Incremental Mining of Sequential Patterns in Large Databases

F. Masseglia(1,2) - P. Poncelet(2) - M. Teisseire(2)

(1)Laboratoire PRiSM, Université de Versailles
45 Avenue des Etats-Unis
78035 Versailles Cedex, France

(2)LIRMM UMR CNRS 5506
161, Rue Ada
34392 Montpellier Cedex 5, France

E-mail: {massegli, poncelet, teisseire}@lirmm.fr

Abstract

In this paper we consider the problem of 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 be useful for mining sequential patterns, since in many cases it is faster to apply our algorithm than mining sequential patterns with a classical algorithm, when the database is broken down into an original database with an increment.

1 Introduction

Most research into data mining has concentrated on the problem of mining association rules [1, 2, 4, 7, 8, 11, 12, 14]. Although sequential patterns are of great practical importance (e.g. alarms in a telecommunication network, identifying plan failures, analysis of Web access databases, etc.) they have received relatively little attention [3, 13, 16]. First introduced in [3], where an efficient algorithm called AprioriAll was proposed, the problem of mining sequential patterns is to discover temporal relations between facts embedded in the database. Considered facts are merely characteristics of individuals, or observations of individual behavior. For example, in a transactional 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 [13], the problem definition is extended by handling time constraints and taxonomies (*is-a* hierarchies) and a new algorithm, called GSP, that 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 [5]. Several efficient algorithms for maintaining association rules have been developed [5, 6, 15]. Nevertheless the problem of maintaining sequential patterns is much more complicated than maintaining association rules, since transaction cutting and sequence permutation has to be taken into account. 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 which can be extended to become frequent. Furthermore, old frequent sequences may become invalid when adding customer. The challenge is thus to discover all frequent patterns in the updated database with far greater efficiency than the naive method of mining sequential patterns from scratch. To the best of our knowledge, not much effort has been spent on maintaining sequential patterns. In [10], the authors propose an incremental mining algorithm, based on the SPADE approach [16], which can update the sequential patterns in a database when new transactions and new customers are added to the database. It is based on an increment sequence Lattice consisting of all the frequent sequences and all sequences in the negative border in the original database. This negative border is the collection of all sequences that are not frequent but both of whose generating sub-sequences are frequent. Furthermore, the support of each member is kept in the Lattice too. The main idea of this algorithm is that when incremental data arrives, the incremental part is scanned once to incorporate the new information into 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 this approach is very efficient, maintaining negative border is very memory consuming and not well adapted for very large databases [10].

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 novelty 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 discussed.
Empirical evaluations have been conducted 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 in an original database with an increment. Our experimental results show a factor of 2 to 5 improvement in performance in the comparison.

The rest of this paper is organized as follows. In section 2, the problem is stated and illustrated. The algorithm ISE is described in section 3. Section 4 presents the detailed experiments and performance results obtained are interpreted. Finally section 5 concludes the paper with future directions.

2 Problem statement

In this section we give the formal definition of the incremental sequential patterns mining problem. First, however, we formulate the concept of sequence mining summarizing the formal description of the problem introduced in [3] and extended in [13]. A brief overview of the GSP algorithm is also provided. Second we look at the incremental update problem in detail.

2.1 Mining of sequential patterns

Let DB be a set of customers' 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 $i_1 < i_2 < ... < i_n$ 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.
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[1]. In other words, the support of a sequence is defined as the fraction of total 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 user, and the sequence is said *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 sequences whose support is greater than a specified threshold (minimum support). Each of which represents a *sequential pattern*, also called a *frequent* sequence.

The task of discovering all 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 frequents sequences of length k) [16]. To the best of our knowledge, the problem of mining sequential patterns according to the previous definitions has received relatively little attention. Closer to the problem definition is the SPADE approach [16] where a very efficient Lattice based algorithm is proposed.
We now briefly review the GSP algorithm. Basically, exhibiting frequent sequences requires firstly retrieving all data sequences satisfying the minimum support. These sequences are considered as candidates for being patterns. The support of candidate sequences is then computed by browsing the database. Sequences for which the minimum support condition does not hold are discarded. The result is the set of frequent sequences.
For building up candidate and frequent sequences, the GSP algorithm makes multiple passes over the database. The first step aims to compute the support of each item in the database. When completed, frequent items (i.e. satisfying the minimum support) are discovered. They are considered as frequent 1-sequences (sequences having a single itemset, itself being a singleton). The set of candidate 2-sequences is built according to the following assumption: candidate 2-sequences could be any couple of frequent items, 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 the candidate generation is to retrieve, among (*k-1*)-sequences, couples of sequences (s, s') such that discarding the first element of the former and the last element of the latter results in two sequences fully matching. When such a condition holds for a couple (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 on customer-id and transaction time. $U = DB \cup db$ is the updated database containing all sequences from DB and 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.

[1] A sequence in a data sequence is taken into account only once to compute the support even if several occurrences are discovered.

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)

Figure 1: An original database (DB) and an increment database with new transactions (db)

Cust-Id	Itemsets		
$C1$	10 20	20	50 70
$C2$	10 20	30	40
$C3$	10 20	40	30
$C4$	60	90	
$C5$			

(DB)

Itemsets	
50 60 70	*80 100*
50 60	*80 90*
10 40	*70 80*

(db)

Figure 2: An original database (DB) and an increment database with new transactions and new customers (db)

Furthermore, the incremental approach has to take advantage of previously discovered patterns in order to avoid re-running 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, reporting facts about a population reduced to a mere four customers. Each transaction is ordered according to its time-stamp. For instance, the sequence of customer $C3$ is $<$ (10 20) (40) (30) $>$. Let us assume that the minimum support value is 50%, thus to be considered as frequent a sequence must be observed for at least two customers. The set of 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 two following data sequences $<$ (60) (90) $>$ and $<$ (10 20) (50 70) $>$ become frequent after database update since they have sufficient support. Let us consider the first one. 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 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 when scanning DB we find that the frequent sequences in L^{DB} were its predecessor.

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%, thus 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 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 frequent since it appears in the sequences of customer $C1$, $C2$ and $C3$. By introducing the increment database, the set of

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 items of 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

frequent sequences in the updated database is $L^U = \{< (10\ 20)\ (50) >, < (10)\ (70) >, < (10)\ (80) >, < (40)\ (80) >, < (60) >\}$. Let us have a closer look to 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 $C1$, $C2$ and the new customer $C5$.

3 ISE algorithm

In this section we introduce the ISE algorithm for computing frequent sequences in the updated database. First we describe our method for efficiently mining new frequent sequences using information collected during an earlier mining process. Then we present our algorithm. Optimization techniques are also addressed.

3.1 An overview

Basically we solve the problem of incremental mining of frequent sequences using information previously discovered. Let us consider that k stands for the length of the longest frequent sequences in DB. We decompose the problem into the two following sub-problems:

1. Find all new frequent sequences of size $j \leq (k+1)$. During this phase 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 adding items of db.
2. Find all frequent sequences of size $j > (k+1)$.

The second sub-problem can be solved in a straightforward manner since we are provided with frequent all $(k+1)$-sequences discovered in the previous phase. Applying a GSP-like approach at the $(k+1)^{th}$ step, we can generate candidate $(k+2)$-sequences and repeat the process until all frequent sequences are discovered. At the end of this phase all frequent sequences embedded in U are found. Hence, the problem of incremental mining of sequential patterns is reduced to the problem of finding frequent sequences of size $j \leq (k+1)$.

We summarize in Table 1 the notation used in the algorithm.

To discover frequent sequences of size $j \leq (k+1)$, the ISE algorithm executes iteratively. Since the main consequence of adding new customers is to verify the support of the frequent sequences in L^{DB}, in the following we first illustrate iterations through examples mainly concerning added transactions to existing customers. Finally, example 6 illustrates the behavior of ISE when new transactions and new customers are added to the original database.

3.1.1 First iteration

In 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 seed set is called L_1^{db}.
At the end of this pass, we prune out frequent sequences in L^{DB} no more verifying the minimum support.

Example 1 *Let us consider the increment database of figure 1. When scanning db 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 during a previous mining on* DB *we are provided 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 on 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. The candidate generation operates by joining L_1^{db} with L_1^{db} and yields the set of candidate 2-sequences. We scan on *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 of necessity be an extension of an original frequent sequence of DB, and then 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 seed set is called $freqExt$ and it is achieved by discarding from *2-candExt*, 2-sequences not verifying the minimum support.

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. For instance, since the candidate* $<(50)\ (60)>$ *does not appear in db, it is no more considered when scanning* U. *At the end of the scan on* U *with remaining candidates, we are thus provided with the following set of frequent 2-sequences,* 2-$freqExt = \{<(50\ 60)>, <(50)\ (80)>, <(50\ 70)>, <(60)\ (80)>\}$.

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

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 support* A *also.*

In order to efficiently find frequent sub-sequences preceding an item 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 for finding 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 of size $j \leq (k+1)$. 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	-

Figure 3: 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. We report in figure 3 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.* $60 \notin L^{DB}$. *In fact, this sequence will be discovered as frequent in the next phases of the algorithm.*
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 having a length lower or equal to k+1 (in *freqSeed*). In subsequent iterations we are led to discover frequent sequences not yet embedded in $freqSeed$ and *2-freqExt*. In fact, for generating $(k+n)$-sequences, with $n > 1$, we need all frequent $(k+1)$-sequences. For instance, if we want to show that $< (10\ 20)\ (40)\ (50\ 60) >$ is a frequent sequence it must be generated as a candidate. According to the GSP generation, this can only be done if we are provided with the two following frequent sequences: $< (20)\ (40)\ (50\ 60) >$ and $< (10\ 20)\ (40)\ (50) >$. Nevertheless, with the previous step we only obtain the last one.

3.1.2 j^{th} iteration (with $j \leq (k+1)$)

Let us assume that we are at the j^{th} pass with $j \leq k+1$. In these subsequent iterations, we start by generating new candidates from the two seed sets found in the previous pass. The main idea of

freqInc	*freqSeed*	*freqExt*
< (10) (50 60) (80) >	< (10 20) (30) (50) >	< (60) (90) >
< (20) (50 60) (80) >	< (10 20) (40) (50) >	
< (30) (50 60) (80) >	< (10 20) (30) (60) >	
< (40) (50 60) (80) >	< (10 20) (40) (60) >	
< (20) (30) (50 60) >	< (10 20) (30) (80) >	
< (20) (40) (50 60) >	< (10 20) (40) (80) >	
< (10 20) (50 60) >		
< (10) (30) (50 60) >		
< (10) (40) (50 60) >		
< (10) (30) (50 60) >		
< (10) (30) (50) (80) >		
< (10) (40) (50) (80) >		
< (20) (30) (50) (80) >		
< (20) (40) (50) (80) >		
< (10 20) (50) (80) >		
< (10 20) (50 70) >		

Figure 4: maximal frequent sequences such as $j \leq (k+1)$

the candidate generation is to retrieve among sequences of $freqSeed$ and *j-freqExt*, two sequences ($s \in freqSeed$, $s' \in j\text{-}freqExt$) such as an item $i \in L_1^{db}$ is the last item of s and the first item of s'. When such a condition holds for a couple (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 generate, using the same candidate generation algorithm as in GSP, 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 seed sets become respectively $freqInc$ and $(j+1)\text{-}freqExt$. These two sets are then used to generate new candidates. The process iterates until all frequent sequences are discovered or $j = k+1$.

At the end of this phase, $L^{U_{k+1}}$, the set of all frequent sequences having a size lower or equal to $k+1$, is obtained from L^{DB} and maximal sequences from $freqSeed \bigcup freqInc \bigcup freqExt$.

Example 4 *Considering our example, we know that $k = 3$, i.e. the longest size of the frequent sequences in L^{DB}. We can thus generate from $freqExt$ a new candidate sequence < (50 60) (80) > since its size is equal to k.*
Let us now consider how new frequent sequences are generated from $freqSeed$ and $2\text{-}freqExt$. Let us consider the sequence $s =$< (20) (40) (50) > from $freqSeed$ and $s' =$< (50 60) > from $2\text{-}freqExt$. The new candidate sequence < (20) (40) (50 60) > is obtained by dropping 50 from s and appending s' to the remaining sequence.
The maximal frequent sequences such as $j \leq (k+1)$ are listed in figure 4.

3.1.3 j^{th} iteration (with $j > (k+1)$)

Now, as all frequent sequences of size $j \leq (k+1)$ are discovered, we find the new frequent j-sequences in L^U where $j > k+1$. First we extract from the three seed sets ($freqSeed$, $freqExt$ and $freqInc$) the frequent (k+1)-sequences. New candidate (k+2)-sequences are then generated applying a GSP like approach and the process iterates until no more candidates are generated.
Pruning out non maximal sequences, we are provided with L^U the set of all frequent sequences in the updated database.

Example 5 *From the frequent (k+1)-sequences discovered in the previous example, we can generate the following candidates:* $<$ (10 20) (30) (50 60) $>$, $<$ (10 20) (40) (50 60) $>$, $<$ (20) (30) (50 60) (80) $>$ *and* $<$ (20) (40) (50 60) (80) $>$. *As they are frequent in* U*, they can be used to generate candidates for the next step, and at the end of the process we are provided with the two frequent sequences:* $<$ (10 20) (30) (50 60) (80) $>$ *and* $<$ (10 20) (40) (50 60) (80) $>$.

For easy understanding, we graphically describe in figure 5 processes in the first and j^{th} (with $j \leq (k+1)$) iterations.

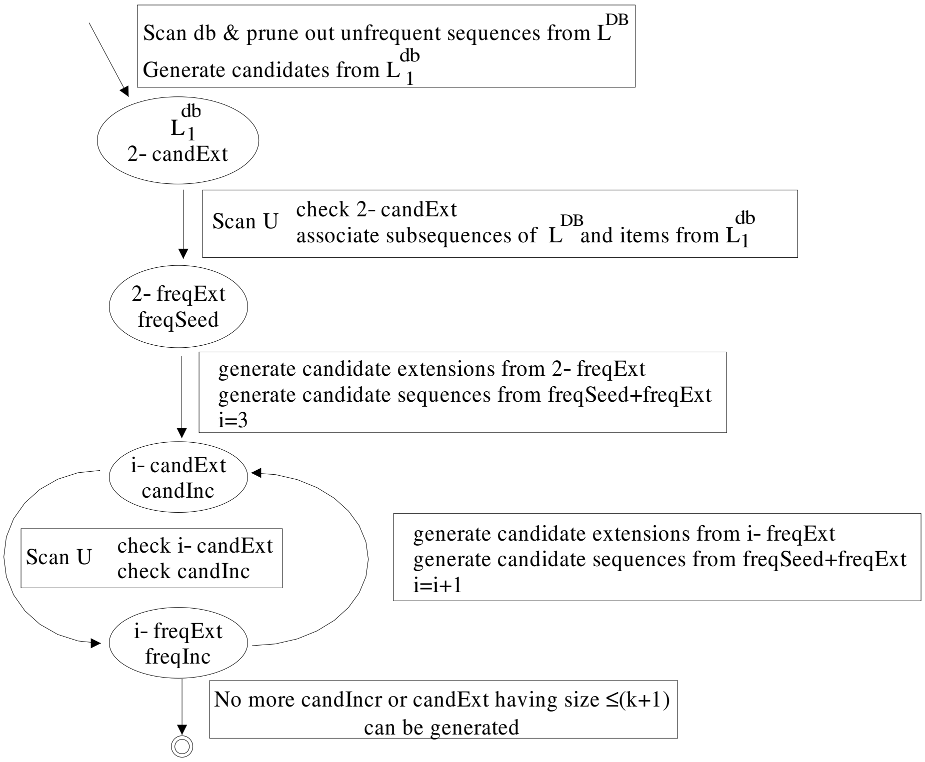


Figure 5: Processes in the first and j^{th} (with $j \leq (k+1)$) iterations of ISE

In order to illustrate how new customers are also taken into account in the ISE algorithm, let us consider the following example.

Example 6 *Let us now consider figure 2 when a new customer as well as new transactions are added to the original database. When scanning db 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 is added, to be frequent, a sequence must appear in at least three transactions. Let us now consider* L^{DB}. *The set* L_1^{DB} *becomes :* $\{(<(10),4>),(<(20)>,3),(<(40)>,3)$. *That is to say that the item* 30 *is pruned out from* L_1^{DB} *since it is no more 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. We can now generate from* L_1^{db} *new candidates in 2-candExt:* $\{<(10\ 40)>,\ <(10)\ (40)>,\ <(10\ 50)>, \ldots\ <(70)\ (80)>\}$. *When scanning db, we prune out candidates not occurring in the increment and we are provided with candidate 2-sequences occurring at least once in db. Next we scan* U *for verifying 2-candidates and sequences of the updated* L^{DB} *preceding, according to the timewise order, sequences of* L_1^{db}. *There is only three* *candidate sequences verifying the support:* $2\text{-}freqExt = \{<(10)\ (70)>,\ <(10)\ (80)><(40)\ (80)\}$. *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	$<(10)>_2$ $<(20)>_2$ $<(10\ 20)>_2$
80	$<(10)>_2$ $<(20)>_2$ $<(10\ 20)>_2$

The minimum support constraint holds only for sequences preceding item 50. *Then, we have:* $freqSeed = \{<(10\ 20)\ (50)>\}$. *Since, we cannot generate new candidates from* $freqSeed$ *and* $2\text{-}freqExt$*, the process complete and all maximal frequent sequences are stored into* $L^U = \{<(10\ 20)\ (50)>, <(10)\ (70)>, <(10)\ (80)>, <(40)\ (80)>, <(60)>\}$.

3.2 The Ise Algorithm

Based on the above discussion, the Ise algorithm is presented.

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-*candExt* ← 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 candidates of 2-candExt and frequent sub-sequences occurring before items of L_1^{db};
  $freqSeed$ ← frequent sub-sequences occurring before items of L_1^{db} and appended with the item;
  2-$freqExt$ ← frequent sequences from 2-candExt;

// j^{th} *Iteration (with* $j \leq (k+1)$*)*
  j=2;
  While (*j*-$freqExt$!=$\emptyset$ AND $j \leq (k+1)$) **do**
    candInc ← generate candidates from $freqSeed$ and j-$freqExt$;
    j++;
    j-*candExt* ← Generate candidate j-sequences from j-$freqExt$;
// find sequences occurring in db
  scan *db* for j-*candExt*;
    if (*j*-*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 $freqSeed \bigcup freqInc \bigcup freqExt$};

// j^{th} Iteration (with $j > (k+1)$)
For each $k+1$-sequence found apply GSP until all frequent sequences are discovered;

$L^U \leftarrow L^U \bigcup$ {maximal frequent sequences obtained from the previous step};
end Algorithm ISE

To prove that ISE provides the set of frequent sequences embedded in U, we first show in the two following lemmas that every new frequent sequence can be broken down in two sub-sequences. The former is a frequent sequence into 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 $|F| > 1$: Let S denote a frequent sequence on U, that is, S=$< <A> <B> >$ with A and B sequences such that $0 \leq |A| < |F|$, $0 < |B| \leq |F|$, $|A| + |B| = |F|$ and $B \notin db$. 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 $\sigma \leq m \leq n$. Furthermore $M_{AB} \in DB$ (since $B \notin db$ and transactions are ordered by time) then $< <A> <B> >$ is frequent on DB and $S \in L^{DB}$. Thus if a sequence F does not appear in L^{DB}, F ends with an itemset occurring at least once in *db* $\square$

Lemma 2 *Let $F = < <D> <S> >$, where D and S are two sequences verifying $|D| \geq 0$, $|S| \geq 1$, be a frequent sequence in U such that F does not appear in L^{DB}, then S occurs 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 be the set of all data sequences containing F and $time_u$ the update time stamp. $\forall s \in M$, $s =< <A> <B> >$ / $\forall i \in A$, $time_i < time_u$, $\forall j \in B$, $time_j \geq time_u$, we have $< (i_{j-1})(i_j)..(i_t) > \not\subset B$ (since S is the maximal sub-sequence ending F and occurring at least once in *db*). Thus, since $\forall i$, M_i contains F, and since transactions in the database are ordered by increasing transaction-time, $< (i_1)(i_2)..(i_{j-1}) > \subset A$. Thus $\forall s \in M$, $\exists A$ a sub-sequence beginning s such that $A \in DB$, $D \subseteq A$. Thus $D \in L^{DB}$ $\square$

Considering the two sub-sequences of a new frequent sequence, we show 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} and $|F| \leq k+1$. F can be written as $<< D >< S >>$ where D and S are two sequences verifying $|D| \geq 0$ and $|S| \geq 1$, S occurs 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 is occurring 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 'a la GSP' from all frequent items in *db* and is thus a superset of all frequent sequences on U occurring in *db* and having length $\leq k+1$ □

The following Theorem guarantees the correctness of the ISE approach at the $(k+1)^{th}$ step. Then, a GSP-like approach is applied based on the maximal frequent sequences obtained from $freqSeed \bigcup freqInc \bigcup freqExt$.

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 $<< D >< (i_{1_1}) >>$ will be generated in $freqSeed$. According to Lemma 3, S is occurring in $freqExt$ and will be used by ISE to build $<< D >< S >>$ in *candInc* and added to $freqInc$ □

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 algorithm 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 as an item $i \in L_1^{db}$ is the last item of s and the first item of s'. We do not generate a new candidate by appending s' to s if there exists an item $j \in L_1^{db}$ such as j is in s' and j is not preceded by s.*

proof: If s is not followed by j, then $< s\ j >$ is not frequent. Hence $< s\ s' >$ is not frequent since there exists a not frequent sub-sequence of $< s\ s' >$.

Using this lemma, at the j^{th} iteration, with $j \leq (k+1)$, we can reduce the number of candidates significantly. In our experiments, the number of candidates was reduced by nearly 40%. The only additional cost is to find for each item occurring in the second sequence whether there is a frequent sub-sequence matching the first one. As we are provided with an array storing for each frequent sub-sequence the items occurring after the sequence, the additional cost of this optimization is relatively inexpensive.

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

|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^-	Number of itemsets removed from U in order 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

Example 7 *Let us consider the frequent sequence $s \in 2\text{-}freqExt$ 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-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 were already found 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 the $freqSeed$ set. In this case the candidate is no longer considered. As illustration, consider that $< (30)\ (40) >$ is a frequent sequence in $2\text{-}freqExt$. 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 scanning U at a 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 an Enterprise 2 (Ultra Sparc) Station with a CPU clock rate of 200 MHz, 256M Bytes of main memory, UNIX System V Release 4 and a non local 9G Bytes disk drive (Ultra Wide SCSI).

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 [13][2]. The generation of DB and db was performed as follows. As we would like to model very well real life updates, like in [5], we first generated all the transactions from the same statistical pattern, then databases of size $|U|$=$|DB$+$db|$ were first 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

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

Name	$\lvert C \rvert$	$\lvert I \rvert$	N	$\lvert D \rvert$
C9-I4-N1K-D50K	9	4	1,000	50,000
C9-I4-N2K-D100K	9	4	2,000	100,000
C12-I2-N2K-D100K	12	2	2,000	100,000
C12-I4-N1K-D50K	12	4	1,000	50,000
C13-I3-N20K-D500K	13	3	20,000	500,000
C15-I4-N30K-D600K	15	4	30,000	600,000
C20-I4-N2K-D800K	20	4	2,000	800,000

Table 3: Parameter values for synthetic datasets

behavior of ISE when adding new customers, the number of removed customers from U was provided by the parameter $C^{\%}$.
Table 2 gives the list of the parameters used in the data generation method and Table 3 shows the databases used and their properties. For experiments we first investigate the behavior of ISE when adding new transactions. For these experiments, I^{-} was set to 4 and $D^{\%}$ was set to 90%. Finally when studying the performance of our algorithm with new customers $C^{\%}$ was set to 10% and 5%.

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 its scale up as the number of transactions increases. Finally, we have carried out experiments for analyzing the performance of the ISE algorithm according to the size of updates.

4.2.1 Naive vs. ISE algorithm

Figure 6 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" will be explained in Section 4.3.

Figure 6 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 6 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.
The reason is that the number of candidates generated for mining j-sequences with $j \leq k+1$ is substantially reduced since candidate generation is based on items embedded in the increment database, i.e. candidate sequences which do not have sufficient supports relative to *db* are pruned out before they are verified by DB and are not used for generating sequences for *candExt*. We will study in Section 4.3.1 the correlation between execution times and the number of candidates.

4.2.2 Performance in scaled-up databases

We examined how ISE behave as the number of total transactions is increased. We would expect the algorithm to have near linear scaleup. This is confirmed by figure 7 which shows that ISE scales up linearly as the number of transactions is increased ten-fold, from 0.1 million to 1 million. Experiments

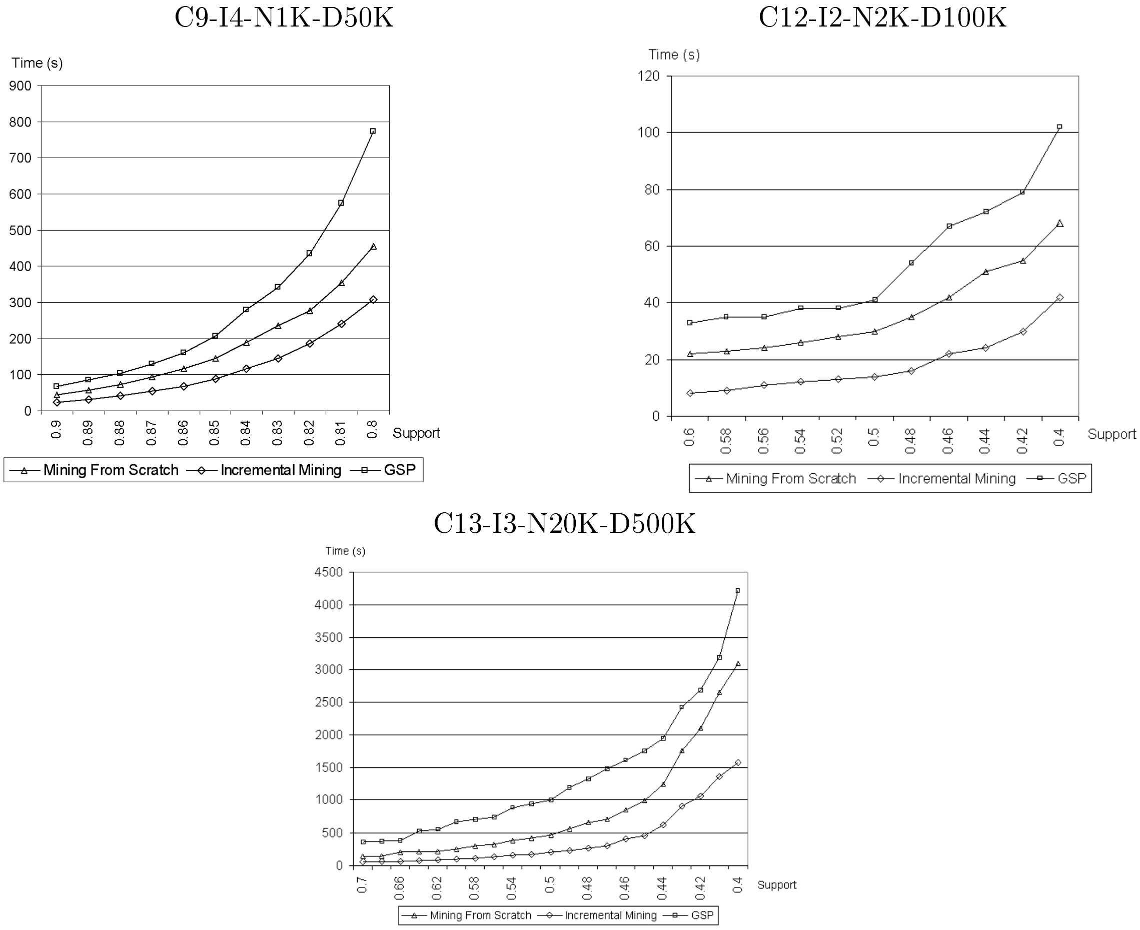


Figure 6: Execution times

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 have carried out some experiments for analyzing 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 0.4%. From these databases, we investigated the performance of ISE when varying the number of itemsets removed from the generated database as well as the number of clients. Transactions deleted were stored in the increment database db while the remaining transactions were stored in the database DB. We first ran GSP for mining L^{DB} and then ran ISE on the updated database. Figure 8 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 amount of changes to the original database are so numerous that the result obtained during an earlier mining are not helpful. Interestingly, we also

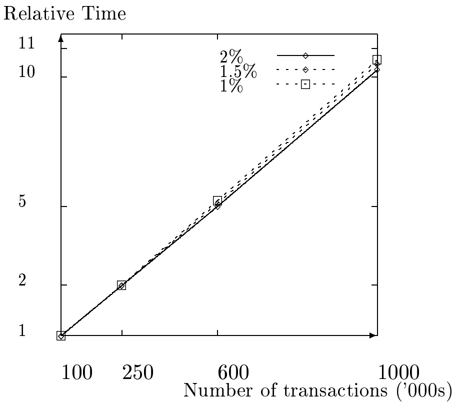
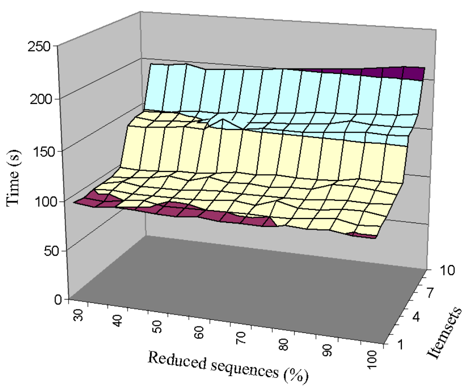


Figure 7: Scale-up: Number of total transactions



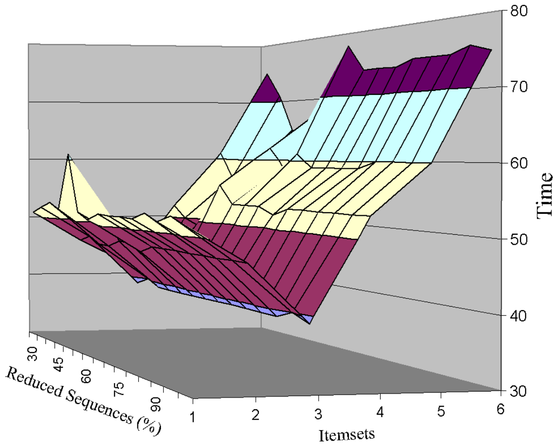


Figure 8: Size of updates

noticed that the time taken by the algorithm does not depend very much on the number of updated transactions.
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 for discovering the set of all sequential patterns.

4.2.4 Varying the number of added customers

We assume, since it is realistic and well adapted to real applications, that the average size of the added sequences is at most equal to the average size of the sequences embedded in the original database. A first intuition could be to study the behavior of ISE when increasing the amount of new customers added to the database. In fact, this naïve idea could not be a significant indicator. This is because when the number of new customers increase, then the number of occurrences of a sequence must also be increased for verifying the minimum support constraint. Obviously, as at the beginning of the ISE algorithm, we prune out from L^{DB} frequent sequences not verifying anymore 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 as well as the execution times.

A much more interesting approach for evaluating ISE performance is to carry out experiments for comparing execution times of GSP vs. ISE on different datasets when varying the minimum support. We report in figure 9 experiments conducted on two datasets C9-I4-N2K-D100K and C20-I4-N2K-D800K where respectively 10% and 5% of customers have been added. We can observe that ISE is very efficient and even when adding customers it is nearly twice faster than applying GSP from scratch.

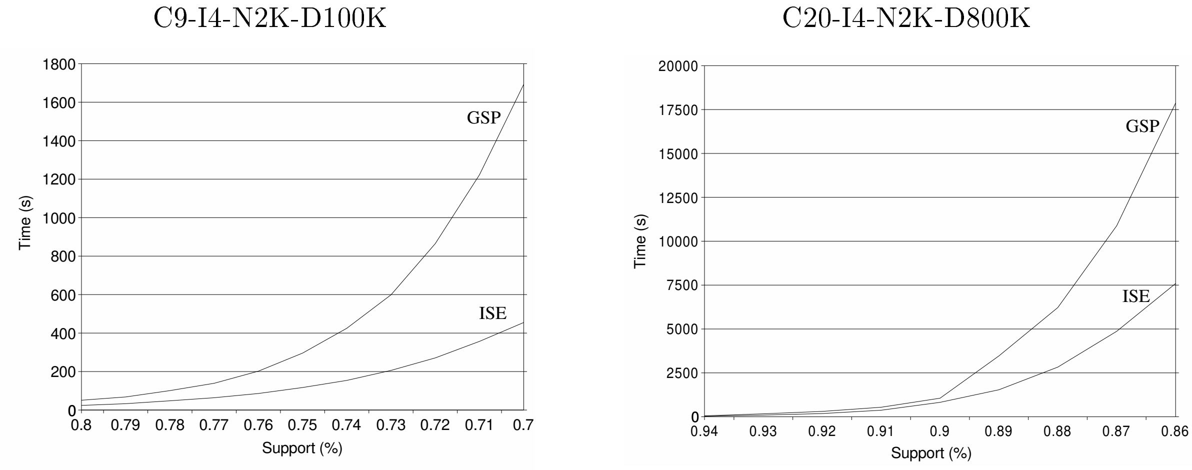


Figure 9: Execution times when 10% and 5% of customers are added to the original database

4.3 ISE for mining sequential patterns

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

We have 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% of transactions in order to provide the increment database. Second we ran GSP for mining the k-frequent sequences in DB. Finally we ran ISE. Figure 6 shows the result of the experiments and the label "Mining from scratch" corresponds to ISE for mining sequential patterns. We can observe that ISE is 1,7 to 3 faster than GSP for mining sequential patterns. The main reason for the performance gain is the reduction in the number of candidates. We study this effect in the next section. As expected, we also can observe that using ISE for incremental mining instead of mining from scratch is still efficient since the incremental mining is nearly twice faster than mining from scratch.

4.3.1 Candidate Sets

In order to explain 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 10. 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 while 8000 candidates are generated by our algorithm.

We have also designed some experiments to analyze the time spent in each phase of the algorithm. Figure 11 shows the result of this experiment. Even if ISE is more efficient than using GSP from scratch, we can see that finding sequential patterns at step $k+1$ is very time consuming. For instance, at low level support, 0.22%, we spent nearly 400 seconds finding frequent sequences. Despite this delay,

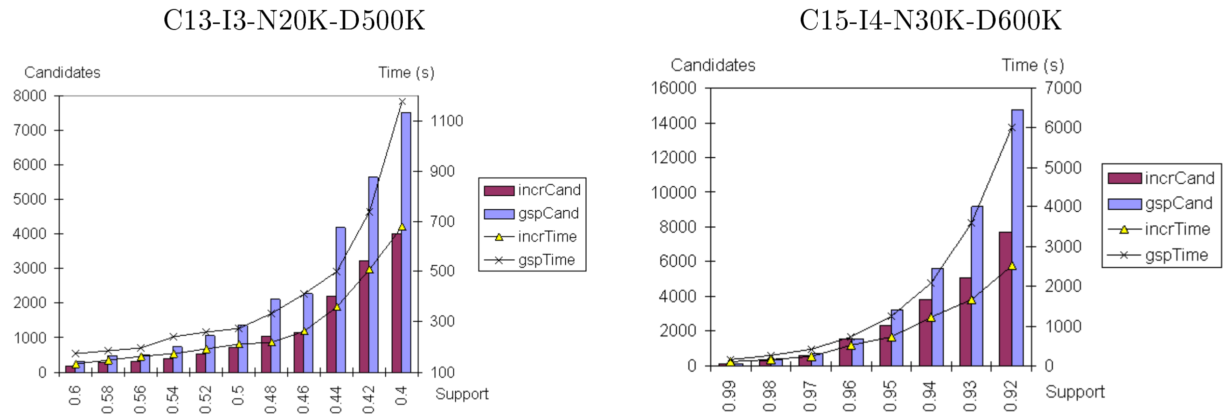


Figure 10: Candidate sets

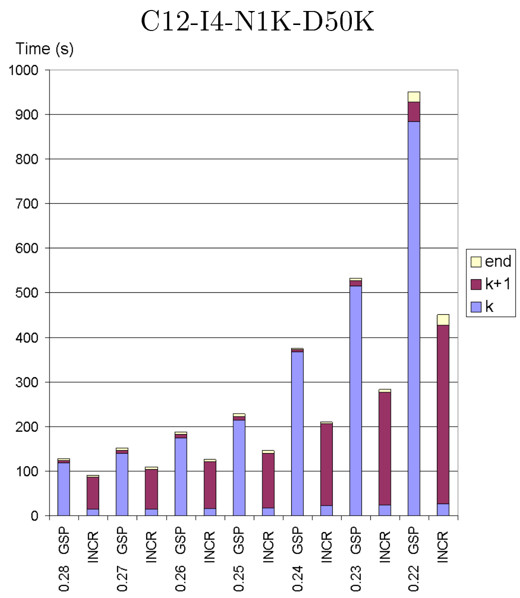


Figure 11: Time spent in each phase

obtaining frequent sequences at $k+1$ is twice as fast as running GSP. Furthermore, as expected, the time spent for finding frequent sequences at the next step, i.e. when $j > K+1$ is the same for GSP and ISE.

4.3.2 Varying the size of updates

Finally, in order to analyze the performance of ISE algorithm according to the size of updates, we have carried out some experiments. 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 12. 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 from 2 to 7 itemsets deleted but when 5 itemsets are deleted to 10 % of customer, ISE is less efficient.
In the second surface of Figure 12, the performance of ISE is quite similar and best results are obtained when 9 itemsets are removed from 80% of customers.

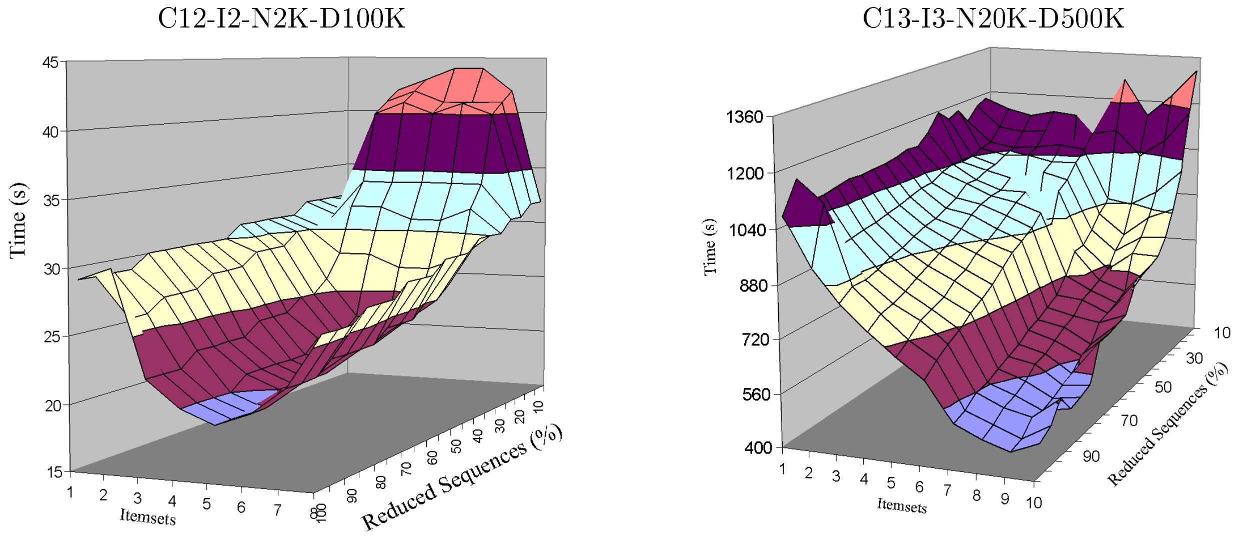


Figure 12: Size of updates

5 Conclusion

In this paper we present the ISE approach for incremental mining of sequential patterns in a large database. This method is based on the discovery of frequent sequences by only considering frequent sequences obtained by an earlier mining. 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 during empirical evaluations that the proposed approach was so efficient that it was quicker to extract an increment from the original database then apply ISE for mining sequential patterns than use the GSP algorithm.

Future work on incremental mining will be done in various directions. First, even if the incremetal approach is applicable to the databases which allow frequent updates when new transactions or new customers are added to an original database, many domains such as electronic commerce or web usage mining impose that deletion or modification must be taken into account in order to save storage space or because the information is out of interest or becomes invalid. We are currently investigating how to manage these operations in the ISE algorithm.
Second, we are currently studying how to improve the whole process of incremental mining. According to experiments we would like to find out some measures which can suggest to us when ISE should be applied to find out the new frequent sequences in the updated database. In other context [9], such an approach is proposed and is based on a sampling technique in order to estimate the difference between the old and new association rules. We are currently investigating if when analysing the data distribution of the original database we could find out other measures.

References

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

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

[3] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, Tapei, Taiwan, March 1995.

[4] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proceedings of the International Conference on Management of Data (SIGMOD'97)*, pages 255–264, Tucson, Arizona, May 1997.

[5] D.W. Cheung, J. Han, V.T. Ng, and C.Y. 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, March 1996.

[6] D.W. Cheung, S.D. Lee, and 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, April 1997.

[7] U.M. Fayad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI Press, Menlo Park, CA, 1996.

[8] G. Gardarin, P. Pucheral, and F. Wu. Bitmap Based Algorithms For Mining Association Rules. In *Actes des journées Bases de Données Avancées (BDA'98)*, Hammamet, Tunisie, October 1998.

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

[10] S. Parthasarathy, M.J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and Interactive Sequence Mining. In *Proceedings of the 8th International Conference on Information and Knowledge Management (CIKM'99)*, pages 251–258, Kansas City, MO, USA, November 1999.

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

[12] A. Savasere, E. Omiecinski, and 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)*, pages 432–444, Zurich, Switzerland, September 1995.

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

[14] H. Toivonen. Sampling Large Databases for Association Rules. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB'96)*, September 1996.

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

[16] M. Zaki. Scalable Data Mining for Rules. Technical Report PHD Dissertation, University of Rochester - New York, 1998.