Incremental Mining of Sequential Patterns in Large Databases

F. Masseglia [a,b] P. Poncelet [c] M. Teisseire [b]

[a] *Laboratoire PRiSM, Université de Versailles- 45 Av. des Etats-Unis 78035 Versailles Cedex, France*

[b] *LIRMM, 161 rue Ada 34392 Montpellier Cedex 5, France*

[c] *Laboratoire LGI2P - Ecole des Mines d'Als - Site EERIE Parc Scientifique Georges Besse 30035 Nmes Cedex 1, France*

Abstract

In this paper we consider the problem of the incremental mining of sequential patterns when new transactions or new customers are added to an original database. We present a new algorithm for mining frequent sequences that uses information collected during an earlier mining process to cut down the cost of finding new sequential patterns in the updated database. Our test shows that the algorithm performs significantly faster than the naive approach of mining on the whole updated database from scratch. The difference is so pronounced that this algorithm could also be useful for mining sequential patterns, since in many cases it is faster to apply our algorithm than to mine sequential patterns using a standard algorithm, by breaking down the database into an original database plus an increment.

Key words: Sequential patterns, incremental mining, data mining

1 Introduction

Most research into data mining has concentrated on the problem of mining association rules [1–8]. Although sequential patterns are of great practical importance (e.g. alarms in telecommunications networks, identifying plan failures, analysis of Web access databases, etc.) they have received relatively little attention [9–11]. First introduced in [9], where an efficient algorithm called AprioriAll was proposed, the problem of mining sequential patterns is to discover temporal relationships between facts embedded in the database. The facts under consideration are simply the characteristics of individuals, or observations of individual behavior. For example, in a video database, a sequential pattern could be "95% of customers bought 'Star Wars

and The Empire Strikes Back', then 'Return of the Jedi', and then 'The Phantom Menace' ". In [10], the definition of the problem is extended by handling time constraints and taxonomies (*is-a* hierarchies) and a new algorithm, called GSP, which outperformed AprioriAll by up to 20 times, is proposed.

As databases evolve the problem of maintaining sequential patterns over a significantly long period of time becomes essential, since a large number of new records may be added to a database. To reflect the current state of the database where previous sequential patterns would become irrelevant and new sequential patterns might appear, there is a need for efficient algorithms to update, maintain and manage the information discovered [12]. Several efficient algorithms for maintaining association rules have been developed [12–15]. Nevertheless, the problem of maintaining sequential patterns is much more complicated than maintaining association rules, since transaction cutting and sequence permutation have to be taken into account [16]. In order to illustrate the problem, let us consider an original and an incremental database. Then, to compute the set of sequential patterns embedded in the updated database, we have to discover all sequential patterns which were not frequent in the original database but become frequent with the increment. We also have to examine all transactions in the original database that can be extended to become frequent. Furthermore, old frequent sequences may become invalid when a new customer is added. The challenge is thus to discover all the frequent patterns in the updated database with far greater efficiency than the naive method of mining sequential patterns from scratch.

In this paper, we propose an efficient algorithm, called ISE (Incremental Sequence Extraction), for computing the frequent sequences in the updated database when new transactions and new customers are added to the original database. ISE minimizes computational costs by re-using the minimal information from the old frequent sequences, i.e. the support of frequent sequences. The main new feature of ISE is that the set of candidate sequences to be tested is substantially reduced. Furthermore, some optimization techniques for improving the approach are also provided.
Empirical evaluations were carried out to analyze the performance of ISE and compare it against cases where GSP is applied to the updated database from scratch. Experiments showed that ISE significantly outperforms the GSP algorithm by a factor of 4 to 6. Indeed the difference is so pronounced that our algorithm may be useful for mining sequential patterns as well as incremental mining, since in many cases, instead of mining the database with the GSP algorithm, it is faster to extract an increment from the database, then apply our approach considering that the database is broken down into an original database plus an increment. Our experimental results show an improvement in performance by a factor of 2 to 5 in the comparison.

The rest of this paper is organized as follows. Section 2, states the problem and describes related research. The algorithm ISE is described in Section 3. Section 4 describes the experiments in detail and interprets the performance results obtained . Finally, Section 5 concludes the paper with future avenues for research.

2 Statement of the Problem

In this section we give the formal definition of the problem of incremental sequential pattern mining. First, we formulate the concept of sequence mining summarizing the formal description of the problem introduced in [9] and extended in [10]. A brief overview of the GSP algorithm is also provided. Second we examine the incremental update problem in detail.

2.1 Mining of Sequential Patterns

Let *DB* be a set of customer transactions where each transaction T consists of customer-id, transaction time and a set of items involved in the transaction.
Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals called *items*. An *itemset* is a non-empty set of items. A sequence s is a set of itemsets ordered according to their time stamp. It is denoted by $< s_1 s_2 ... s_n >$, where s_j, $j \in 1..n$, is an itemset. A *k-sequence* is a sequence of k items (or of length k). For example, let us consider that a given customer purchased items $1,2,3,4,5$, according to the following sequence:$s =<$ (1) (2, 3) (4) (5)>. This means that apart from 2 and 3 that were purchased together, i.e. during a common transaction, items in the sequence were bought separately. s is a 5-sequence.
A sequence $< s_1 s_2 ... s_n >$ is a sub-sequence of another sequence $< s'_1 s'_2 ... s'_m >$ if there exist integers $i_1 < i_2 < ... i_j ... < i_n$ such that $s_1 \subseteq s'_{i_1}, s_2 \subseteq s'_{i_2}, ..., s_n \subseteq s'_{i_n}$. For example, the sequence $s' = <$ (2) (5) $>$ is a sub-sequence of s because (2) $\subseteq$ (2, 3) and (5) $\subseteq$ (5). However $<$ (2) (3) $>$ is not a sub-sequence of s since items were not bought during the same transaction.

Property 1 *If $A \subseteq B$ for sequences A, B then $supp(A) \geq supp(B)$ because all transactions in DB that support B necessarily also support A.*

All transactions from the same customer are grouped together and sorted in increasing order and are called a *data sequence*. A support value (*supp(s)*) for a sequence gives its number of actual occurrences in *DB*. Nevertheless, a sequence in a data sequence is taken into account only once to compute the support even if several occurrences are discovered. In other words, the support of a sequence is defined as

the fraction of total distinct data sequences that contain s. A data sequence contains a sequence s if s is a sub-sequence of the data sequence. In order to decide whether a sequence is frequent or not, a minimum support value (*minSupp*) is specified by the user, and the sequence is said to be*frequent* if the condition $supp(s) \geq minSupp$ holds.

Given a database of customer transactions the problem of sequential pattern mining is to find all the sequences whose support is greater than a specified threshold (minimum support). Each of these represents a *sequential pattern*, also called a *frequent* sequence.

The task of discovering all the frequent sequences in large databases is quite challenging since the search space is extremely large (e.g. with m attributes there are $O(m^k)$ potentially frequent sequences of length k) [11]. To the best of our knowledge, the problem of mining sequential patterns according to the previous definitions has received relatively little attention.
We shall now briefly review the GSP algorithm. For building up candidate and frequent sequences, the GSP algorithm makes multiple passes over the database. The first step aims at computing the support of each item in the database. When this step has been completed, the frequent items (i.e. those that satisfy the minimum support) have been discovered. They are considered as frequent 1-sequences (sequences having a single itemset, itself a singleton). The set of candidate 2-sequences is built up according to the following assumption: candidate 2-sequences could be any couple of frequent items, whether embedded in the same transaction or not. Frequent 2-sequences are determined by counting the support. From this point, candidate *k*-sequences are generated from frequent (*k-1*)-sequences obtained in pass-(*k-1*). The main idea of candidate generation is to retrieve, from among (*k-1*)-sequences, pairs of sequences (s, s') such that discarding the first element of the former and the last element of the latter results in two fully matching sequences. When such a condition holds for a pair (s, s'), a new candidate sequence is built by appending the last item of s' to s. The supports for these candidates are then computed and those with minimum support become frequent sequences. The process iterates until no more candidate sequences are formed.

2.2 Incremental Mining on Discovered Sequential Patterns

Let *DB* be the original database and *minSupp* the minimum support. Let *db* be the increment database where new transactions or new customers are added to *DB*. We assume that each transaction on *db* has been sorted by customer-id and transaction time. $U = DB \cup db$ is the updated database containing all sequences from *DB* and *db*.

Cust-Id	Itemsets		
$C1$	10 20	20	50 70
$C2$	10 20	30	40
$C3$	10 20	40	30
$C4$	60	90	

(DB)

Itemsets	
50 60 70	*80 100*
50 60	*80 90*

(db)

Fig. 1. An original database (DB) and an increment database with new transactions (db)

Let L^{DB} be the set of frequent sequences in DB. The problem of incremental mining of sequential patterns is to find frequent sequences in U, noted L^U, with respect to the same minimum support. Furthermore, the incremental approach has to take advantage of previously discovered patterns in order to avoid re-running all mining algorithms when the data is updated.

First, we consider the problem when new transactions are appended to customers already existing in the database. In order to illustrate this problem, let us consider the base DB given in Figure 1, giving facts about a population reduced to just four customers. Transactions are ordered according to their time-stamp. For instance, the data sequence of customer $C3$ is $<(10\ 20)\ (40)\ (30)>$. Let us assume that the minimum support value is 50%, which means that in order to be considered as frequent a sequence must be observed for at least two customers. The set of all maximum frequent sequences embedded in the database is the following: $L^{DB} = \{<(10\ 20)\ (30)>, <(10\ 20)\ (40)>\}$. After some update activities, let us consider the increment database db (described in Figure 1) where new transactions are appended to customers $C2$ and $C3$. Assuming that the support value is the same, the following two sequences $<(60)\ (90)>$ and $<(10\ 20)\ (50\ 70)>$ become frequent after the database update since they have sufficient support. Let us consider the first of these. The sequence is not frequent in DB since the minimum support does not hold (it only occurs for the last customer). With the increment database, this sequence becomes frequent since it appears in the data sequences of the customer $C3$ and $C4$. The sequence $<(10\ 20)>$ could be detected for customers $C1$, $C2$ and $C3$ in the original database. By introducing the increment database the new frequent sequence $<(10\ 20)\ (50\ 70)>$ is discovered because it matches with transactions of $C1$ and $C2$. Furthermore, new frequent sequences are discovered: $<(10\ 20)\ (30)\ (50\ 60)\ (80)>$ and $<(10\ 20)\ (40)\ (50\ 60)\ (80)>$. $<(50\ 60)\ (80)>$ is a frequent sequence in db and on scanning DB we find that the frequent sequences in L^{DB} are its predecessor.

Cust-Id	Itemsets (DB)			Itemsets (db)	
$C1$	10 20	20	50 70		
$C2$	10 20	30	40	*50 60 70*	*80 100*
$C3$	10 20	40	30	*50 60*	*80 90*
$C4$	60	90			
$C5$				*10 40*	*70 80*

(DB) (db)

Fig. 2. An original database (DB) and an increment database with new transactions and new customers (db)

Let us now consider the problem when new customers and new transactions are appended to the original database (Figure 2). Let us consider that the minimum support value is still 50%, which means that in order to be considered as frequent a sequence must now be observed for at least three customers since a new customer $C5$ has been added. According to this constraint the set of frequent sequences embedded in the original database becomes $L^{DB} = \{< (10\ 20) >\}$ since the sequences $< (10\ 20)\ (30) >$ and $< (10\ 20)\ (40) >$ occur only for customers $C2$ and $C3$. Nevertheless, the sequence $< (10\ 20) >$ is still frequent since it appears in the data sequences of customer $C1$, $C2$ and $C3$. By introducing the increment database, the set of frequent sequences in the updated database is $L^U = \{< (10\ 20)\ (50) >$, $< (10)\ (70) >$, $< (10)\ (80) >$, $< (40)\ (80) >$, $< (60) >\}$. Let us now take a closer look at the sequence $< (10\ 20)\ (50) >$. This sequence could be detected for customer $C1$ in the original database but it is not a frequent sequence. Nevertheless, as the item 50 becomes frequent with the increment database, this sequence also matches with transactions of $C2$ and $C3$. In the same way, the sequence $< (10)\ (70) >$ becomes frequent since, with the increment, it appears in the data sequences of $C1$, $C2$ and the new customer $C5$.

2.3 Related Work

The problem of incremental association rule mining has been much addressed ([12,13,17–21]), but incremental sequential pattern mining has received very little attention. Furthermore, among the available work in the field, no research has dealt with time constraints or is ready to do so. This section is intended to give two points of view: FASTUP [22] and a SuffixTree approach [23] on the one hand, and ISM [16] on the other.

2.3.1 SuffixTree and FASTUP Approaches

In [23], the authors proposed a solution based on the suffix tree techniques. The structure used in that context acquires the data and builds up the frequent sequences in one scan, by means of a suffix tree. This method is thus very appropriate to incremental sequence extraction, because it only has to continue the data reading after the update. Even though the effectiveness of such a method cannot be denied, its complexity has to be discussed. The complexity in space of the proposed algorithm (as well as that of ISM, described below) depends on the size of the database.

FASTUP, proposed by [22], is an example of the first work done for incremental sequential pattern mining, where complexity in space depends on the size of the result. Indeed, FASTUP stands for an enhanced GSP, taking into account the previous mining result, before generating and validating candidates, using the generating-pruning method.

The main idea is that FASTUP, by means of the previous result, takes advantage of information about sequence thresholdsto generate candidates, . It can therefore avoid generating some sequences, depending on their support.

2.3.2 ISM

The ISM algorithm, proposed by [16], is actually an extension of SPADE [24], which aims at considering the update by means of the negative border and a rewriting of the database.

Figure 3 is an example of a database and its update (items in bold characters). We observe that 3 clients have been updated.

The first iterations of SPADE on *DBspade*, ended in the lattice given in Figure 4 (without the gray section). The main idea of ISM is to keepthe negative border (in grey Fig. 4) NB, which is made of j-candidates, at the bottom of the hierarchy in the lattice. In other words, let s be a sequence in NB, then $\nexists s'/s'$ is child of s and $s' \in NB$, and more precisely NB is made of sequences which are not frequent but being generated by frequent subsequences. We can observe, in Figure 4 the lattice and negative border for *DBspade*. Note that hash lines stand for a hierarchy that does not end in a frequent sequence.

The first step of ISM aims at pruning, the sequences that become infrequent from the set of frequent sequences after the update. One scan of the database is enough to update the lattice as well as the negative border. The second step aims at taking into account the new frequent sequences one by one, in order to make the information browse the lattice using the SPADE generating process. The field of observation

Client	Itemset	Items
1	10	A B
	20	B
	30	A B
	100	**A C**
2	20	A C
	30	A B C
	50	B
3	10	A
	30	B
	40	A
	110	**C**
	120	**B**
4	30	A B
	40	A
	50	B
	140	**C**

Fig. 3. *DBspade*, a database and its update

considered by ISM is thus limited to the new items. For further information you can refer to [16,25].

Example 1 *Let us consider item "C" in DBspade. This item only has a threshold of 1 sequence according to SPADE. After the update given in Figure 3, ISM will consider that support, which is now of 4 sequences. "C" is now going from NB to the set of frequent sequences. In the same way, the sequences* $< (A) (A) (B) >$ *and* $< (A) (B) (B) >$ *become frequent after the update and go from NB to the set of frequent sequences. This is the goal of the first step.*

The second step is intended to consider the generation of candidates, but is limited to the sequences added to the set of frequent sequences during the first step. For instance, sequences $< (A) (A) (B) >$ *and* $< (A) (B) (B) >$ *can generate the candidate* $< (A) (A) (B) (B) >$ *which will have a support of 0 sequences and will be added to the negative border. After the update, the set of frequent sequences will thus be: A, B, C,* $< (A) (A) >$, $< (B) (A) >$, $< (A B) >$, $< (A) (B) >$, $< (B) (B) >$, $< (A) (C) >$, $< (B) (C) >$, $< (A) (A) (B) >$, $< (A B) (B)$

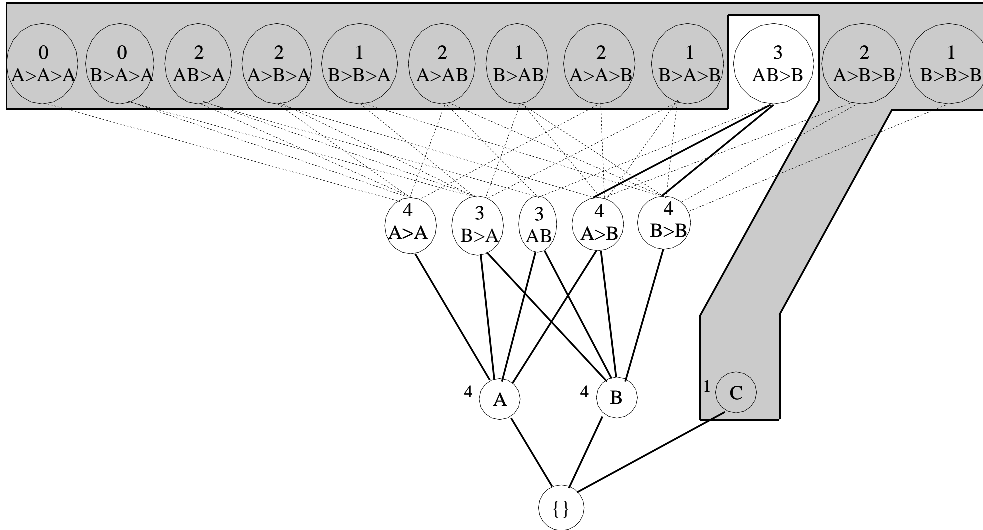


Fig. 4. The negative border, considered by ISM after using SPADE on the database from Figure 3, before the update

$>, < (A)(B)(B) >, < (A)(A)(C) >, < (A)(B)(C) > .$

At the end of the second and last step, the lattice is updated and ISM can give the new set of frequent sequences, as well as a new negative border, allowing the algorithm to take a new update into account. As we observe in Figure 4, the lattice storing the frequent itemsets and the negative border can be very large and memory intensive. Our proposal aims at providing better memory management and at studying candidate generation in order to reduce the number of sequences to be evaluated at each scan of the database.

3 ISE Algorithm

In this section we introduce the ISE algorithm for computing frequent sequences in the updated database. After a brief description of our proposal, we explain, step by step, our method for efficiently mining new frequent sequences using information collected during an earlier mining process. Then we present the associated algorithm and the optimization techniques.

L^{DB}	Frequent sequences in the original database.
L_1^{db}	Frequent 1-sequences embedded in db and validated on U.
$candExt$	Candidate sequences generated from db.
$freqExt$	Frequent sequences obtained from $candExt$ and validated on U.
$freqSeed$	Frequent sub-sequences of L^{DB} extended with an item from L_1^{db}.
$candInc$	Candidate sequences generated by appending sequences of $freqExt$ to sequences of $freqSeed$.
$freqInc$	Frequent sequences obtained from $candInc$ and validated on U.
L^U	Frequent sequences in the updated database.

Table 1
Notation for Algorithm

3.1 An overview

How to solve the problem of incremental mining of frequent sequences by using previously discovered information? To find all new frequent sequences, three kinds of frequent sequences are considered. First, sequences embedded in *DB* could become frequent since they have sufficient support with the incremental database, i.e. sequences similar to sequences embedded in the original database appear in the increment. Next, new frequent sequences embedded in *db* but not appearing in the original database. Finally, sequences of *DB* might become frequent when items from *db* are added.

To discover frequent sequences, the ISE algorithm executes iteratively. in Table 1 we summarize the notation used in the algorithm. Since the main consequence of adding new customers is to verify the support of the frequent sequences in L^{DB}, in the next section we first illustrate iterations through examples mainly concerning added transactions to existing customers. Finally, example 5 illustrates the behavior of ISE when new transactions and new customers are added to the original database.

3.1.1 First Iteration

During the first pass on *db*, we count the support of individual items and we are provided with *1-candExt* standing for the set of items occurring at least once in *db*. Considering the set of items embedded in *DB* we determine which items of *db* are frequent in U. This set is called L_1^{db}.
At the end of this pass, if there are additional customers, we prune out frequent sequences in L^{DB} that no longer verify the minimum support.

Example 1 *Let us consider the increment database in Figure 1. When db is scanned*

we find the support of each individual item during the pass over the data: $\{(<(50)>, 2), (<(60)>, 2), (<(70)>, 1), (<(80)>, 2), (<(90)>, 1), (<(100)>, 1)\}$. *Let us consider that a previous mining of DB provided us with the items embedded in DB with their support:*

item	10	20	30	40	50	60	70	90
support	3	3	2	2	1	1	1	1

Combining these items with the result of the scan db, we obtain the set of frequent 1-sequences which are embedded in db and frequent in U: $L_1^{db} = \{<(50)>, <(60)>, <(70)>, <(80)>, <(90)>\}$.

We use the frequent 1-sequences in *db* to generate new candidates. This candidate generation works by joining L_1^{db} with L_1^{db} and yields the set of candidate 2-sequences. We scan *db* and obtain the 2-sequences embedded in *db*. Such a set is called *2-candExt*. This phase is quite different from the GSP approach since we do not consider the support constraint. We assume, according to Lemma 2 (Cf. Section 3.2), that a candidate 2-sequence is in *2-candExt* if and only if it occurs at least once in *db*. The main reason is that we do not want to provide the set of all 2-sequences, but rather to obtain the set of potential extensions of items embedded in *db*. In other words, if a candidate 2-sequence does not occur in *db* it cannot possibly be an extension of an original frequent sequence of *DB*, and thus cannot give a frequent sequence for U. In the same way, if a candidate 2-sequence occurs in *db*, this sequence might be an extension of previous sequences in *DB*.
Next, we scan U to find out frequent 2-sequences from *2-candExt*. This set is called $freqExt$ and it is achieved by discarding the 2-sequences that do not verify the minimum support from *2-candExt*,.

Example 2 *Let us consider* L_1^{db} *in the previous example. From this set, we can generate the following sequences* $<(50\ 60)>$, $<(50)\ (60)>$, $<(50\ 70)>$, $<(50)\ (70)>$, ..., $<(80)\ (90)>$. *To discover* 2-*candExt in the updated database, we only have to consider if an item occurs at least once in db. For instance, since the candidate* $<(50)\ (60)>$ *does not appear in db, it is no longer considered when U is scanned. After the scan of U with remaining candidates, we are thus provided with the following set of frequent 2-sequences,* $2\text{-}freqExt = \{<(50\ 60)>, <(50)\ (80)>, <(50\ 70)>, <(60)\ (80)>, <(60)\ (90)>\}$.

An additional operation is performed on the frequent items discovered in *db*. Based on Property 1 and Lemma 2 (Cf. Section 3.2) the main idea is to retrieve in *DB* the frequent sub-sequences of L^{DB} preceding items of *db*, according to their order in time.

In order to find the frequent sub-sequences preceding an item efficiently, we create for each frequent sub-sequence an array that has as many elements as the number of frequent items in db. When scanning U, for each data sequence and for each frequent sub-sequence we check whether it is contained in the data sequence. In such a case, the support of each item following the sub-sequence is incremented.

During the scan to find out *2-freqExt*, we also obtain the set of frequent sub-sequences preceding items of db. From this set, by appending the items of db to the frequent sub-sequences we obtain a new set of frequent sequences. This set is called $freqSeed$. In order to illustrate how this new set of frequent sequences is obtained, let us consider the following example.

Items	Frequent sub-sequences
50	$<(10)>_3$ $<(20)>_3$ $<(30)>_2$ $<(40)>_2$ $<(10)\ (30)>_2$ $<(10)\ (40)>_2$ $<(20)\ (30)>_2$ $<(20)\ (40)>_2$ $<(10\ 20)>_3$ $<(10\ 20)\ (30)>_2$ $<(10\ 20)\ (40)>_2$
60	$<(10)>_2$ $<(20)>_2$ $<(30)>_2$ $<(40)>_2$ $<(10)\ (30)>_2$ $<(10)\ (40)>_2$ $<(20)\ (30)>_2$ $<(20)\ (40)>_2$ $<(10\ 20)>_2$ $<(10\ 20)\ (30)>_2$ $<(10\ 20)\ (40)>_2$
70	$<(10)>_2$ $<(20)>_2$ $<(10\ 20)>_2$
80	$<(10)>_2$ $<(20)>_2$ $<(30)>_2$ $<(40)>_2$ $<(10)\ (30)>_2$ $<(10)\ (40)>_2$ $<(20)\ (30)>_2$ $<(20)\ (40)>_2$ $<(10\ 20)>_2$ $<(10\ 20)\ (30)>_2$ $<(10\ 20)\ (40)>_2$
90	-

Fig. 5. Frequent sub-sequences occurring before items of db

Example 3 *Let us consider the item* 50 *in* L_1^{db}. *For customer* C_1, 50 *is preceded by the following frequent sub-sequences:* $<(10)>$, $<(20)>$ *and* $<(10\ 20)>$. *If we now consider customer* C_2 *with the updated transaction, we are provided with the following set of frequent sub-sequences preceding* 50*:* $<(10)>$, $<(20)>$, $<(30)>$, $<(40)>$, $<(10\ 20)>$, $<(10)\ (30)>$, $<(10)\ (40)>$, $<(20)\ (30)>$, $<(20)\ (40)>$, $<(10\ 20)\ (30)>$ *and* $<(10\ 20)\ (40)>$. *The process is repeated until all transactions are examined. In Figure 5 we show the frequent sub-sequences as well as their support in* U.
Let us now examine item 90. *Even if the sequence* $<(60)\ (90)>$ *could be detected for* C_3 *and* C_4, *it is not considered since* 60 *was not frequent in the original database, i.e.* $60 \notin L^{DB}$. *Actually, this sequence is discovered as frequent in*

2-freqExt.
The set $freqSeed$ is obtained by appending to each item of L_1^{db} its associated frequent sub-sequences. For example, if we consider item 70*, then the following sub-sequences are inserted into $freqSeed$:* $<(10)\ (\mathbf{70})>$, $<(20)\ (\mathbf{70})>$ *and* $<(10\ 20)\ (\mathbf{70})>$.

At the end of the first scan on U, we are thus provided with a new set of frequent 2-sequences (in *2-freqExt*) as well as a new set of frequent sequences (in *freqSeed*). In subsequent iterations we go on to discover the all frequent sequences not yet embedded in $freqSeed$ and *2-freqExt*.

3.1.2 j^{th} iteration

Let us assume that we are at the jth pass. In these subsequent iterations, we start by generating new candidates from the two sets found in the previous pass. The main idea of the candidate generation is to retrieve among sequences of $freqSeed$ and *j-freqExt*, two sequences ($s \in freqSeed$, $s' \in j\text{-}freqExt$) such that the last item of s is the first item of s'. When such a condition holds for a pair (s,s'), a new candidate sequence is built by dropping the last item of s and appending s' to the remaining sequence. Furthermore, an additional operation is performed on j-freqExt: we use the same candidate generation algorithm as in GSP to produce new candidate (j+1)-sequences from $j\text{-}freqExt$. Candidates occurring at least once in db, are inserted in the $(j+1)$-*candExt* set. The supports for all candidates are then obtained by scanning U and those with minimum support become frequent sequences. The two sets become respectively $freqInc$ and $(j+1)\text{-}freqExt$. The last one and $freqSeed$ are then used to generate new candidates. The process iterates until all frequent sequences are discovered, i.e. until no more candidates are generated.

For ease of understanding, Fig 6 illustrates, candidate generation at the jth iteration. We can observe that, for the sake of efficiency, each scan aims at counting support for extensions and incremental candidates obtained by means of previously discovered extensions.

In the end, L^U, the set of all frequent sequences, is obtained from L^{DB}, and the maximal sequences from $freqSeed \bigcup freqInc \bigcup freqExt$. At this step, ISE provides all the frequent sequences in the updated database, as shown in Theorem 1.

For ease of understanding, Figure 7 graphically describes the processes in the first and j^{th} iterations.

Example 4 *Considering our example, 3rd iteration, we can thus generate from 2-freqExt a new candidate sequence <(50 60) (80)>. Let us now consider how new candidate sequences are generated from $freqSeed$ and 2-$freqExt$. Let us consider the sequence s =<(20) (40) (50)> from $freqSeed$ and s' =<(50 60)> from 2-$freqExt$. The new candidate sequence <(20) (40) (50 60)> is obtained by drop-*

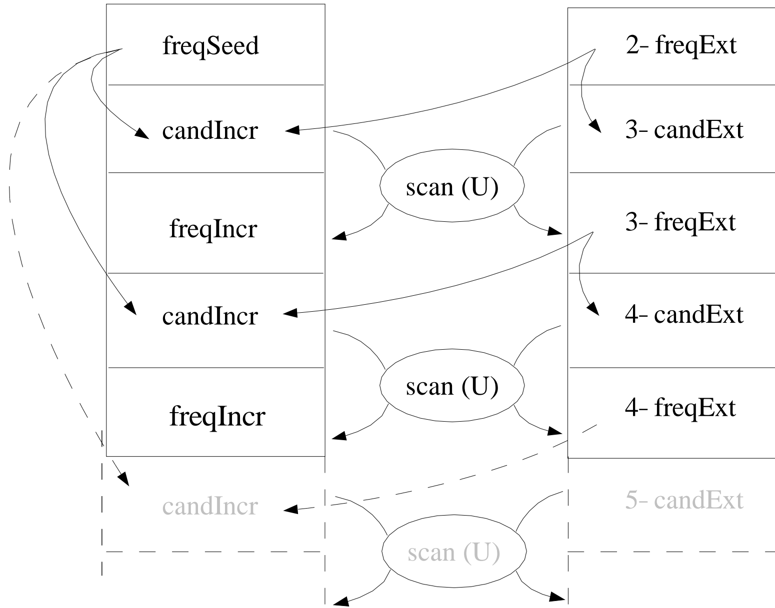


Fig. 6. ISE iterations with j≥2

ping 50 *from s and appending s' to the remaining sequence.*
At the 4th iteration,<(50 60) (80)> is added to 3-freqExt and combined with $freqSeed$, it generates new candidates as example: <(10 20) (30) (50 60) (80)>, <(10 20) (40) (50 60) (80)>, <(20) (40) (50 60) (80)> and <(20) (30) (50 60) (80)>. Nevertheless, there are no more candidates generated from 3-freqExt, and the process ends by verifying the support of the candidates on U. The final maximal frequent sequence set obtained is $L^U = \{$<(60 90)>, <(10 20) (50 70)>, <(10 20) (30) (50 60) (80)>, <(10 20) (40) (50 60) (80)>$\}$.

Now let us examine how new customers are taken into account in the ISE algorithm. As previously described, frequent sequences on the original database may become invalid when adding customer since the support constraint does not hold anymore. The main consequence for the ISE algorithm is to prune out from L^{DB}, the set of sequences that no longer satisfies the support. This is achieved at the beginning of the process. In order to illustrate how such a situation is managed by ISE, let us consider the following example.

Example 5 *Let us now consider Figure 2, where a new customer as well as new transactions are added to the original database. When db is scanned we find the support of each individual item during the pass over the data:* $\{(<(10>,1),(<(40)>,1),(<(50)>,2),(<(60)>,2),(<(70)>,2),(<(80)>,3),(<(90)>,1),(<(100),1)>\}$. *Combining these items with L_1^{DB}, we obtain* $L_1^{db} = \{<(10)>,<(40)>,<(50)>,<(60)>,<(70)>,<(80)>\}$. *As one customer has been added, in order to be frequent a sequence must appear in at least three transactions. Let us now consider L^{DB}. The set L_1^{DB} becomes:* $\{(<(10),4>),(<(20)>$

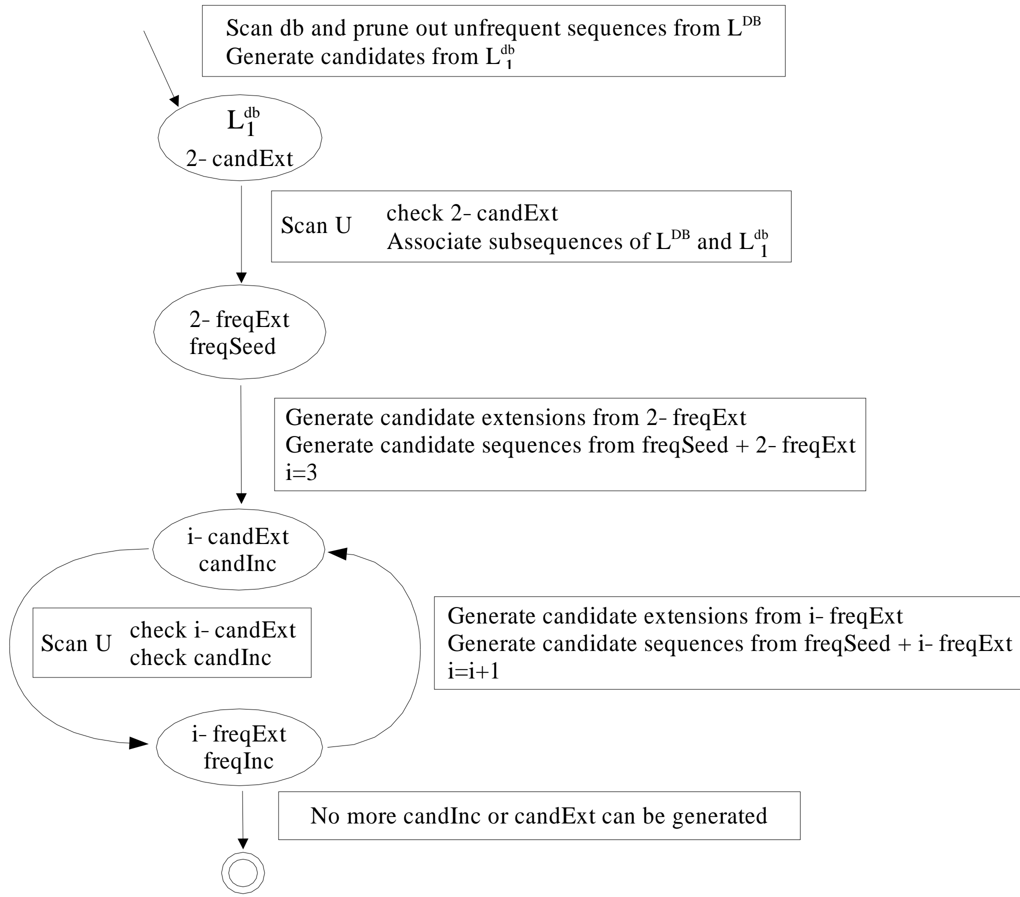


Fig. 7. Processes in the first and i^{th} iterations of ISE

$, 3), (<(40)>, 3)$. *That is to say that item* 30 *is pruned out from* L_1^{DB} *since it is no longer frequent. According to Property 1, the set* L_2^{DB} *is reduced to* $\{(<(10\ 20), 3>)$ *and* L_3^{DB} *is pruned out because the minimum support constraint does not hold anymore. From* L_1^{db}*, we can now generate new candidates in* 2-*candExt*: $\{<(10\ 40)>, <(10)\ (40)>, <(10\ 50)>, \ldots <(70)\ (80)>\}$. *When db is scanned, we prune out candidates not occurring in the increment and are provided with candidate 2-sequences occurring at least once in db. Next we scan U to verify 2-candidates and sequences of the updated* L^{DB} *that chronologically precede sequences of* L_1^{db}. *There are only three candidate sequences that satisfy the support:* 2-*freqExt* $= \{<(10)\ (70)>, <(10)\ (80)><(40)\ (80)\}$. *Let us now have a closer look to frequent sequences occurring before items of* L_1^{db}:

Items	*Frequent Sub-sequences*
10	$<(10)>_0$ $<(20)>_0$ $<(10\ 20)>_0$
40	$<(10)>_2$ $<(20)>_2$ $<(10\ 20)>_2$
50	$<(10)>_3$ $<(20)>_3$ $<(10\ 20)>_3$
60	$<(10)>_2$ $<(20)>_2$ $<(10\ 20)>_2$
70	$<(10)>_3$ $<(20)>_2$ $<(10\ 20)>_2$
80	$<(10)>_3$ $<(20)>_2$ $<(10\ 20)>_2$

The minimum support constraint holds for the following sequences: $freqSeed = \{<(10)\ (70)>, <(10)\ (80)>\ <(10\ 20)\ (50)>\}$ *(The sequences* $<(10)\ (70)>$ *and* $<(10)\ (80)>$ *are also in* $2\text{-}freqExt$*, this is a particular case addressed in section 3.3). Since, we cannot generate new candidates from* $freqSeed$ *and* $2\text{-}freqExt$*, the process completes and all maximal frequent sequences are stored in* $L^U = \{<(10\ 20)\ (50)>, <(10)\ (70)>, <(10)\ (80)>, <(40)\ (80)>, <(60)>\}$.

3.2 The ISE *Algorithm*

Building on the above discussion, we shall now describe the ISE algorithm.

Algorithm ISE
Input: *DB* the original database, L^{DB} the set of frequent sequences in *DB*, the support of each item embedded in *DB*, *db* the increment database, *minSupp* the minimum support threshold and *k* the size of the maximal sequences in L^{DB}.
Output: The set L^U of all frequent sequences in $U = DB \cup db$
Method:

//First Iteration
$L_1^{db} \leftarrow \emptyset$
foreach $i \in db$ **do**
if $(support_{DB \bigcup db}(i) \geq minSupp)$ **then** $L_1^{db} \leftarrow L_1^{db} \bigcup \{i\}$;
enddo

Prune out from L^{DB} sequences no more verifying the minimum support;

$2\text{-}candExt \leftarrow$ generate candidate 2-sequences by joining L_1^{db} with L_1^{db};

// find sequences occurring in db
Scan *db* for 2-*candExt*;
Generate from L^{DB}, the set of frequent sub-sequences;
Scan U to validate candidate 2-candExt and frequent sub-sequences occurring before
items of L_1^{db};
$freqSeed \leftarrow$ frequent sub-sequences occurring before items of L_1^{db} and appended with the item;
2-$freqExt \leftarrow$ frequent sequences from 2-candExt;
// j^{th} Iteration
j=2;
While $(j\text{-}freqExt != \emptyset)$ **do**
$candInc \leftarrow$ generate candidates from $freqSeed$ and j-$freqExt$;
j++;
j-$candExt \leftarrow$ Generate candidate
j-sequences from j-$freqExt$;

// find sequences occurring in db
Scan *db* for j-*candExt*;
if ($j\text{-}candExt != \emptyset$ OR $candInc != \emptyset$)**then**
Scan U for j-*candExt* and *candInc*;
endif
$j\text{-}freqExt \leftarrow$ frequent j-sequences;
$freqInc \leftarrow freqInc$ + candidates from $candInc$ verifying the support on U;
enddo
$L^U \leftarrow L^{DB} \bigcup \{$maximal frequent sequences in $freqSeed \bigcup freqInc \bigcup freqExt\}$;
end Algorithm ISE

To prove that ISE provides the set of frequent sequences embedded in U, we first show in the following two lemmas that every new frequent sequence can be written as the composition of two sub-sequences. The former is a frequent sequence in the original database while the latter occurs at least once in the updated data.

Lemma 1 *Let F be a frequent sequence on U such that F does not appear in L^{DB}. Then F is such that its last itemset occurs at least once in db.*

Proof:

- case $|F| = 1$: Since $F \notin L^{DB}$, F contains an itemset occurring at least once in db, thus F ends with a single itemset occurring at least once in db.
- case $|F| > 1$: F can be written as $<<A><B>>$ with A and B sequences such that $0 \leq |A| < |F|$, $0 < |B| \leq |F|$, $|A| + |B| = |F|$ with $B \notin db$. Let M_B be the set of all data sequences containing B. Let M_{AB} be the set of all data sequences

containing F. We know that if $|M_B| = n$ and $|M_{AB}| = m$ then $minSupp \leq m \leq n$ (according to Property 1). Furthermore $M_{AB} \in DB$ (since $B \notin db$ and transactions are ordered by time) then $<< A >< B >>$ is frequent on DB, this implies $F \in L^{DB}$ which contradicts assumption $F \notin L^{DB}$. Thus, if a frequent sequence F does not appear in L^{DB}, F ends with an itemset occurring at least once in db $\square$

Lemma 2 *Let F be a frequent sequence on U such that F does not appear in L^{DB}. F can thus be written as $<< D >< S >>$, where D and S are two sequences, $|D| \geq 0$, $|S| \geq 1$, such that S is the maximal sub-sequence occurring at least once in db and D is included in (or is) a frequent sequence from L^{DB}.*

Proof:

- case $|S| = |F|$: thus $|D| = 0$ and $D \in L^{DB}$.
- case $1 \leq |S| < |F|$: that is, $D =< (i_1)(i_2)..(i_{j-1}) >$ and $S =< (i_j)..(i_t) >$ where S is the maximal sub-sequence ending F and occurring at least once in db (from Lemma 1 we know that $|S| \geq 1$). Let M_D be the set of all data sequences containing D. Let M_F be the set of all data sequences containing F. We know that if $|M_D| = n$ and $|M_F| = m$ then $minSupp \leq m \leq n$ (according to Property 1). Furthermore, $M_D \in DB$ (since by assumption $i_{j-1} \notin db$ and transactions are ordered chronologically). Thus $D \in L^{DB}$ $\square$

Considering a new frequent sequence, we show that it can be written as two subsequences such that the latter is generated as a candidate extension by ISE.

Lemma 3 *Let F be a frequent sequence on U such that F does not appear in L^{DB}. F can be written as $<< D >< S >>$ where D and S are two sequences verifying $|D| \geq 0$ and $|S| \geq 1$, S is the maximal sub-sequence occurring at least once in db, D is included in (or is) a frequent sequence from L^{DB} and S is included in candExt.*

Proof: Thanks to Lemma 2, we only have to show that S occurs in *candExt*.

- case S is a one transaction sequence, reduced to a single item: S is thus found at the first scan on db and added to 1-*candExt*.
- case S contains more than one item: *candExt* is built up 'a la GSP' from all frequent items in db and is thus a superset of all frequent sequences on U occurring in db $\square$

The following Theorem guarantees the correctness of the ISE approach.

Theorem 1 *Let F be a frequent sequence on U such that F does not appear in L^{DB} and $|F| \leq k+1$. Then F is generated as a candidate by* ISE.

Proof: From Lemma 2 let us consider different possibilities for S.

- case $S = F$: Thus S will be generated in *candExt* (Lemma 3) and added to

$freqExt$.

- cases $S \neq F$:
  - case S is a one transaction sequence, reduced to a single item i: Thus $< < D > < (i) > >$ will be considered in the association made by $freqSeed$.
  - case S contains more than one item: Let us consider i_{1_1} the first item from the first itemset of S. i_{1_1} is frequent on db, thus $< < D > < (i_{1_1}) > >$ is generated in $freqSeed$. According to Lemma 3, S occurs in $freqExt$ and will be used by ISE to build $< < D > < S > >$ in *candInc* □

3.3 Optimizations

In order to speed up the performance of the ISE algorithm we consider two optimization techniques for generating candidates.

As the speed of algorithms for mining association rules, as well as sequential patterns, depends very much on the size of the candidate set, we first improve performance by using information on items embedded in L^{db}, i.e. frequent items in db. The optimization is based on the following lemma:

Lemma 4 *Let us consider two sequences ($s \in freqSeed, s' \in freqExt$) such that an item $i \in L_1^{db}$ is the last item of s and the first item of s'. If there exists an item $j \in L_1^{db}$ such that j is in s' and j is not associated to s in $freqSeed$, the sequence obtained by appending s' to s is not frequent.*

Proof: If s is not followed by j in $freqSeed$, then $< s\ j >$ is not frequent. Hence $< s\ s' >$ is not frequent since there exists an infrequent sub-sequence of $< s\ s' >$.

Using this lemma, at the jth iteration, with $j \geq 2$, we can reduce the number of candidates significantly by avoiding the generation of $< s\ s' >$ as a candidate. In our experiments, the number of candidates was reduced by nearly 40%. The only additional cost is to find out whether there is a frequent sub-sequence matching the first one for each item occurring in the second sequence. As we are provided with an array that stores the items occurring after the sequence for each frequent sub-sequence, the additional cost of this optimization is relatively low.

In order to illustrate this optimization, let us consider the following example.

Example 6 *Let us consider the frequent sequence $s \in 2$-$freqExt$ such as s =<(50 70)>. We have found in $freqSeed$ the following frequent sequence <(10) (30) (50)>. According to the previous generation phase, we would generate <(10) (30) (50 70)>. Nevertheless, the sequence <(10) (30)> is never followed by* 70*. So, we can conclude that <(10) (30) (70)> is not frequent. This sequence is a sub-*

$\|D\|$	Number of customers (size of Database)
$\|C\|$	Average number of transactions per Customer
$\|T\|$	Average number of items per Transaction
$\|S\|$	Average length of maximal potentially large Sequences
$\|I\|$	Average size of Itemsets in maximal potentially large sequences
N_S	Number of maximal potentially large Sequences
N_I	Number of maximal potentially large Itemsets
N	Number of items
I^-	Average number of itemsets removed from sequences in U to build db
$D^{\%}$	Percentage of updated transactions in U
$C^{\%}$	Percentage of customers removed from U in order to build db

Table 2
Parameters

sequence of <(10) (30) (50 70)>, thus before generating we know that <(10) (30) (50 70)> is not frequent. Hence, this last sequence is not generated.

The main concern of the second optimization is to avoid generating candidate sequences that have already been found to be frequent in a previous phase. In fact, when generating a new candidate by appending a sequence of $freqExt$ to a sequence of $freqSeed$ we first test if this candidate was not already discovered frequent. In this case the candidate is no longer considered. To illustrate, consider <(30) (40)> to be a frequent sequence in 2-$freqExt$. Let us now assume that <(10 20) (30) (40)> and<(10 20) (30)> are frequent in $freqSeed$. From the last sequence the generation would provide the following candidate $<$ (10 20) (30) (40) $>$ which was already found frequent. This optimization reduces the number of candidates before U is scanned at negligible cost.

4 Experiments

In this section, we present the performance results of our ISE algorithm and the GSP algorithm. All experiments were performed on a PC Station with a CPU clock rate of 450 MHz, 64MB of main memory, a Linux System and a 9GB disk drive (IDE).

Name	—C—	—I—	N	—D—	Size (Mo)
C9-I4-N1K-D50K	9	4	1,000	50,000	12
C9-I4-N2K-D100K	9	4	2,000	100,000	30
C12-I2-N2K-D100K	12	2	2,000	100,000	30
C12-I4-N1K-D50K	12	4	1,000	50,000	18
C13-I3-N20K-D500K	13	3	20,000	500,000	230
C15-I4-N30K-D600K	15	4	30,000	600,000	320
C20-I4-N2K-D800K	20	4	2,000	800,000	460

Table 3
Parameter values for synthetic datasets

4.1 Datasets

We used synthetic datasets to study the algorithm performance. The synthetic datasets were first generated using the same techniques as introduced in [10] [1] . The generation of *DB* and *db* was performed as follows. As we wanted to model real life updates very accurately, as in [12], we first generated all the transactions from the same statistical pattern, then databases of size $|U|=|DB+db|$ were generated.
In order to assess the relative performance of ISE when new transactions were appended to customers already existing in *DB*, we removed itemsets from the database U using the user defined parameter I^-. The number of transactions which were modified was provided by the parameter $D^{\%}$ standing for the percentage of transactions modified. The transactions embedding removed itemsets were randomly chosen according to $D^{\%}$. Finally, removed transactions were stored in the increment database *db* while remaining transactions were stored in the database *DB*. In the same way, in order to investigate the behavior of ISE when new customers were added, the number of customers removed from U was provided by the parameter $C^{\%}$.
Table 2 lists the parameters used in the data generation method and Table 3 shows the databases used and their properties. For experiments we first investigated the behavior of ISE when new transactions were added. For these experiments, I^- was set to 4 and $D^{\%}$ was set to 90%. Finally, to study the performance of our algorithm with new customers, $C^{\%}$ was set to 10% and 5%.

[1] The synthetic data generation program is available at the following URL (*http://www.almaden.ibm.com/cs/quest).*

4.2 *Comparison of* ISE *with GSP*

In this section, we compare the naive approach, i.e. using GSP for mining the updated database from scratch, and our incremental algorithm. We also test how it scales up as the number of transactions increases. Finally, we carried out experiments to analyze the performance of the ISE algorithm according to the size of updates.

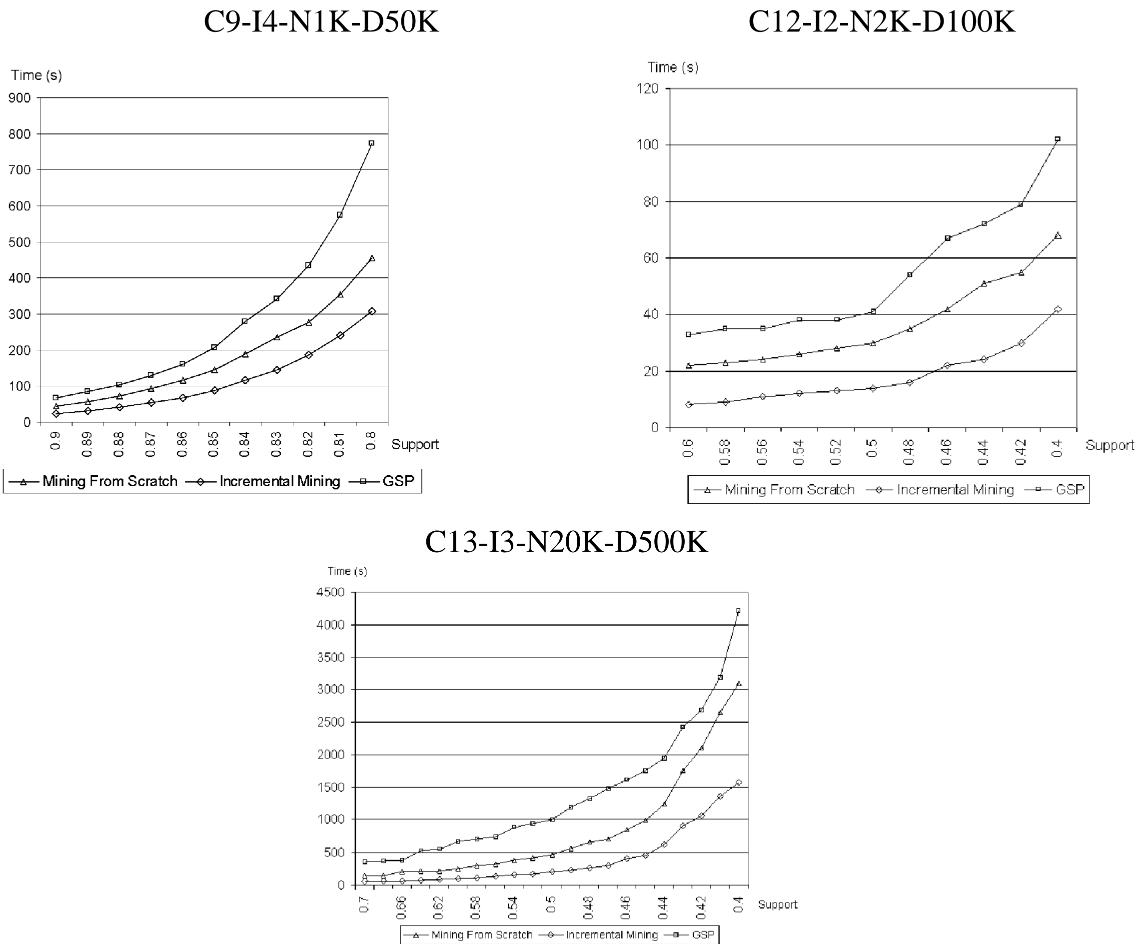


Fig. 8. Execution times

4.2.1 *Naive vs.* ISE *algorithm*

Figure 8 shows experiments conducted on the different datasets using different minimum support ranges to get meaningful response times. Note the minsupport thresholds are adjusted to be as low as possible while retaining reasonable execution times. The label "Incremental Mining" corresponds to the ISE algorithm while "GSP" stands for GSP used for mining the updated database from scratch. "Mining from scratch" corresponds to ISE for mining sequential patterns, i.e. assuming that

no previous mining has been performed.

Figure 8 clearly indicates that the performance gap between the two algorithms increases with decreasing minimum support. We can observe that ISE is 3.5 to 4 times faster than running GSP from scratch. It can also be noticed that ISE outperforms GSP for small support as well as large support value: ISE is still 2.5 to 3 times faster for large support. The same results are found even if the number of itemsets is large. For instance, the last graph in Figure 8 reports an experiment conducted for investigating the effect of the number of itemsets on the performance. When the support is lower the GSP algorithm provides the worst performance.
In Section 4.3.1, we shall investigate the correlation between execution times and the number of candidates.

4.2.2 Performance in scaled-up databases

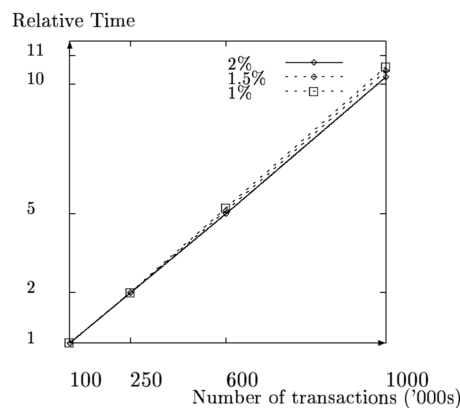


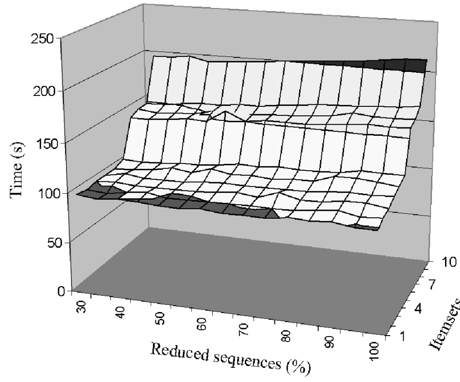
Fig. 9. Scale-up: Number of total transactions

We examined how ISE behaves as the number of total transactions is increased. We would expect the algorithm to have almost linear scale-up. This is confirmed by figure 9 which shows that ISE scales up linearly as the number of transactions is increased ten-fold, from 0.1 million to 1 million. Experiments were performed on the C12-I4-N1K-D50K dataset with three levels of minimum support (2%, 1.5% and 1%). During our evaluation, the size of the increment database was always proportional ($D^{\%} = 90\%$ and $I^{-} = 4$) to the number of new added transactions. The execution times are normalized with respect to the time for the 0.1 million dataset.

4.2.3 Varying the size of added transactions

We carried out some experiments to analyze the performance of the ISE algorithm according to the size of updates. We used the databases C13-I3-N20K-D500K and C12-I2-N2K-D100K for experiments with a threshold of respectively 0.6% and

C13-I3-N20K-D500K C12-I2-N2K-D100K



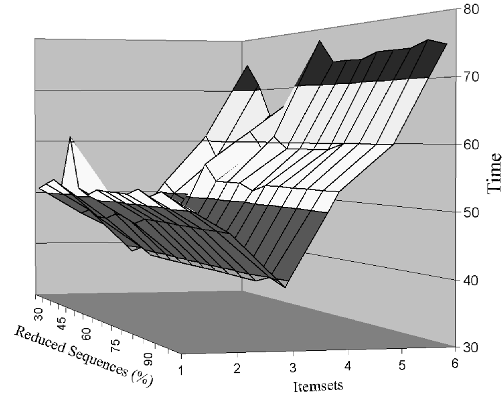


Fig. 10. Size of updates

0.4%. From these databases, we investigated the performance of ISE the number of itemsets removed from the generated database was varied, together with the number of clients. Deleted transactions were stored in the increment database *db* while the remaining transactions were stored in the *DB* database. We first ran GSP to mine L^{DB} and then ran ISE on the updated database. Figure 10 shows the result of this experiment when considering the time for ISE.

For the first one, we can observe that ISE is very efficient from 1 to 6 itemsets removed. The frequent sequences in L^U are obtained in less than 110 seconds. As the number of removed transactions increases, the amount of time taken by ISE increases. For instance, when 10 itemsets are deleted from the original database, ISE takes 180 seconds for 30% of transactions to 215 seconds if the items were deleted from all the transactions. The main reason is that the changes to the original database are so numerous that the results obtained during an earlier mining are not helpful. Interestingly, we also noticed that the time taken by the algorithm does not depend very much on the number of transactions updated.

Let us consider the second surface, the algorithm takes more and more time as the number of itemsets removed grows. Nevertheless, when 3 itemsets are removed from the generated database, ISE takes only 30 seconds to discover the set of all sequential patterns.

4.2.4 Varying the number of added customers

We assume, since it is realistic and suited to real applications, that the average size of the added sequences is less than or equal to the average size of the sequences embedded in the original database. Intuitively, an obvious approach would be to study the behavior of ISE when the number of new customers added to the database is increased. In fact, this nave idea could not be a significant indicator. This is because

when the number of new customers increases, then the number of occurrences of a sequence must also be increased to satisfy the minimum support constraint. Obviously, as at the beginning of the ISE algorithm, we prune out from L^{DB} frequent sequences that no longer satisfy the support, so the more of cutomers are added, the more of previous frequent sequences are pruned out. The main consequence is that the number of frequent sequences decreases, together with the execution times.
A much more interesting approach for evaluating ISE performance is to carry out experiments comparing execution times of GSP vs. ISE on different datasets while varying the minimum support. Figure 11 shows experiments conducted on two datasets C9-I4-N2K-D100K and C20-I4-N2K-D800K where 10% and 5% of customers have been added respectively. We can observe that ISE is very efficient and even when customers are added it is nearly twice as fast as applying GSP from scratch.

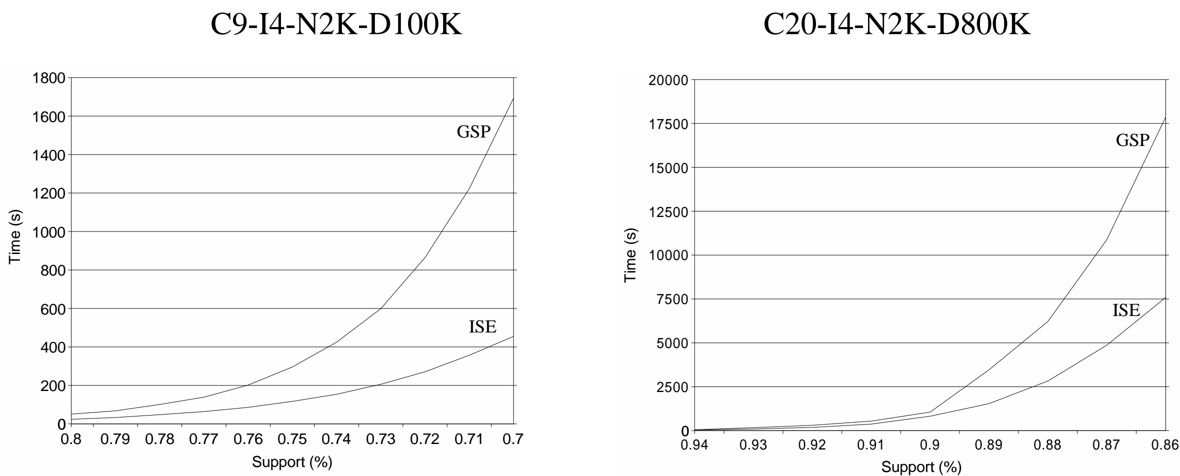


Fig. 11. Execution times when 10% and 5% of customers are added to the original database

4.3 ISE *for Mining Sequential Patterns*

In this section we investigate the performance of ISE for mining sequential patterns.

We designed some experiments to analyze the performance of ISE when mining sequential patterns using the same datasets as in Section 4.2.1. Nevertheless we performed the following operation on each dataset. First we removed 6 items to 60[l22]% of transactions in order to provide the increment database. Second we ran GSP to mine the k-frequent sequences in DB. Finally we ran ISE. In other words, the graphs in Figure 8 show two behaviors. The graph labeled "GSP" indicates the time response of GSP on U, whereas the "ISE" graph shows GSP on DB plus ISE on db. We observe that ISE is from 1.7 to 3 times faster than GSP for mining sequential patterns. The main reason for the gain in performance is the reduced

number of candidates. We study this effect in the next section. As expected, we also observe that using ISE for incremental mining instead of mining from scratch is still efficient since the incremental mining is nearly twice as faster as mining from scratch with these data sets.

4.3.1 Candidate Sets

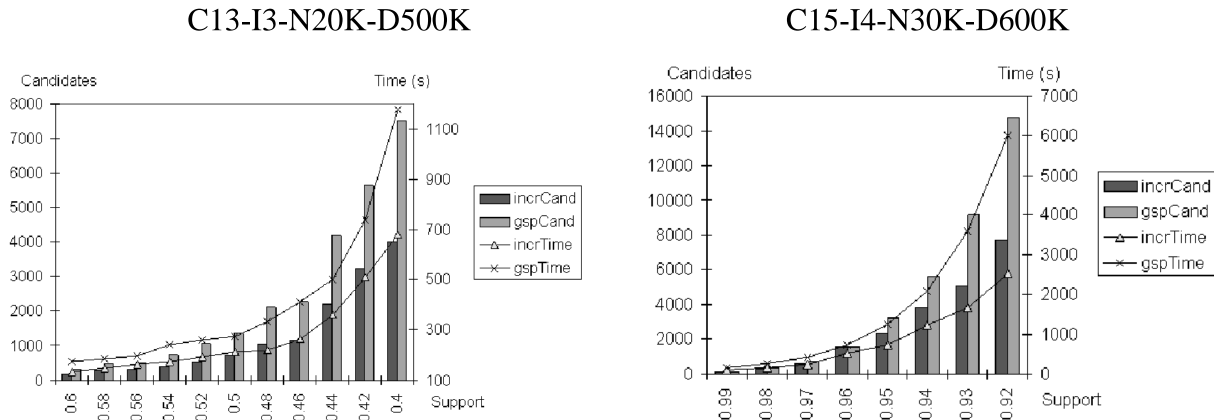


Fig. 12. Candidate sets

In order to explain the correlation between the number of candidates and the execution times we compared the number of candidate sets generated by GSP and our algorithm. Results are depicted in Figure 12. As we can see, the number of candidates for GSP is nearly twice the number for ISE. Let us have a closer look at low support. In the first graph, GSP generates more than 7000 candidates while ISE generates only 4000 candidates. The same result is obtained in the second graph, where GSP generates more than 14000 candidates while our algorithm generates 8000.

4.3.2 Varying the size of updates

Finally, we carried out some experiments in order to analyze the performance of the ISE algorithm with respect to the size of updates. Experiments were conducted on datasets C12-I2-N2K-D100K and C13-I3-N20K-D500K with a threshold of 0.4%. Let us consider the first surface in Figure 13. The best results are obtained when 5 itemsets are deleted from the database. All frequent sequences are then obtained in less than 17 seconds. The algorithm is still very efficient with from 2 to 7 itemsets deleted but when 5 itemsets are deleted from 10 % of customers, ISE is less efficient.
In the second surface of Figure 13, the performance of ISE is quite similar and best results are obtained when 9 itemsets are removed from 80% of customers.

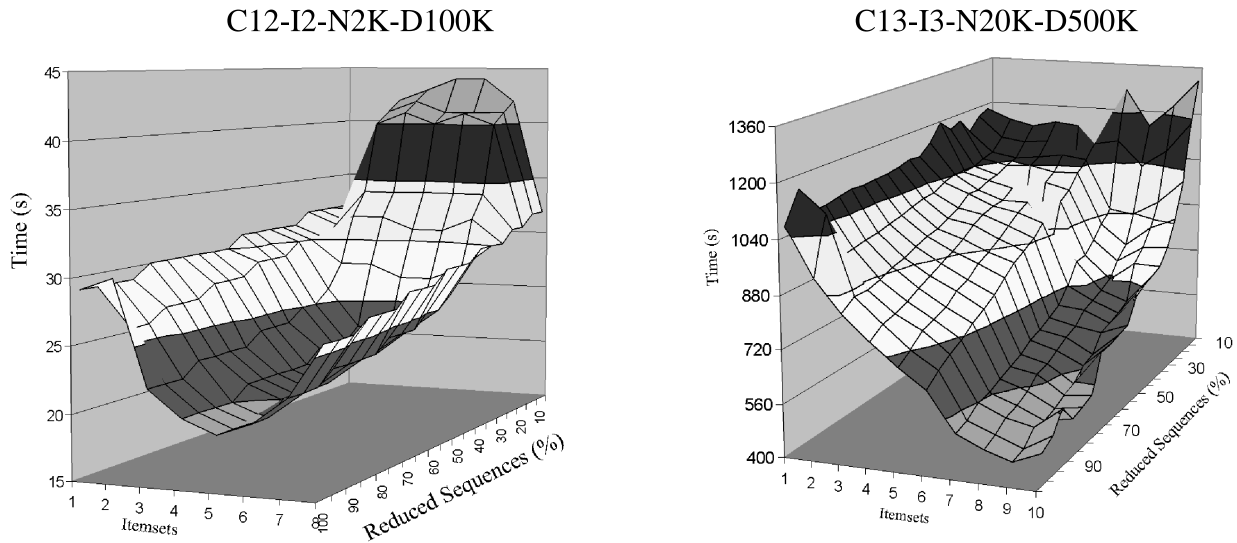


Fig. 13. Size of updates

5 Conclusion

In this paper we present the ISE approach for the incremental mining of sequential patterns in large databases. This method is based on the discovery of frequent sequences by only considering frequent sequences obtained by an earlier mining step. By proposing an iterative approach based only on such frequent sequences we are able to handle large databases without having to maintain negative border information, which was proved to be very memory consuming [16]. Maintaining such a border is well adapted to incremental association mining [26,19], where association rules are only intended to discover intra-transaction patterns (itemsets). Nevertheless, in sequence mining, we also have to discover inter-transaction patterns (sequences) and the set of all frequent sequences is an unbounded superset of the set of frequent itemsets (bounded) [16]. The main consequence is that such approaches are very limited by the negative border size.
Our performance results show that the ISE method is very efficient since it performs much better than re-run discovery algorithms when data is updated. We found by means of empirical evaluations that the proposed approach was so efficient that it was quicker to extract an increment from the original database then apply ISE to mine sequential patterns than to use the GSP algorithm. Experiments on incremental web usage mining were also performed, for further information refer to [27].

There are various avenues for future work on incremental mining. Firstly, while the incremental approach is applicable to databases, which are frequently updated when new transactions or new customers are added to an original database, it also appropriate to many other fields. For example, both electronic commerce and web usage mining require deletion or modification to be taken into account in order to

save storage space or because information is no longer of interest or has become invalid. We are currently investigating how to manage these operations in the ISE algorithm.

Second, we are currently studying how to improve the overall process of incremental mining. By means of experimentation, we would like to discover measures that can suggest to us when ISE should be applied to find out the new frequent sequences in the updated database. Such an approach has been proposed in another context, [28], based on a sampling technique in order to estimate the difference between old and new association rules. We are currently investigating whether other measures could be found by analyzing the data distribution of the original database.

A recaser

In [16], the authors propose an incremental mining algorithm, based on the SPADE approach [11], which can update the sequential patterns in a database when new transactions and new customers are added. It is based on an increment sequence lattice consisting of all frequent sequences and all sequences in the negative border of the original database. This negative border is the collection of all sequences that are not frequent but whose generating sub-sequences are both frequent. Furthermore, the support of each member is also retained in the lattice. The main idea of this algorithm is that when incremental data arrives the incremental part is scanned once to incorporate the new information in the lattice. Then new data is combined with the frequent sequences and negative border in order to determine the portions of the original database that need to be re-scanned. Even though this approach is very effective, maintaining the negative border is very memory consuming and not appropriate for very large databases [16].

References

[1] R. Agrawal, T. Imielinski, A. Swami, Mining Association Rules between Sets of Items in Large Databases, in: Proceedings of the 1993 ACM SIGMOD Conference, Washington DC, USA, 1993, pp. 207–216.

[2] R. Agrawal, R. Srikant, Fast Algorithms for Mining Generalized Association Rules, in: Proceedings of the 20th International Conference on Very Large Databases (VLDB'94), Santiago, Chile, 1994.

[3] S. Brin, R. Motwani, J. Ullman, S. Tsur, Dynamic Itemset Counting and Implication Rules for Market Basket Data, in: Proceedings of the International Conference on Management of Data (SIGMOD'97), Tucson, Arizona, 1997, pp. 255–264.

[4] U. Fayad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy (Eds.), Advances in Knowledge Discovery and Data Mining, AAAI Press, Menlo Park, CA, 1996.

[5] G. Gardarin, P. Pucheral, F. Wu, Bitmap Based Algorithms For Mining Association

Rules, in: Actes des journées Bases de Données Avancées (BDA'98), Hammamet, Tunisie, 1998.

[6] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Efficient Mining of Association Rules Using Closed Itemset Lattices, Information Systems 19 (4) (1998) 33–54.

[7] A. Savasere, E. Omiecinski, S. Navathe, An Efficient Algorithm for Mining Association Rules in Large Databases, in: Proceedings of the 21 st International Conference on Very Large Databases (VLDB'95), Zurich, Switzerland, 1995, pp. 432–444.

[8] H. Toivonen, Sampling Large Databases for Association Rules, in: Proceedings of the 22nd International Conference on Very Large Databases (VLDB'96), 1996.

[9] R. Agrawal, R. Srikant, Mining Sequential Patterns, in: Proceedings of the 11th International Conference on Data Engineering (ICDE'95), Tapei, Taiwan, 1995.

[10] R. Srikant, R. Agrawal, Mining Sequential Patterns: Generalizations and Performance Improvements, in: Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96), Avignon, France, 1996, pp. 3–17.

[11] M. Zaki, Scalable Data Mining for Rules, Tech. Rep. PHD Dissertation, University of Rochester - New York (1998).

[12] D. Cheung, J. Han, V. Ng, C. Wong, Maintenance of Discovered Association Rules in Large Databases: An Incremental Update Technique, in: Proceedings of the 12th International Conference on DataEngineering (ICDE'96), New-Orleans, Louisiana, 1996.

[13] D. Cheung, S. Lee, B. Kao, A General Incremental Technique for Maintening Discovered Association Rules, in: Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFA'97), Melbourne, Australia, 1997.

[14] V. Pudi, J. Haritsa, Quantifying the Utility of the Past in Mining Large Databases, Information Systems 25 (5) (2000) 323–344.

[15] K. Wang, Discovering Patterns from Large and Dynamic Sequential Data, Journal of Intelligent Information System (1997) 8–33.

[16] S. Parthasarathy, M. Zaki, M. Ogihara, S. Dwarkadas, Incremental and Interactive Sequence Mining, in: Proceedings of the 8th International Conference on Information and Knowledge Management (CIKM'99), Kansas City, MO, USA, 1999, pp. 251–258.

[17] R. Agrawal, G. Psaila, Active Data Mining, in: Proceedings of the 1st International Conference on Knowledge Discovery in Databases and Data Mining, 1995.

[18] N. Sarda, N. V. Srinivas, An Adaptive Algorithm for Incremental Mining of Association Rules, in: Proceedings of the 9th International Workshop on Database and Expert Systems Applications, Indian Institute of Technology Bombay, 1998.

[19] S. Thomas, S. Bodagala, K. Alsabti, S. Ranka, An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases, in: Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD '97), Newport Beach, California, 1997.

[20] C. P. Rainsford, M. K. Mohania, J. F. Roddick, Incremental Maintenance Techniques for Discovered Classification Rules, in: Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications, Kyoto, Japan, 1996, pp. 302–305.

[21] C. P. Rainsford, M. K. Mohania, J. F. Roddick, A Temporal Windowing Approach to the Incremental Maintenance of Association Rules, in: Proceedings of the Eighth International Database Workshop, Data Mining, Data Warehousing and Client/Server Databases (IDW'97), Hong Kong. Fong,, 1997, pp. 78–94.

[22] M. Lin, S. Lee, Incremental Update on Sequential Patterns in Large Databases, in: Proceedings of the Tools for Artificial Intelligence Conference (TAI'98), 1998, pp. 24–31.

[23] K. Wang, J. Tan, Incremental Discovery of Sequential Patterns, in: Proceedings of ACM SIGMOD'96 Data Mining Workshop, Montréal, Canada, 1996, pp. 95–102.

[24] M. J. Zaki, Fast Mining of Sequential Patterns in Very Large Databases, Tech. rep., The University of Rochester, New York 14627 (1999).

[25] F. Masseglia, P. Poncelet, M. Teisseire, Incremental Mining of Sequential Patterns in Large Databases, in: Actes des Journes Bases de Donnes Avances (BDA'00), Blois, France, 1999.

[26] R. Feldman, Y. Aumann, Efficient Algorithms for Discovering Frequent Sets in Incremental Databases (1997).

[27] F. Masseglia, P. Poncelet, R. Cicchetti, An Efficient Algorithm for Web Usage Mining, Networking and Information Systems Journal .

[28] S. Lee, D. Cheung, B. Kao, Is Sampling Useful in Data Mining? A Case in the Maintenance of Discovered Association Rule, Data Mining and Knowledge Discovery 2 (3) (1998) 233–262.