSs are composed of *potentially frequent itemsets* (*PFIs*).

Table 3 summarizes the symbols and the parameters used in the experiments. A database generated with these parameters is described as follows. The updated database has $|UD|$ customer sequences, each customer has $|C|$ transactions on average, and each transaction has average $|T|$ items. A table of total N_I PFIs and a table of total N_S PFSs were generated before picking items for the transactions of customer sequences. On average, a PFS has $|S|$ transactions and a PFI has $|I|$ items. The total number of possible items for all PFIs is N.

The number of transactions for the next customer and the average size of transactions for this customer are determined first. The *size of the customer's data sequence* is picked from a Poisson distribution with mean equal to $|C|$. The average *size of the transaction*s is picked from a Poisson distribution with mean equal to $|T|$. Items are then assigned to the transactions of the customer. Each customer is assigned a series of PFSs from table Γ_S, the table of PFSs. Next, we describe the generation of PFS and then the assignment of PFS.

The *number of itemsets in a PFS* is generated by picking from a Poisson distribution with mean equal to $|S|$. The itemsets in a PFS are picked from table Γ_I, the table of PFIs. In order to model that there are common itemsets in frequent sequences, subsequent PFSs in Γ_S are related. In the subsequent PFS, a fraction of itemsets are chosen from the previous PFS and the other itemsets are picked at random from Γ_I. The fraction $corr_S$, called *correlation level*, is decided by an exponentially distributed random variable with mean equal to μ_{corr_S}. Itemsets in the first PFS in Γ_S are randomly picked. The generations of PFI and Γ_I are analogous to the generations of PFS and Γ_S, with parameters N items, mean $|I|$, *correlation level* $corr_I$ and mean μ_{corr_I} correspondingly.

The assignment of PFSs is based on the weights of PFSs. The weight of the PFS, representing the probability that this PFS will be chosen, is exponentially distributed and then normalized in such a way that the sum of all the weights is equal to one. Since all the itemsets in a PFS are not always bought together, each sequence in Γ_S is assigned a *corruption level* $crup_S$. When selecting

Table 3
Parameters used in the experiments

Parameter	Description	Value
$\|UD\|$	Number of data sequences in database UD	10 K, 100 K, 250 K, 500 K, 750 K, 1000 K
$\|C\|$	Average size (number of transactions) per customer	10, 20
$\|T\|$	Average size (number of items) per transaction	2.5, 5
$\|S\|$	Average size of potentially sequential patterns	4, 8
$\|I\|$	Average size of potentially frequent itemsets	1.25, 2.5
N	Number of possible items	1000, 10,000
N_I	Number of potentially frequent itemsets	25,000
N_S	Number of possible sequential patterns	5000
Γ_S	The table of potentially frequent sequences (PFSs)	
Γ_I	The table of potentially frequent itemsets (PFIs)	
$corr_S$	Correlation level (sequence), exponentially distributed	$\mu_{crup_S} = 0.25$
$crup_S$	Corruption level (sequence), normally distributed	$\mu_{crup_S} = 0.75$, $\sigma_{crup_S} = 0.1$
$corr_I$	Correlation level (itemset), exponentially distributed	$\mu_{crup_I} = 0.25$
$crup_I$	Corruption level (itemset), normally distributed	$\mu_{crup_I} = 0.75$, $\sigma_{crup_I} = 0.1$
R_{inc}	Ratio of increment database db to updated database UD	1%, 2%, 5%, 8%, 10%, 20%, 30%,…, 90%
R_{cb}	Ratio of *c*ome*b*ack customers to all customers in increment database db	0%, 10%, 25%, 50%, 75%, 100%
R_f	Ratio of former transactions to all transactions for each "old" customer	10%, 20%,…, 90%

itemsets from a PFS to a customer sequence, an itemset is dropped as long as a uniformly distributed random number between 0 and 1 is less than $crup_S$. The $crup_S$ is a normally distributed random variable with mean μ_{crup_S} and variance σ_{crup_S}. The assignment of PFIs (from Γ_I) to a PFS is processed analogously with parameters $crup_I$, mean μ_{crup_I} and variance σ_{crup_I} correspondingly.

All datasets used here were generated by setting μ_{crup_S} and μ_{crup_I} to 0.75, σ_{crup_S} and σ_{crup_I} to 0.1, μ_{corr_S} and μ_{corr_I} to 0.25, $N_S = 5000$, $N_I = 25000$. Two values of N (1000 and 10000) were used. A dataset created with $|C| = \alpha$, $|T| = \beta$, $|S| = \chi$, and $|I| = \delta$ is denoted by the notation Cα.Tβ.Sχ.Iδ.

5.2. Performance comparisons of GSP, ISM, and IncSP

To realize the performance improvements of IncSP, we first compare the efficiency of incremental updating with that of re-mining from scratch, and then contrast that with other incremental mining approaches. The well-known GSP algorithm [7], which is a re-mining based algorithm, is used as the basis for comparison. The *PrefixSpan* algorithm [14] mines patterns by recursively projecting data sequences to smaller intermediate databases. Starting from prefix-items (the frequent items), sequential patterns are found by recursively growing subsequence fragments in each intermediate database. Except re-mining, mechanisms of modifying *PrefixSpan* to solve incremental updating is not found in the literature. Since it demands a totally different framework to handle the sequence projection of the original database and the increment database, the *PrefixSpan* is not included in the experiments. The ISM algorithm [5], which is the incremental mining version of the SPADE algorithm [8], deals with database update using databases of vertical layout. We pre-processed the databases for ISM into vertical layout and the pre-processing time is not counted in the following context.

Extensive experiments were performed to compare the execution times of GSP, ISM, and IncSP with respect to critical factors that reflect the performance of incremental updating, including *minsup*, increment ratio, comeback ratio, and former ratio. We set $R_{inc} = 10\%$, $R_{cb} = 50\%$, and $R_f = 80\%$ to model common database updating

scenarios. The dataset has 20,000 sequences ($|UD| = 20\,\mathrm{K}$, 3.8 MB), generated with $|C| = 10$, $|T| = 2.5$, $|S| = 4$, $|I| = 1.25$.

The effect on performance with various *minsup*s was evaluated first. Re-mining is less efficient than incremental updating, as indicated in Fig. 9. In the experiments, both ISM and IncSP are faster than GSP for all values of minimum supports. Fig. 9a shows that ISM is faster than IncSP when the number of items (N) is 1000 and $minsup \leqslant 1\%$. When N is 10,000, IncSP outperforms ISM for all values of *minsup*, as shown in Fig. 9b. The total execution time is longer for all the three algorithms for smaller *minsup* value, which allows more patterns to pass the frequent threshold. GSP suffers from the explosive growth of the number of candidates and the re-counting of supports for each pattern. For example, when *minsup* is 1% and $N = 10,000$, the number of candidate 2-sequences in GSP is 532,526 and that of 'new' candidate 2-sequences in IncSP is 59. Only 59 candidate 2-sequences required counting over the data sequences in *UD*. The other candidate 2-sequences are updated, rather than re-counted, against the 2000 sequences in *UD* ($UD \times 10\%$).

Comparing Fig. 9a with Fig. 9b, it indicates that ISM is more efficient with a smaller N. ISM keeps all frequent sequences, as well as the maximally potential frequent sequences (negative borders), in memory. Take $minsup = 0.75\%$ for example. The number of frequent sequences is 701 for $N = 1000$ and 1017 for $N = 10,000$, respectively. Accordingly, the size of negative borders of size two is 736,751 and 1,550,925, respectively. Those turn-into-frequent patterns that were in negative borders before database updating must intersect with the complete set of frequent patterns. Consequently, with a smaller *minsup* like 0.75%, the larger N provides more possible items to pass the frequent threshold so that the total execution is less efficient in ISM. Instead of frequent-pattern intersection, IncSP deals with candidates separately, the explosively increased frequent items (because of the larger N) affect the efficiency of the pattern updating less. This also accounts for the performance gaps between IncSP and ISM, no matter how increment ratio, comeback ratio or former ratio changes.

The results of varying increment ratio from 1% to 50% are shown in Fig. 10. The *minsup* is fixed at 2%. In general, IncSP gains less at higher increment ratio because larger increment

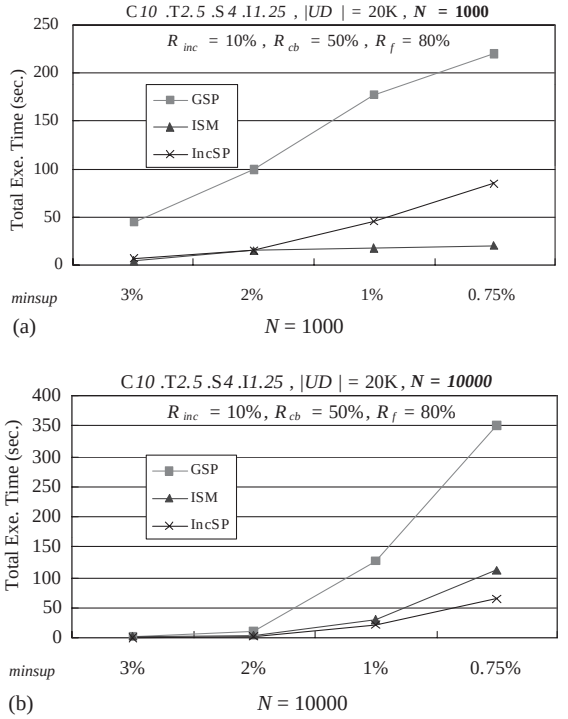


Fig. 9. Total execution times over various *minsup:* (a) $N = 1000$ and (b) $N = 10,000$.

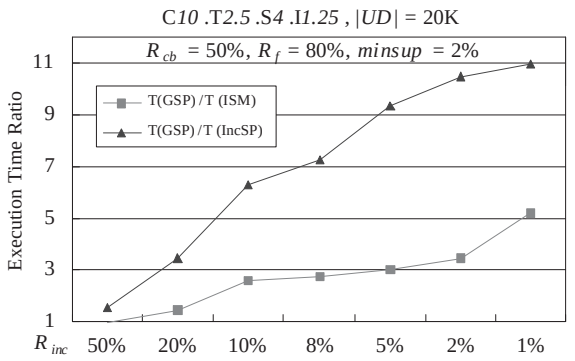


Fig. 10. Total execution times over various incremental ratios.

ratio means more sequences appearing in *db* and causes more pattern updatings. As indicated in Fig. 10, the smaller the increment database *db* is, the more time on the discovery IncSP could save.

IncSP is still faster than GSP even when increment ratio is 50%. When increment ratio becomes much larger, say over 60%, IncSP is slower than GSP. Clearly, when most of the frequent sequences in *DB* turn out to be invalid in *UD*, the information used by IncSP in pattern updating might become useless. When the size of the increment database becomes larger than the size of the original database, i.e. the database has accumulated dramatic change and not incremental change any more, re-mining might be a better choice for the total new sequence database.

The impact of the comeback ratio is presented in Fig. 11. IncSP updates patterns more efficiently than GSP and ISM for all the comeback ratios. High comeback ratio means that there are many 'old' customers in the increment database. Consequently, the speedup ratio decreases as the comeback ratio increases because more sequence merging is required. Fig. 11 shows that IncSP was efficient with implicit merging, even when the comeback ratio was increased to 100%, i.e., all the sequences in the increment database must be merged.

Fig. 12 depicts the performance comparisons concerning former ratios. It can be seen from the figure that IncSP was constantly about 6.5 times faster than GSP over various former ratios, ranging from 10% to 90%.

5.3. Scale-up experiments

To assess the scalability of our algorithm, several experiments of large databases were conducted. Since the basic construct of IncSP is similar to that of GSP, similar scalable results could be expected. In the scale-up experiments, the total number of customers was increased from 100 K (18.8 MB) to 1000 K (187.9 MB), with fixed parameters C*10*.T*2.5*.S*4*.I*1.25*, $N = 10,000$, $R_{inc} = 10\%$, $R_{cb} = 50\%$, and $R_f = 80\%$. Again, IncSP are faster than GSP for all the datasets. The execution times were normalized with respect to the execution time for 100 K customers here. Fig. 13 shows that the execution time of IncSP increases linearly as the database size increases, which demonstrates good scalability of IncSP.

5.4. Memory requirements

Although IncSP uses separate counting to speed up mining, it generates candidates and then

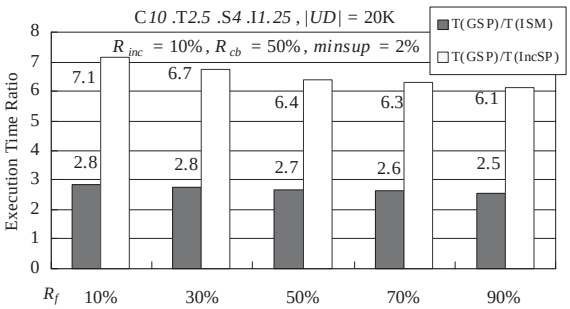


Fig. 12. Total execution times over various former ratios.

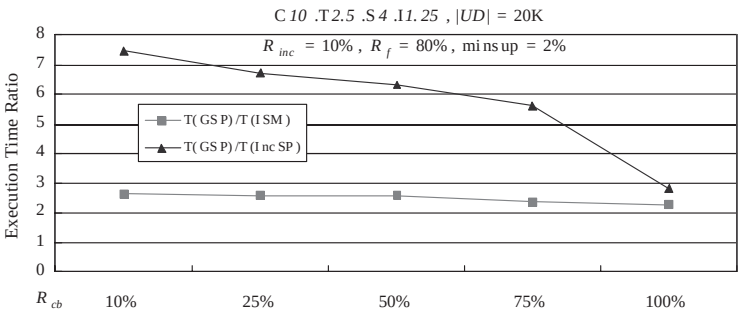


Fig. 11. Total execution times over various comeback ratios.

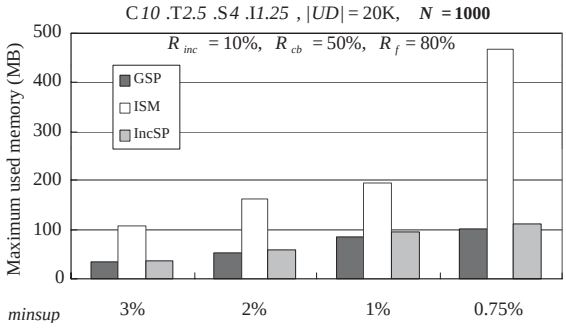


Fig. 13. Linear scalability of the database size.

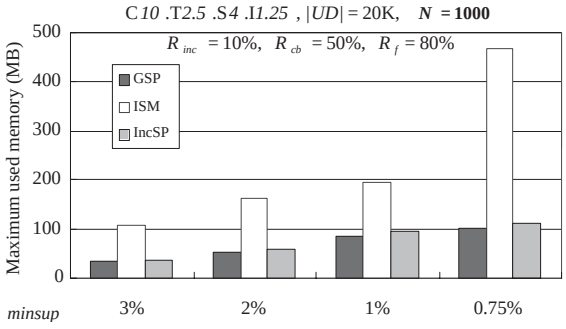


Fig. 14. Maximum required memory with respect to various *minsup*.

performs counting by multiple database scanning, like GSP. The pattern updating process in IncSP reads in the previous discovered patterns and stores them into a hash-tree for fast support updating. Therefore, the maximum size of memory required for both GSP and IncSP is determined by the space required to store the candidates. A smaller *minsup* often generates a large number of candidates, thereby demanding a larger memory space.

In contrast, ISM applies item-intersection in each class for new pattern discovery, assuming that all frequent sequences as well as potentially frequent sequences are stored in a lattice in memory. Storing every possible frequent sequence costs a huge memory space, not to mention those required for lattice links. For instance, the size of negative borders of size two is over 1.5 million with $N = 10,000$ ($minsup = 0.75\%$) in the experiment of Fig. 9b. As shown in Fig. 14, the required memory for IncSP is smaller than that of ISM. More memory is required in vertical approaches like SPADE, which is also observed in [13].

6. Conclusions

The problem of sequential pattern mining is much more complicated than association discovery due to sequence permutation. Validity of discovered patterns may change and new patterns may emerge after updates on databases. In order to keep the sequential patterns current and up-to-dated, re-execution of the mining algorithm on the whole database updated is required. However, it takes more time than required in prior mining because of the additional data sequences appended. Therefore, we proposed the IncSP algorithm utilizing previously discovered knowledge to solve the maintenance problem efficiently by incremental updating without re-mining from scratch. The performance improvements result from effective implicit merging, early candidate pruning, and efficient separate counting.

Implicit merging ensures that IncSP employs correctly combined data sequences while preserving previous knowledge useful for incremental updating. Candidate pruning after updating pattern supports against the increment database further accelerates the whole process, since fewer but more promising candidates are generated by just checking counts in the increment database. Eventually, efficient support counting of promising candidates over the original database accom-

plishes the discovery of new patterns. IncSP both updates the supports of existing patterns and finds out new patterns for the updated database. The simulation performed shows that the proposed incremental updating mechanism is several times faster than re-mining using the GSP algorithm, with respect to various data characteristics or data combinations. IncSP outperforms GSP with regard to different ratios of the increment database to the original database except when the increment database becomes larger than the original database. It means that it has been long time since last database maintenance and most of the patterns become obsolete. In such a case, re-mining with new *minsup* over the whole database would be more appropriate since the original *minsup* might not be suitable for current database any more.

The IncSP algorithm currently solves the pattern updating problems using previously specified minimum support. Further researches could be extended to the problems of dynamically varying minimum supports. Generalized sequential pattern problems [7], such as patterns with *is-a* hierarchy or with sliding-time window property, are also worthy of further investigation since different constraints induce diversified maintenance difficulties. In addition to the maintenance problem, constantly updated database generally create a pattern-changing history, indicating changes of sequential patterns at different time. It is challenging to extend the proposed algorithm to exploring the pattern changing history for trend prediction.

Acknowledgements

The authors thank the referees for their valuable comments and suggestions.

Appendix A

As noted in Table 1, s^{DB}_{count} is the support count of candidate sequence s in DB, and s^{db}_{count} denotes the increase in support count of candidate sequence s due to db. The candidate k-sequences in UD is partitioned into $\boldsymbol{X\kappa}_{(DB)}$ and $\boldsymbol{X\kappa'}_{(DB)}$. That is, $X_k = \boldsymbol{X\kappa}_{(DB)} \cup \boldsymbol{X\kappa}'_{(DB)}$, where $\boldsymbol{X\kappa}_{(DB)} = \{s | s \in X_k \wedge s \in S^{DB}_k\}$ and $\boldsymbol{X\kappa}_{(DB)}{}' = X_k - \boldsymbol{X\kappa}_{(DB)}$. The data sequences in UD could be partitioned into three sets: sequences with *cid*s appearing in DB only, sequences with *cid*s appearing in db only, and sequences with *cid*s occurring in both DB and db. The *cid* of a data sequence ds is represented by $ds.cid$. Let $UD = UD_{DB} \cup UD_{db} \cup UD_{Dd}$, where $UD_{DB} = \{ds | ds \in DB \wedge ds \notin db\}$, $UD_{db} = \{ds | ds \in db \wedge ds \notin DB\}$, and $UD_{Dd} = \{ds | ds = ds_1 + ds_2, ds_1 \in DB \wedge ds_2 \in db \wedge ds_1.cid = ds_2.cid\}$.

Lemma 1. *The support count of any candidate k-sequence s in UD is equal to $s^{DB}_{count} + s^{db}_{count}$.*

Proof. The support count of s in UD is the support count of s in DB, plus the count increase due to the data sequences in db. That is $s^{DB}_{count} + s^{db}_{count}$ by definition. □

Lemma 2. *A candidate sequence s, which is not frequent in DB, is a frequent sequence in UD only if $s^{db}_{count} \geqslant minsup \times (|UD| - |DB|)$.*

Proof. Since $s \notin S^{DB}$, we have $s^{DB}_{count} < minsup \times |DB|$. If $s^{db}_{count} < minsup \times (|UD| - |DB|)$, then $s^{DB}_{count} + s^{db}_{count} < minsup \times |UD|$. That is, $s \notin S^{UD}$. □

Lemma 3. *The separate counting procedure* (*in* Fig. 8) *completely counts the supports of candidate k-sequences against all data sequences in UD.*

Proof. Considering a data sequence ds in UD and a candidate k-sequence $s \in X_k$,

(i) For each candidate k-sequence s contained in ds where $ds \in UD_{db}$: The support count increase (due to ds) is accumulated in s^{db}_{count}, by line 4 of Support Counting (I) in Fig. 8.

(ii) For each candidate k-sequence s contained in ds where $ds \in UD_{DB}$: (a) If $s \in \boldsymbol{X\kappa}_{(DB)}$, no counting is required since s had been counted while discovering S^{DB}. The support count of s in DB is read in s^{DB}_{count} by line 6 in Fig. 7. (b) If $s \in \boldsymbol{X\kappa}_{(DB)}{}'$, s^{DB}_{count} accumulates the support count of s, by line 3 of Support Counting (II) in Fig. 8. Note that in this counting, we reduce $\boldsymbol{X\kappa}_{(DB)}{}'$ to $\boldsymbol{X\kappa}'$ by Lemma 4.

(iii) For each candidate k-sequence s contained in ds where $ds \in UD_{Dd}$: Now ds is formed by appending ds^{db} to ds^{DB}. (a) If $s \not\subset ds^{DB}$, i.e., ds^{DB} of the ds does not contain s. We accumulate the increase in s^{db}_{count}, by line 9 of Support Counting (I) in Fig. 8. (b) If $s \subseteq ds^{DB} \wedge s \in \boldsymbol{X\kappa}_{(DB)}$, similar to (ii)-(a), the support count is already read in s^{DB}_{count} so that no counting is required. (c) If $s \subseteq ds^{DB} \wedge s \in \boldsymbol{X\kappa}_{(DB)}'$, similar to (ii)-(b), we calculate s^{DB}_{count} by line 3 of Support Counting (II) in Fig. 8. Again, $\boldsymbol{X\kappa}_{(DB)}'$ is reduced to $\boldsymbol{X\kappa}'$ by Lemma 4 here.

The separate counting considers all the data sequences in UD as described here. Next, we show that the supports of all candidates are calculated. By Lemma 1, the support count of s in UD is the sum of s^{DB}_{count} and s^{db}_{count}.

(iv) For any candidate s in $\boldsymbol{X\kappa}_{(DB)}$: The s^{DB}_{count} is from (ii)-(a) and (iii)-(b), and the s^{db}_{count} is accumulated by (i) and (iii)-(a).

(v) For any candidate s in $\boldsymbol{X\kappa}_{(DB)}'$: The s^{DB}_{count} is counted by (ii)-(b) and (iii)-(c), and the s^{db}_{count} is counted by (i) and (iii)-(a). The separate counting is complete. □

Lemma 4. *The candidates required for checking against the data sequences in DB in Support Counting (II) is the set* $\boldsymbol{X\kappa}'$, *where* $\boldsymbol{X\kappa}' = X_k - \{s | s \in S^{DB}_k\} - \{s | s^{db}_{count} < minsup \times (|UD| - |DB|)\}$.

Proof. Since $UD = UD_{DB} \cup UD_{db} \cup UD_{Dd}$ and UD_{db} contains no data sequence in DB, the data sequences concerned are in UD_{DB} and UD_{Dd}. Considering a candidate s,

If $s \in S^{DB}_k$: For any data sequence $ds \in UD_{DB}$ or $ds \in UD_{Dd} \wedge s \subseteq ds^{DB}$, s was counted while discovering S^{DB}_k. For $ds \in UD_{Dd} \wedge s \not\subset ds^{DB}$, the increase in support count s^{db}_{count} is accumulated by line 9 of Support Counting (I). Therefore, in Support Counting (II), we can exclude any candidate s which is also in S^{DB}_k.

If $s \in S^{DB}_k$: After Support Counting (I), the s^{db}_{count} now contains the support count counted for data sequence ds, where $ds \in UD_{db}$ or $ds \in UD_{Dd} \wedge s \not\subset ds^{DB}$. By Lemma 2, if the s^{db}_{count} is less than $minsup \times (|UD| - |DB|)$, this candidate s cannot be frequent in UD. Therefore, such candidate s could be filtered out.

By (i) and (ii), we have $\boldsymbol{X\kappa}' = X_k - \{s | s \in S^{DB}_k\} - \{s | s^{db}_{count} < minsup \times (|UD| - |DB|)\}$. □

References

[1] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the 11th International Conference on Data Engineering, 1995, pp. 3–14.

[2] D. Gunopulos, R. Khardon, H. Mannila, H. Toivonen. Data mining, hypergraph transversals, and machine learning, in: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1997, pp. 209–216.

[3] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.C. Hsu, FreeSpan: Frequent pattern-projected sequential pattern mining, in: Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2000, pp. 355–359.

[4] M.Y. Lin, S.Y. Lee, Incremental update on sequential patterns in large databases. in: Proceedings of the 10th IEEE International Conference on Tools with Artificial Intelligence, 1998, pp. 24–31.

[5] S. Parthasarathy, M.J. Zaki, M. Ogihara, S. Dwarkadas, Incremental and interactive sequence mining, in: Proceedings of the 8th International Conference on Information and Knowledge Management, 1999, pp. 251–258.

[6] T. Shintani, M. Kitsuregawa, Mining algorithms for sequential patterns in parallel: Hash based approach, in: Proceedings of the Second Pacific–Asia Conference on Knowledge Discovery and Data Mining, 1998, pp. 283–294.

[7] R. Srikant, R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, in: Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining, 1996, pp. 3–17.

[8] M.J. Zaki, Efficient enumeration of frequent sequences, in: Proceedings of the 7th International Conference on Information and Knowledge Management, 1998, pp. 68–75.

[9] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A.I. Verkamo, Fast discovery of association rules, in: U.M. Fayyad, G. Piatesky-Shapiro, P. Smyth, R. Uthurusamy (Eds.), Advances in Knowledge Discovery and Data Mining, AAAI Press, Menlo Park, CA, 1996, pp. 307–328.

[10] S. Brin, R. Motwani, J. Ullman, S. Tsur, Dynamic itemset counting and implication rule for market basket data, in: Proceedings of the 1997 SIGMOD Conference on Management of Data, 1997, pp. 255–264.

[11] D.W.L. Cheung, J. Han, V. Ng, C.Y. Wong, Maintenance of discovered association rules in large databases: An incremental updating technique, in: Proceedings of the 16th International Conference on Data Engineering, 1996, pp. 106–114.

[12] J.S. Park, M.S. Chen, P.S. Yu, Using a hash-based method with transaction trimming for mining association rules, IEEE Trans. Knowledge Data Eng. 9 (5) (1997) 813–825.
[13] I. Tsoukatos, D. Gunopulos, Efficient mining of spatiotemporal patterns, in: Proceedings of the 7th International Symposium of Advances in Spatial and Temporal Databases, 2001, pp. 425–442.
[14] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, M.C. Hsu, PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth, in: Proceedings of 2001 International Conference on Data Engineering, 2001, pp. 215–224.
[15] S.J. Yen, A.L.P. Chen, An efficient approach to discovering knowledge from large databases, in: Proceedings of the 4th International Conference on Parallel and Distributed Information Systems, 1996, pp. 8–18.
[16] H. Mannila, H. Toivonen, A.I. Verkamo, Discovery of frequent episodes in event sequences, Data Mining Knowledge Discovery 1 (3) (1997) 259–289.
[17] D.W.L. Cheung, S.D. Lee, B. Kao, Ageneral incremental technique for maintaining discovered association rules, in: Proceedings of the 5th International Conference on Database Systems for Advanced Applications, 1997, pp. 185–194.
[18] K. Wang, Discovering patterns from large and dynamic sequential data, J. Intell. Inform. Syst. 9 (1) (1997) 33–56.