Incremental and Interactive Sequence Mining *

S. Parthasarathy, M. J. Zaki[†], M. Ogihara, S. Dwarkadas
Computer Science Dept., U. of Rochester, Rochester, NY 14627
[†] Computer Science Dept., Rensselaer Polytechnic Inst., Troy NY 12180

Abstract

The discovery of frequent sequences in temporal databases is an important data mining problem. Most current work assumes that the database is static, and a database update requires rediscovering all the patterns by scanning the entire old and new database. In this paper, we propose novel techniques for maintaining sequences in the presence of a) database updates, and b) user interaction (e.g. modifying mining parameters). This is a very challenging task, since such updates can invalidate existing sequences or introduce new ones. In both the above scenarios, we avoid re-executing the algorithm on the entire dataset, thereby reducing execution time. Experimental results confirm that our approach results in substantial performance gains.

1 Introduction

Sequence mining is an important data mining task, where one attempts to discover frequent sequences over time, of attribute sets in large databases. This problem was originally motivated by applications in the retailing industry, including attached mailing, add-on sales and customer satisfaction. It also applies to many scientific and business domains. For instance, in the health care industry it can be used for predicting the onset of disease from a sequence of symptoms, and in the financial industry it can be used for predicting investment risk based on a sequence of stock market events.

Discovering all frequent sequences in a very large database can be very compute and I/O intensive because the search space size is essentially exponential in the length of the longest transaction sequence in it. This high computational cost may be acceptable when the database is static since the discovery is done only once, and several approaches to this problem have been presented in the literature. However, when the databases are updated on

*Contact Author: S. Parthasarathy, 716-275-1192. This work was supported in part by NSF grants CDA–9401142, CCR–9702466, CCR–9705594, CCR–9701911, CCR–9725021, and INT-9726724; and an external research grant from Digital Equipment Corporation.

a regular basis, running the discovery program all over again when there is an update might produce significant computation and I/O loads. Hence, there is a need for algorithms that perform frequent sequence searches on incrementally updated databases without having to run the entire mining algorithm again.

Although the incremental mining problem has been studied for association rules [2, 3, 10], no work has been done for incremental mining of sequential patterns. In this paper, we present a method for incremental sequence mining. Our goal is to minimize the I/O and computation requirements for handling incremental updates. Our algorithm accomplishes this goal by maintaining information about "maximally frequent" and "minimally infrequent" sequences. When incremental data arrives, the incremental part is scanned once to incorporate the new information. The new data is combined with the "maximal" and "minimal" information in order to determine the portions of the original database that need to be re-scanned. This process is aided by the use of a vertical database layout — where attributes are associated with the list of transactions in which they occur. The result is an improvement in execution time by several orders of magnitude in practice, both for handling increments to the database, as well as for handling interactive queries.

The rest of the paper is organized as follows. In Section 2, we formulate the sequence discovery problem. In Section 3, we describe the SPADE algorithm upon which we build our incremental approach. Section 4 describes our incremental sequence mining algorithm. In Section 5, we describe how we support online querying. An experimental evaluation is presented in Section 6. We discuss related work in Section 7, and conclude in Section 8.

2 Problem Formulation

In this section, we define the incremental sequence mining problem that this paper is concerned with. We begin by defining the notation we use. Let the *items*, denoted $\mathcal{I}$, be the set of all possible attributes. We assume a fixed enumeration of all members in $\mathcal{I}$ and identify the items with their indices in the enumeration. An *itemset* is a set of items. An itemset is denoted by the enumeration of its elements in increasing order. The order is enfordec purely in order to avoid redundant traversals. For an itemset i, its *size*, denoted by $|i|$, is the number of elements in it. An itemset of size k is called a k-*itemset*.

A *sequence* is an ordered list of nonempty itemsets. A sequence of itemsets $\alpha_1, \ldots, \alpha_n$ is denoted by $(\alpha_1 \mapsto \cdots \mapsto \alpha_n)$. The *length* of a sequence is the sum of the sizes of each of its itemsets. For each integer k, a sequence of length k is called a *k-sequence*. A sequence α is a *subsequence* of a sequence β, denoted by $\alpha \preceq \beta$ if α, can be constructed from β by striking out some (or none) of the items in β, and then by eliminating all the occurrences of $\emptyset \mapsto$ and $\mapsto \emptyset$ one at a time. For example, $B \mapsto AC$ is a subsequence of $AB \mapsto E \mapsto ACD$. We say that α is a *proper subsequence* of β, denoted $\alpha \prec \beta$, if $\alpha \neq \beta$ and $\alpha \preceq \beta$. For $k \geq 3$, the *generating subsequences* of a length k sequence are the two length $k-1$ subsequences of α obtained by dropping exactly one of its first or second items. By definition, the generating sequences share a common suffix of length $k-2$. For example, the two generating subsequences of $AB \mapsto CD \mapsto E$ are $A \mapsto CD \mapsto E$ and $B \mapsto CD \mapsto E$, and they share the common suffix $CD \mapsto E$. A sequence is *maximal* in a collection $\mathcal{C}$ of sequences if the sequence is not a subsequence of any other sequence in $\mathcal{C}$.

Our database is a collection of *customers*, each with a sequence of *transactions*, each of which is an itemset. For a database D and a sequence α, the *support* or *frequency* of α in D, denoted by $support_D(\alpha)$, is the number of customers in D whose sequences contain α as a subsequence. The *minimum-support*, denoted by *min_support*, is a user-specified threshold that is used to define "frequent sequences": a sequence is frequent in D if its support in D is at least *min_support*. A rule $A => B$involving sequence A and sequence B is said to have *confidence* c if $c\%$ of the customers that contain A also contain B.

Suppose that new data is about to be added to a database D. The database D is referred to as the *original database* and the new update δ as the incremental database. The updated database is denoted by $D + \delta$. For each $k \geq 1$, $\mathcal{F}_k$ denotes the collection of all frequent sequences of length k. *FS* denotes the set of all frequent sequences. The *negative border* (*NB*) is the collection of all sequences that are not frequent but both of whose generating subsequences are frequent. By the *old sequences,* we mean the set of all frequent sequences in the original database and by the *new sequences* we mean the set of all frequent sequences in the join of the original and the increment.

For example, consider the customer database shown in Figure 1. The database has three items (A, B, C), and four customers. The figure also shows all the frequent sequences (the frequency is also shown with each node) and the negative border, when a minimum support of 75%, or 3 customers, is used. For each frequent sequence, the figure shows its two generating subsequences in bold lines. Figure 2 shows how the frequent set and the negative border change when we mine over the combined original and incremental database. For example, C is not frequent in the original database D, but C (along with some of its supersequences)

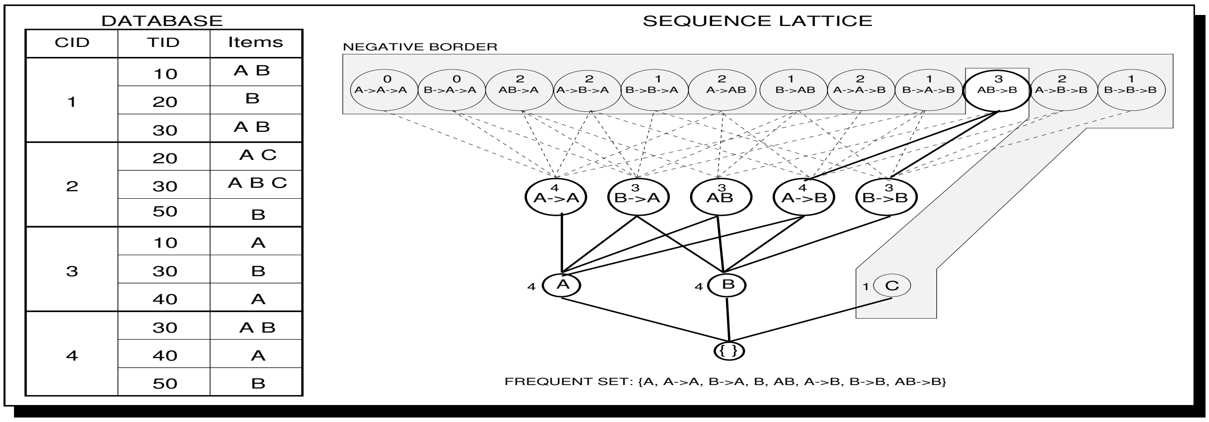

Figure 1: Original Database

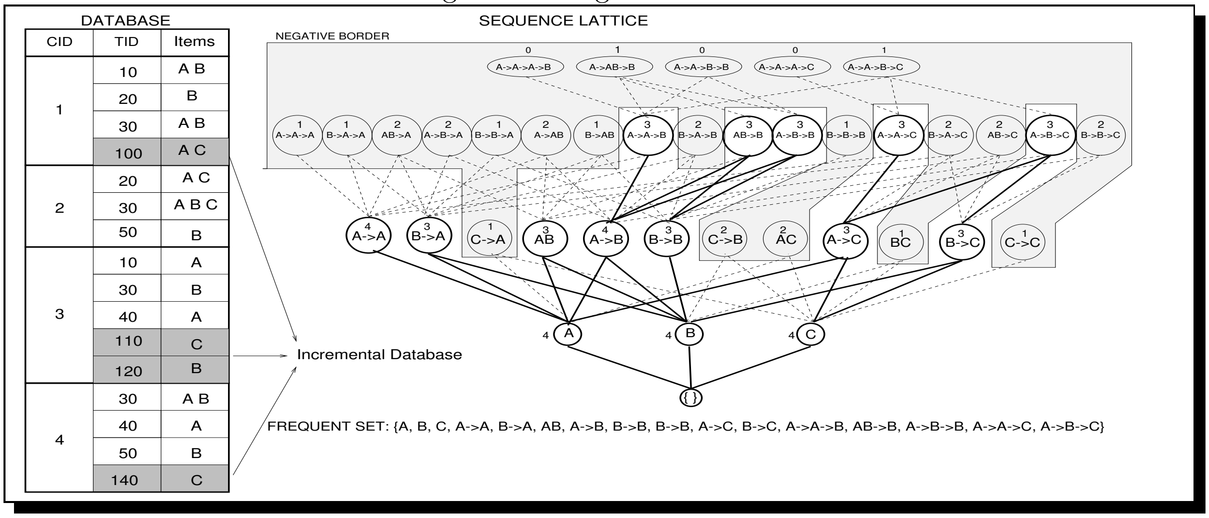

Figure 2: Original plus Incremental Database and Lattice

becomes frequent after the update $D + \delta$. The update also causes some elements to move from NB to the new FS.

Incremental Sequence Discovery Problem: Given an original database D of sequences, and a new increment to the database δ, find all frequent sequences in the database $D + \delta$, with minimum possible recomputation and I/O.

3 The SPADE Algorithm

In this section we describe SPADE [12], an algorithm for fast discovery of frequent sequences, which forms the basis for our incremental algorithm.

Sequence Lattice: SPADE uses the observation that the subsequence relation $\preceq$ defines a partial order on the set of sequences, also called a *specialization relation*. If $\alpha \preceq \beta$, we say that α is more general than β, or β is more specific than α. The second observation used is that the relation $\preceq$ is a *monotone specialization relation* with respect to the frequency $\sigma(\alpha, \mathcal{D})$, i.e., if β is a frequent sequence, then all subsequences $\alpha \preceq \beta$ are also frequent. The algorithm systematically searches the sequence lattice spanned by the subsequence relation, from the most general to the maximally specific frequent sequences in a breadth/depth-first manner. Figure 3 shows the lattice of frequent sequences for our example database.

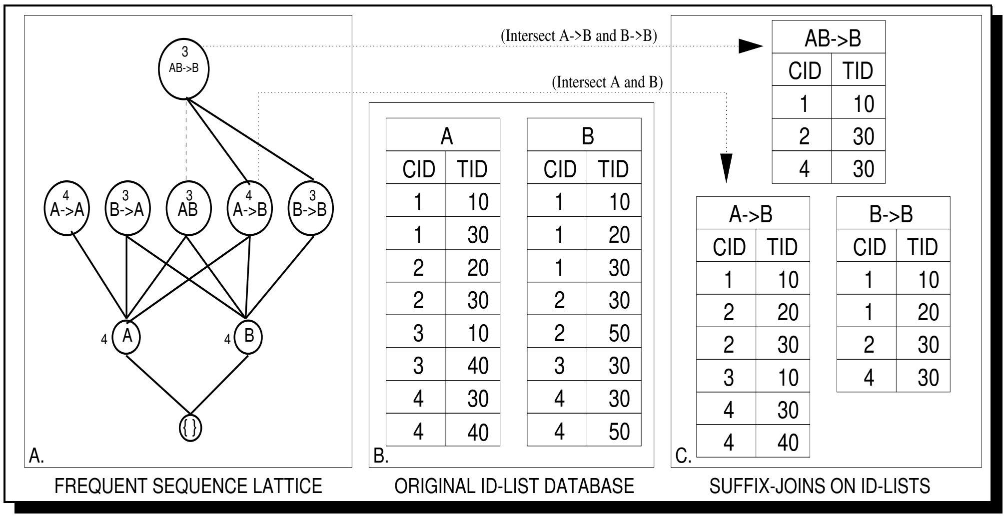

Figure 3: A. Frequent Sequence Lattice; B. Initial Idlist Database; C. idlist Intersections

Support Counting: Most of the current sequence mining algorithms [8] assume a *horizontal* database layout such as the one shown in Figure 1. In the horizontal format, the database consists of a set of customers (*cid's*). Each customer has a set of transactions (*tid's*), along with the items contained in the transaction. In contrast, we use a *vertical* database layout, where we associate with each item X in the sequence lattice its *idlist*, denoted $\mathcal{L}(X)$, which is a list of all customer (*cid*) and transaction identifier (*tid*) pairs containing the atom. Figure 3 B) shows the idlists for all the items.

Given the sequence idlists, we can determine the support of any k-sequence by simply intersecting the idlists of any two of its $(k-1)$ length subsequences. In particular, we use the two $(k-1)$ length subsequences that share a common suffix (the generating sequences) to compute the support of a new k length sequence. A simple check on the support of the resulting idlist tells us whether the new sequence is frequent or not. Figure 3 shows this process pictorially. It shows the initial vertical database with the idlist for each item. The intermediate idlist for $A \mapsto B$ is obtained by intersecting the lists of A and B, i.e., $\mathcal{L}(A \mapsto B) = \mathcal{L}(A) \cap \mathcal{L}(B)$. Similarly, $\mathcal{L}(AB \mapsto B) = \mathcal{L}(A \mapsto B) \cap \mathcal{L}(B \mapsto B)$.

Lattice Decomposition – Suffix-Based Classes: If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing inter-

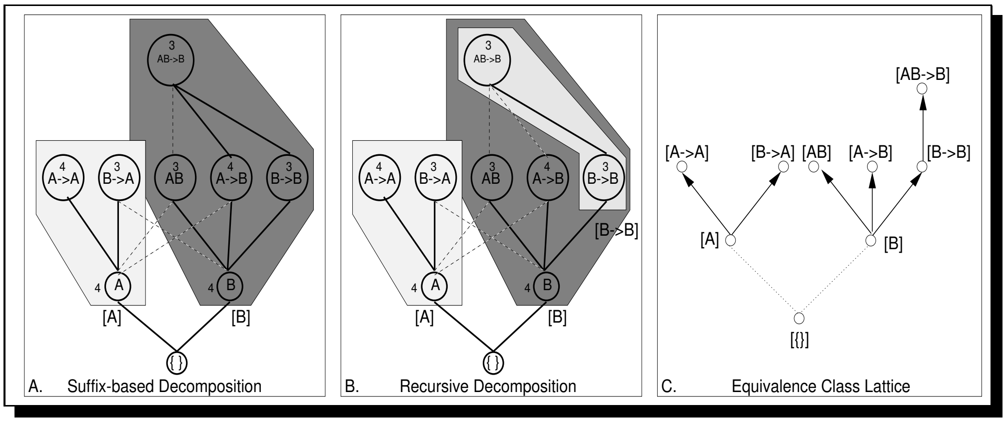

Figure 4: A) Initial Decomposition: Suffix Length 1, B) Level 2 Decomposition: Suffix Length 2, C) Recursive Decomposition: Class Tree

sections to obtain sequence supports. In practice, however, we only have a limited amount of main-memory, and all the intermediate idlists will not fit in memory. SPADE breaks up this large search space into small, manageable chunks which can be processed *independently* in memory. This is accomplished via suffix-based equivalence classes. We say that two k length sequences are in the same class if they share a common $k-1$ length suffix. The key observation is that each class is a sub-lattice of the original sequence lattice and can be processed independently. For example, Figure 4A) shows the effect of decomposing the frequent sequence lattice for our example database, by collapsing all sequences with the same 1 length suffix into a single class. There are two resulting suffix classes, namely, $\{[A], [B]\}$, which are referred to as *parent classes*. Each class is independent in the sense that it has complete information for generating all frequent sequences that share the same suffix. For example, if a class $[X]$ has the elements $Y \mapsto X$, and $Z \mapsto X$, the only possible frequent sequences at the next step can be $Y \mapsto Z \mapsto X$, $Z \mapsto Y \mapsto X$, and $(YZ) \mapsto X$. It should be obvious that no other item Q can lead to a frequent sequence with the suffix X, unless (QX) or $Q \mapsto X$ is also in $[X]$.

SPADE recursively decomposes the sequences at each new level into even smaller independent classes. Figure 4B) shows the effect of using 2-length suffixes. If we do this at all levels we obtain a tree of independent classes as shown in Figure 4C). This computation tree is processed in a breadth-first manner, within each parent class. In other words, parent classes are processed one-by-one, but within a parent class we process the new classes in a breadth-first search. Figure 5 shows the pseudo-code for the breadth-first search. The input to the procedure is a class, along with the idlist for each of its elements. Frequent sequences are generated by intersecting the idlists of all distinct pairs of sequences in each class and

SPADE($\mathcal{D}$, *min_sup*):
$\mathcal{C} = \{$ parent classes, $C_i\}$
for each parent class C_i **do** *Enumerate-Frequent-Seq(C_i)*

Enumerate-Frequent-Seq(S):
for all elements $A_i \in S$ **do**
$T_i = \emptyset$;
for all elements $A_j \in S$, with $j > i$ **do**
$R = A_i \cup A_j$;
if (*Prune*(R) == FALSE) **then**
$\mathcal{L}(R) = \mathcal{L}(A_i) \cap \mathcal{L}(A_j)$;
if $support(R) \geq min\_sup$ **then**
$T_i = T_i \cup \{R\}$; $\mathcal{F}_{|R|} = \mathcal{F}_{|R|} \cup \{R\}$;
for all $T_i \neq \emptyset$ **do** *Enumerate-Frequent-Seq*(T_i);

Figure 5: SPADE: Pseudo-code for Enumerating Frequent Sequences

checking the support of the resulting idlist against *min_sup*. The sequences found to be frequent at the current level form classes for the next level. This level-wise process is repeated until all frequent sequences have been enumerated. In terms of memory management, it is easy to see that we need memory to store intermediate idlists for at most two consecutive levels. Once all the frequent sequences for the next level have been generated, the sequences at the current level can be deleted. For more details on SPADE, see [12].

4 Incremental Mining Algorithm

Our purpose is to minimize recomputation or re-scanning of the original database when mining sequences in the presence of increments to the database (the increments are assumed to be appended to the database, i.e., later in time).

In order to accomplish this, we provide use an efficient memory management scheme that indexes into the database efficiently, and create an Increment Sequence Lattice (ISL), exploiting its properties to prune the search space for potential new sequences. The ISL consists of all elements in the negative border and the frequent set, and is initially constructed using the SPADE algorithm described in the previous section. In the ISL, the children of each nonempty sequence are its generating subsequences. Each node of the ISL contains the support for the given sequence. Figure 1 shows the ISL for the original database, while Figure 2 shows the ISL after the database update.

We begin this section by first describing the memory management scheme to access the database in an efficient way. We then explore the properties of the ISL and show how one can prune the search space for potential new sequences. Finally we provide details on our algorithm.

Memory Management: In order to run incremental mining fast, it is crucial that memory access is minimized. We wish to (a) limit (disk) accesses to the original database, and (b) be able to directly index to the given location on disk. To handle the former, we identify the set of customers appearing in the increment database. Then for each item we create its temporary database consisting of all cid-tid pairs for customers appearing in the increment database. We further split that temporary database into the original part and the increment part, which correspond to the transactions in the current database and those in the increment database, respectively. In the best case scenario where there are no changes to the sequences, these are the only parts of the database that we need to read in.

To directly index into a database based on items and customers, we provide a two level disk-file indexing scheme. The vertical database is partitioned into a number of blocks such that each individual block fits in memory. The block lengths are a function dependent on customer identifiers. Within each block there exists an item dereferencing array, pointing to the first entry for each item. The first cid-index locates the block in which the particular customer id can be found. The second index then locates the item within the given partition. After this we perform a linear search for the given customer identifier. Using this two level indexing scheme we can quickly jump to only that portion of the database which will be affected by the update, without having to touch the entire database. Note that using a vertical data format we were able to efficiently retrieve all affected item's cids, without having to touch the entire database. However, this is not possible in the horizontal format, since a given item can appear in any transaction, which is found only by scanning the entire database.

Theory of Incremental Sequences In this section, we explore properties of the ISL. We construct two new databases, D' and D''. Let C', T' and I' be the set of all cid's, tid's and items that appear in the incremental part δ. The database D' is the collection of all records (in $D \cup \delta$) with cid in C', i.e., D' contains (cid,tid) pairs for all the customers that have been affected by the update (the pairs are from both the original and increment databases). Clearly, these are the only (cid,tid) pairs that can cause supports to change. Those customers that do not have an update need not be considered. The database D'' is $D' \setminus \delta$, i.e., D'' is the set of customers of the original database that have an update (the (cid,tid) pairs of the increment are not part of D'').

By inclusion-exclusion, the support of a sequence in *FS* or *NB* can be updated by computing its support in D' and D''. The support in the updated database is the one in the original plus the one in D' minus the one in D'' since the transactions in D'' are counted twice. More precisely, for every sequence X in FS_D or in NB_D,

$$\mathit{support}_{D+\delta}(X) = \mathit{support}_D(X) + \mathit{support}_{D'}(X) - \mathit{support}_{D''}(X). \tag{1}$$

On the other hand, for a sequence not in either *FS* or *NB*, it moves into the frequent set only if its last element is in I'. Otherwise, the sequence would not have new occurrences, and therefore would have remained outside *FS* and *NB*. Namely,

Proposition 1 *For every sequence X, if $\mathit{support}_{D+\delta}(X) > \mathit{support}_D(X)$, then the last item of X belongs to I'.*

Finally, we present the property that forms the basis of our incremental mining method. We call a sequence Y a *generating descendant* of X if there exists a list $[Z_1, Z_2, \ldots, Z_m]$ of sequences such that $Z_1 = Y$, $Z_m = X$, and for every i, $1 \leq i \leq m-1$, Z_i is a generating subsequence of Z_{i+1}. The following theorem states that if a sequence has become a member of $FS \cup NB$ in the updated database, but it was not a member before the update, then one of its generating descendants was in *NB* and now is in *FS*.

Proposition 2 *Let X be a sequence of length at least 2. If X is in $FS_{D+\delta} \cup NB_{D+\delta}$ but not in $FS_D \cup NB_D$, then X has a generating descendant in $NB_D \cap FS_{D+\delta}$.*

Proof The proof is by induction on k, the length of X. Let X_1 and X_2 be the two generating subsequences of X. Note that if both X_1 and X_2 belong to FS_D then X is in $FS_D \cup NB_D$, which contradicts our assumption. Therefore, either X_1 or X_2 is out of FS_D. For the base case, $k = 2$, since X_1 and X_2 are of length 1, by definition both belong to $FS_D \cup NB_D$, and by the above, at least one must be in NB_D. For X to be in $FS_{D+\delta} \cup NB_{D+\delta}$, X_1 and X_2 must be in $FS_{D+\delta}$ by definition. Thus the claim holds, since either X_1 or X_2 must be in $NB_D \cap FS_{D+\delta}$, and they are generating descendants of X. For the induction step, suppose $k > 2$ and that the claim holds for all $k' < k$. Suppose X_1 and X_2 are both in $FS_D \cup NB_D$. Then, either X_1 or $X_2 \in NB_D$. We know $X \in FS_{D+\delta} \cup NB_{D+\delta}$, so X_1 and X_2 belong to $FS_{D+\delta}$. Since X_1 and X_2 are generating subsequences of X, the claim holds for X. Finally, we have to consider the case where either X_1 or X_2 is not in $FS_D \cup NB_D$. We know that as $X \in FS_{D+\delta} \cup NB_{D+\delta}$, both X_1 and X_2 belong to $FS_{D+\delta} \cup NB_{D+\delta}$. Now suppose that $X_1 \notin FS_D \cup NB_D$. We know that X is in $FS_{D+\delta} \cup NB_{D+\delta}$, X_1 is in $FS_{D+\delta} \cup NB_{D+\delta}$. Therefore from the induction step (since X_1 has length less than k) the claim holds for X_1. Let Y be a generating descendant satisfying the claim for X_1. Since X_1 is a generating subsequence of X, Y is also a generating descendant of X. Thus the claim holds for k. The same argument can be applied to the case when $X_2 \notin FS_D \cup NB_D$.

4.1 Incremental Sequence Mining (ISM) Algorithm

The goal of the incremental algorithm is to maintain the sequence lattice in the presence of database updates. We categorize increments based on whether new customers are added, or whether new transactions are added. Adding new customers may reduce the support of certain sequences in the ISL. New transactions on old customers cannot cause any sequences in the ISL to be eliminated, as the number of customers is kept constant, and therefore the support requirements are the same as before. Adding a customer raises the absolute support. All we have to do is to use the new *min_sup* value and remove from the ISL any sequences that fall below the new support. After doing this, we default to the algorithm described below.

The ISL gives us the support of all the frequent set and negative border elements in the original database. The goal of Phase 1 of our algorithm is to update these supports, accounting for the incremental database. Note that by Equation (1), these can be computed from the original support values from the ISL as follows: $support_{D+\delta}(X) = support_D(X) + support_{D'}(X) - support_{D''}(X)$.

Referring to Figure 6, Phase 1 starts by computing the D' and D'' vertical id-list databases for each one-sequence. Note that Phase 1 only requires re-accessing those customers and those items from the original database that are also in the increment. Once these are generated, the lowest level of the ISL, i.e., the frequent one-item sequences, are inserted into the Phase 1 queue if the item i is in the increment set I' (from Proposition 1). $support_{D+\delta}(X)$ is then computed for each element in the queue. Note that this value has already been computed for single items. For two- and larger sequences, we compute this by intersecting the corresponding D' and D'' id-lists of its generating subsequences, and then computing $support_{D+\delta}(X)$, using the Compute_Support procedure. For a sequence p we represent these lists as $D'(p)$ and $D''(p)$.

Once the support is updated, if the sequence, p, (of length k) is in the frequent set (line 12), all length $k+1$ members of its equivalence class, if any exist, are placed at the end of the queue. If the sequence, p, is in the negative border (line 10) and its support suggests it is frequent, then this element is placed in the *NB-to-FS* hash table. There is one table for every length. This process repeats until the queue is emptied.

At the end of Phase 1, we have exact and up-to-date supports for all elements in the ISL. We further have a list of elements that were in the Negative border but have become frequent as a result of the database increment (in *NB-to-FS*). From the example in Figures 1 and 2, the following elements had supports updated: $A \to A \to A, B \to A \to A, A \to A \to B, B \to A \to B, A \to B \to B$, and C. Of these, the following moved from the negative border to the frequent set: $A \to A \to B, A \to B \to B$, and C.

```
PHASE 1:
1. compute D'_1 ... D'_m for all items i
2. compute D''_1 ... D''_m for all items i
3. for all items i in I'
4.    for all 2-sequences S in parent class [i]
5.       Q.enqueue(S);
6. while (Q is not empty)
7.    p = Q.dequeue(); Compute_Support(p);
8.    if (support(p) ≥ min_sup )
9.       k = length(p);
10.      if (p is in the negative border)
11.         NB-to-FS[k].enqueue(p);
12.      else if (D'(p) ≠ ∅)
13.         for all k+1-sequences S in class [p]
14.            Q.enqueue(S);
```

```
PHASE 2:
1. for each item i in NB-to-FS[1]
2.    construct suffix class [i];
3.    NB-to-FS[2].enqueue([i]);
4. for (k = 2 to MaxLevel)
5.    for each class C in NB-to-FS[k]
6.       Enumerate-Frequent-Seq(C);

Compute_Support(p):
1. A = generating_subsequence1(p);
2. B = generating_subsequence2(p);
3. support_D'(p) = intersect(D'(A), D'(B));
4. support_D''(p) = suffix_join(D''(A), D''(B));
5. support(p) = support(p) + support_D'(p)
   -support_D''(p);
```

Figure 6: The ISM Incremental Algorithm

We next describe Phase 2 (see Figure 6). At the end of Phase 1 the *NB-to-FS* data structure contains an array of hash tables containing elements that moved from the negative border to the frequent set. For all 1-sequences that have moved we intersect it with all possible other frequent 1-sequences. We add all such Frequent 2-sequences to the *NB-to-FS*[2] table for further processing. From our running example (Figures 1 and 2), we note that $A \to C$, and $B \to C$, get added to the *NB-to-FS*[2] table. At the same time all other evaluated two-sequences involving C that were not frequent are placed in the $NB_{D+\delta}$. Thus, $C \to A, C \to B, AC, BC$ and $C \to C$ get placed in $NB_{D+\delta}$.

Then, starting with the second level hash table, we pick an element that has not been processed and create the list of frequent sets, along with associated id-lists from $D \cup \delta$, in its equivalence class (in get_next_eqclass). The next step is to pass the resulting equivalence class to a modified version of *Enumerate-Frequent-Set*, where it adds any new Frequent or Negative Border elements and associated elements to the ISL. We repeat this process until all the *NB-to-FS* tables are empty. As an example let us consider the equivalence class associated with $A \to C$. From Figures 1 and 2 we see that the only other frequent sequence of its suffix class is $B \mapsto C$. As both the above sequences are frequent, they are placed in $FS_{D+\delta}$. Recursively enumerating the frequent itemsets results in the sequences $A \to A \to C$ and $A \to B \to C$ being added to $FS_{D+\delta}$. Similarly, the sequences $AB \to C, B \to A \to C, B \to B \to C, A \to A \to A \to C,$, and $A \to A \to B \to C$ are added to $NB_{D+\delta}$.

Correctness: In this section, we will validate our algorithm. The input to Phase 1 is the databases D' and D''. We start by inserting only items which belong to I' in the queue. By Proposition 1 for any item whose support is increased as a result of the increment database, the last item of the sequence is in I'. By placing the item i in the queue, from the recursive nature of the algorithm we ensure that all ascendents will be evaluated. Note that there is a pruning condition ($D'(p) \neq \emptyset$) that may result in certain ascendents not being evaluated. However we know that if $D'(p) \neq \emptyset$, then $D''(p) \neq \emptyset$. This ensures that all ascendents q of p will satisfy $D'(q) \neq \emptyset$, and thus, by Equation (1), we know that the support of any ascendant of p will remain unchanged. The supports themselves are updated by directly applying Equation (1). Finally the algorithm must keep track of any sequence that moved from the negative border to the frequent set which it does (if p is frequent..). The goal of Phase 2 is to update the support of all sequences that belong to $FS_{D+\delta} \cup NB_{D+\delta} - FS_D \cup NB_D$. From Proposition 2 we have that any element in this set must have a generating descendant in $NB_D \cap FS_{D+\delta}$. In Phase 1 we generate the list of all elements in $NB_D \cap FS_{D+\delta}$. For each sequence s (of length k) in the list we reconstruct the set of frequent sequences (of length k) belonging to the same length-$(k-1)$ suffix class as s does. We then apply the procedure *Enumerate-Frequent-Set* to this suffix class. This ensures that any ascendents of s will be evaluated during Phase 2.

5 Interactive Sequence Mining

The idea in interactive sequence mining is that an end user ought to be able to query the database for sequence rules using differing values of support and confidence without excessive

I/O or computation. In previous work [8, 12], multiple passes have to be made over the database for each $< support, confidence >$ pair. Hence, response times are unacceptable for online queries. The problem of sequence mining is suited to an online approach. This is because it is actually quite hard for the user to estimate how many rules might satisfy a given $< support, confidence >$ pair. Typically, one might be interested in only a few rules. Further, one might be interested only in rules involving a few items. This involves a lot of manual tuning that may play havoc with the memory subsystem on the server.

Our approach to the problem of supporting such queries efficiently is to adapt the Increment Sequence Lattice. Ideally, all interactive queries are performed on this lattice without having to rescan the database. The preprocessing step of the algorithm involves computing the lattice for a small enough support S_{min}, such that all future queries will involve a support larger than S_{min}. In order to handle certain queries, we modify the lattice to allow links from a k-length sequence to all its k subsequences of length $k-1$ (rather than just its generating subsequences). It is then easy to see that if a directed path exists from a vertex V to a vertex U then $V \subset U$. Further, each node in the lattice is weighted by the support of the sequence it represents. Some important properties of the sequence lattice include: 1) It is a directed acyclic graph, 2) $support(X) >= support(Y)$ for all connected X, Y, where X is a $(k-1)$ length sequence, and Y is a k length sequence. Given the above lattice and its properties, we can produce answers to the following queries at interactive speeds.

1) **Simple queries:** We want to answer what are the frequent sequences for support $x\%$, where $x > S_{min}$? We maintain a queue of unprocessed nodes, initialized with the root of the lattice. At every stage, we pop the first element and place all the $(k+1)$ length sequences that it is connected to and that satisfy the query support into the queue, ensuring that the same node is not placed in the queue multiple times. By property 2 above all nodes satisfying $x > S_{min}$ are placed on the queue. **Refined queries** where the support value is modified ($x+y$ or $x-y$) involves the same procedure.

2) **Quantified queries:** The goal here is to find the k most frequent sequences or the support at which we return exactly k sequences. To handle this case our queue becomes a priority queue, prioritized on the weight (support) of the node. As soon as we find the k most significant sequences, we stop. The property of the lattice ensures correctness.

3) **Including queries:** The task is to return only those sequences containing certain items. These can be handled in a manner similar to simple queries. Instead of starting at the root of the lattice, we initialize the queue with the nodes corresponding to the sequences containing

all the items $i_1, \ldots, i_n$ for "and" (conjunctive) inclusions. For "or" (disjunctive) inclusions, we initialize the queue with the set of nodes that contain *any* of the items $i_1, \ldots, i_n$. The rest of the algorithm is identical to the one for simple queries.

4) **Excluding queries:** We want to be able to exclude certain items. To handle such queries, we make two traversals over the lattice space. Over the first traversal, we follow the algorithm for Including queries, but simply mark all nodes that are placed in the queue as having been visited. The next traversal starts at the root node and follows the Simple queries traversal, except that it will not place nodes that have been marked visited by the previous traversal. This ensures that we will have only those sequences excluding the items $i_1, \ldots, i_n$. [1] We can handle "or" exclusions wherein no sequences with any of the items $i_1, \ldots, i_n$ is generated. We do this by placing the nodes corresponding to $i_1, \ldots, i_n$, individually in the queue at the start of the first traversal. Alternately if the exclusion is an "and", we initialize the queue with the node corresponding to sequences containing all of $i_1, \ldots, i_n$.

5)**Hierarchical queries:** The goal here is to treat a set of related items as one super-item. For example we may want to treat chips, cookies, peanuts, etc. all together as a single item called "snacks". We would like to know what are the frequent sequences involving this super-item. To generate the resulting sequences, we have to modify the SPADE algorithm. We reconstruct the id-list for the new item $(i_1, \ldots, i_n)$ via a special union operator, and we remove from consideration the individual items $i_1, \ldots, i_n$. Then, we rerun the equivalence class algorithm for this new item and return the set of frequent sequences.

6 Experimental Evaluation

All the experiments were on a single processor of a DECStation 4100. The DECStation 4100 contains 4 600MHz Alpha 21164 processors, with 256 MB of memory. No other user processes were running at the time. We used different synthetic databases with size ranging from 20MB to 55MB, which were generated using the procedure described in [8]. Although the size of our benchmark databases fit in memory, our goal is to work with out-of-core databases. Hence, we assume that the database resides on disk.

The datasets are generated using the following process. First N_I maximal itemsets of average size I are generated by choosing from N items. Then N_S maximal sequences of

[1] We note that an alternative approach could be to filter nodes and have only one traversal. However using two traversals is found to be more efficient.

average size S are created by assigning itemsets from N_I to each sequence. Next, a customer of average T transactions is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T. The generation stops when C customers have been generated. Table 1 shows the databases used and their properties. The total number of transactions is denoted as $|\mathcal{D}|$, average transaction size per customer as T, and the total number of customers C. The parameters we used for database generation were $N = 1000$, $N_I = 25000$, $I = 1.25$, $N_S = 5000$, $S = 4$.

Database	C	T	$\mathcal{D}$	Total Size
C100.T10	100000	10	1000,000	20 MB
C100.T12	100000	12	1200,000	25 MB
C100.T15	100000	15	1500,000	30 MB
C150.T10	150000	10	1500,000	31 MB
C200.T10	200000	10	2000,000	42 MB
C250.T10	250000	10	2500,000	55 MB

Table 1: Database properties

To evaluate the incremental algorithm, we modified the database generation mechanism to construct two datasets — one corresponding to the original database, and one corresponding to the increment database. The input to the generator also included an increment percentage roughly corresponding to the number of customers in the increment and the percentage of transactions for each such customer that belongs in the increment database. Assuming the database being looked at is C100.T10, if we set the increment percentage to 5% and the percentage of transactions to 20%, then we could expect 5000 customers (5% of 100,000) to belong to C', each of which would contain on average two transactions (20% of 10) in the increment database. The actual number of customers in the increment is determined by drawing from a uniform distribution (increment percentage as parameter). Similarly, for each customer in the increment the number of transactions belonging to the increment is also drawn from a uniform distribution (transaction percentage as parameter).

Incremental Performance: For the first experiment, we varied the increment percentage for 4 databases while fixing the transaction percentage to 20%. We ran the SPADE algorithm on the entire database (original and increment) combined, and evaluated the cost of running just the incremental algorithm (after constructing the ISL from the original database) for increment database values of five, three and one percent. For each database, we also evaluated the breakdown of the cost of the incremental algorithm phases. The results show that the speedup obtained by using the incremental algorithm in comparison to re-running

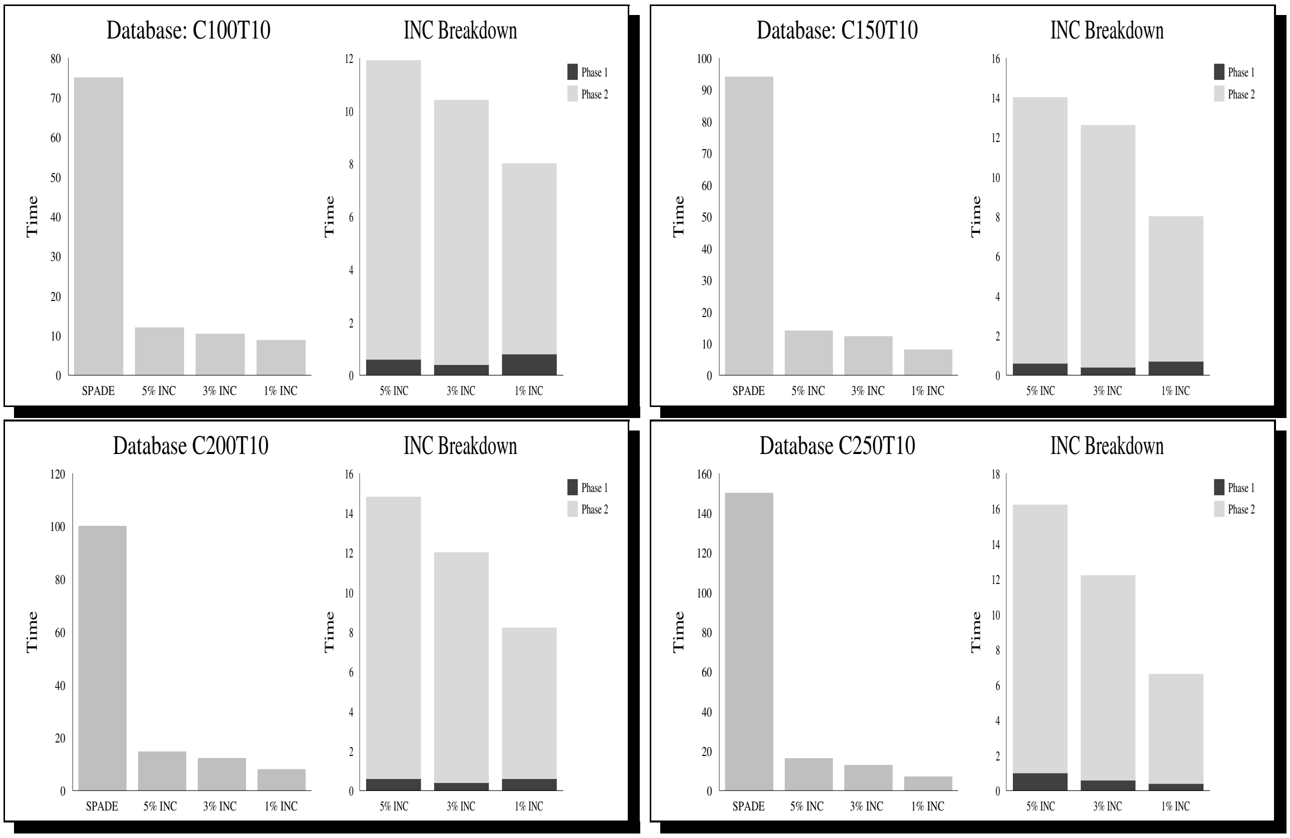

Figure 7: Incremental Algorithm Performance Comparison

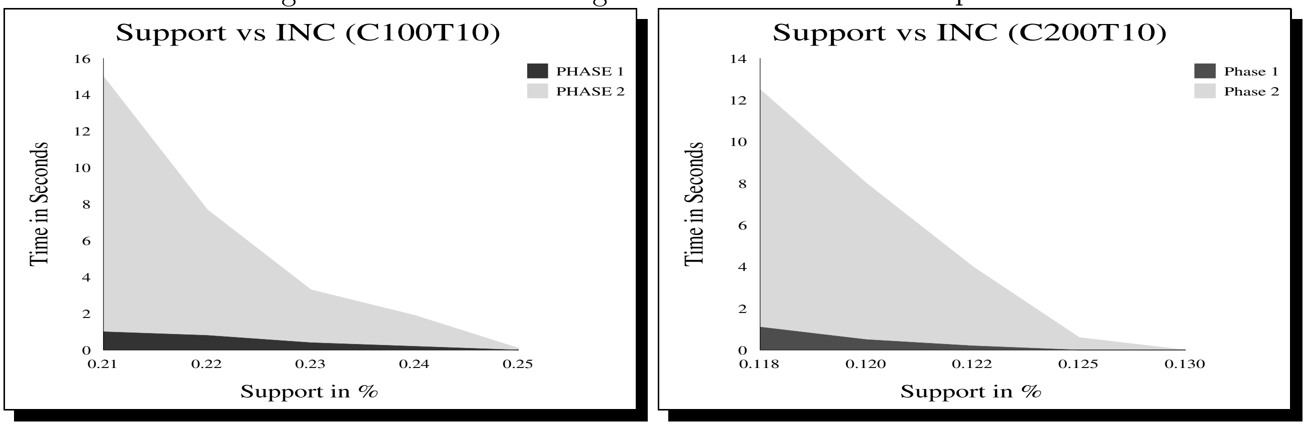

Figure 8: Effect of Varying Support: C100.T10 and C200.T10

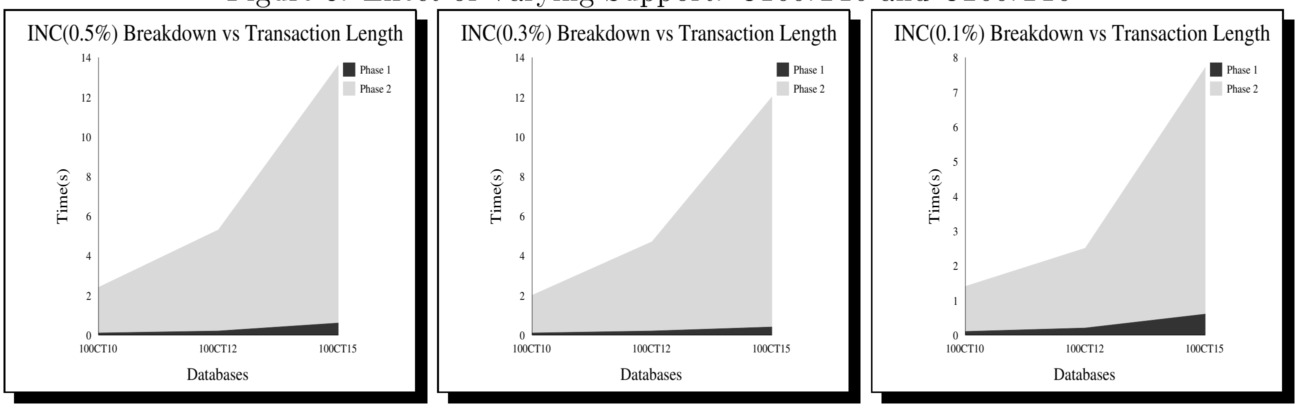

Figure 9: Effect of Varying Transaction Length

the SPADE algorithm over the entire database range from a factor of 7 to over two orders of magnitude. As expected, on moving from a larger increment value to a smaller one, the improvements increase, since there are fewer new sequences from a smaller increment.

The breakdown figures reveal that the phase one time is pretty negligible, requiring under 1 second for all the datasets for all increment values. It also shows that the phase two times, while an order of magnitude larger than the phase one times, are still much faster than re-executing the entire algorithm. Further, while increasing database size does increase the overall running time of phase 2, it does not increase at the same rate as re-executing the entire algorithm for these datasets.

The second experiment we conducted was to vary the support sizes for a given increment size (1%), and for two databases. The results for this experiment are documented in Figure 8. For both databases, as the support size is increased, the execution time of phase 1 and phase 2 rapidly approaches 0. This is not surprising when you consider that at higher supports, the number of elements in the ISL are fewer (affecting phase 1) and the number of new sequences are much smaller (affecting phase 2).

The third experiment we conducted was to keep the support, the number of customers, and the transaction percentage constant (0.24%, 100,000, and 20% respectively), and vary the number of transactions per customer (10, 12, and 15). Figure 9 depicts the breakdown of the two phases of the *ISM* algorithm for varying increment values. We see that moving from 10 to 15 transactions per customer, the execution time of both phases progressively increases for all database increment sizes. This is because the number of sequences in the ISL are more (affecting phase1) and the number of new sequences are also more (affecting phase2). The relative increase is smaller for lower increment values.

Database	Simple	Refined(0.005)	Priority(50)	Including	Excluding	Rerunning SPADE
C100K.T10	0.06	0.011	1.11	negligible	0.07	75
C100K.T12	0.06	0.014	1.08	negligible	0.06	82
C100K.T15	0.08	0.026	1.90	negligible	0.08	90
C150K.T10	0.085	0.022	2.48	negligible	0.09	94
C200K.T10	0.08	0.022	1.5	negligible	0.08	102
C250K.T10	0.044	0.02	1.28	negligible	0.16	150

Table 2: Interactive Performance

Interactive Performance: In this section, we evaluate the performance of the interactive queries described in Section 5. All the interactive query experiments were performed on a SUN UltraSparc, 167MHz processor with 256 MB of memory. We envisage off-loading the interactive querying feature onto client machines as opposed to executing on the server, and shipping the results to the data mining client. Thus we wanted to compare executing interactive queries on a slower machine. Another reason for evaluating the queries on a slower machine is that the relative speeds of the various interactive approaches is better seen on a slower machine (on the DECs all queries executed in negligible time).

Since hierarchical queries simply entail a modified execution of phase 2, we do not evaluate it again. We evaluated simple querying on supports ranging from 0.1%-0.25%, refined querying (support refined to 0.5% for all the datasets), priority querying (querying for the 50 sequences with highest support), including queries (including a random item) and excluding queries (excluding a random item). Results are presented in Table 2 along with the cost of rerunning the SPADE algorithm on the DEC machine. We see that the querying time for refined, priority, including and excluding queries are very low and capable of achieving interactive speeds. The priority query takes more time, since it has to sort the sequences according to support value, and this sorting dominates the computation time. Comparing with rerunning SPADE (on a much faster DEC machine) we see that the interactive querying is several orders of magnitude faster, in spite of executing it on a much slower machine.

7 Related Work

Sequence Mining Algorithms: GSP [8] is one of the best previous algorithms. Recently, SPADE [12] was shown to outperform GSP by a factor of two in the general case, and by a factor of ten with a pre-processing step. The problem of finding *frequent episodes* in a single long sequence of events was presented in [5]. The MEDD and MSDD algorithms [7] discover patterns in multiple event sequences; they explore the rule space directly instead of the sequence space.

Incremental Sequence Mining: There has been almost no work addressing the incremental mining of sequences. One related proposal in [11] uses a dynamic suffix tree based approach to incremental mining in a single long sequence. However, we are dealing with sequences across different customers, i.e., multiple sequences of sets of items as opposed to a single long sequence of items. The other closest work is in incremental association mining. In [2], the FUP algorithm is presented. A major limitation of FUP is that it may require $O(k)$ database (original plus increment) scans, where k is the size of the largest frequent itemset. In [3] two incremental algorithms were presented – the *Pairs* approach stores the set of frequent 2-sequences, while the *Borders* algorithm keeps track of the frequent set and the negative border. An approach very similar to the Borders algorithm was also proposed in [10]. These approaches may require only a single scan of the original data, but they still make $O(k)$ scans of the increment database. In contrast, our new *ISM* algorithm makes only one scan of the incremental database, and may make a partial scan of the original database if needed.

Although we have cited related work in association mining, there are important differences. While association rules discover only intra-transaction patterns (itemsets), we now also have to discover inter-transaction patterns (sequences). The set of all frequent sequences is a superset of the set of frequent itemsets. In fact the itemset space is a very tiny fraction of the sequence space. If n is the number of items, then the itemset space is of size 2^n regardless of the database size while the size is not bounded for the sequence space: if k is the maximum sequence length then the space size is $O(n^k)$. So sequence search is much more complex and challenging than the itemset search thereby further necessitating fast algorithms.

Interactive Sequence Mining: A mine-and-examine paradigm for interactive exploration of associations and sequence episodes was presented in [4]. The idea is to mine and produce a large collection of frequent patterns. The user can then explore this collection by the use of *templates* specifying what's interesting and what's not. They only consider inclusive and exclusive templates. A second approach to exploratory analysis is to integrate the constraint checking inside the mining algorithm. One approach was presented in [9]. Recently, [6] presented the CAP algorithm for extracting all frequent associations matching a rich class of constraints. This piece of work is complimentary to ours. Furthermore, our approach tries to integrate the incremental and interactive components (i.e., we allow new transactions as well as changes in support and constraints on items), and is more suited to the mine-and-examine framework. An online algorithm for mining associations at different values of support and confidence was presented in [1]. Like their approach, *ISM* uses a lattice framework. However, *ISM* supports both online and incremental mining, and works in the sequence data space.

8 Conclusions

In this paper, we propose novel techniques that maintain data structures for mining sequences in the presence of a) database updates, and b) user interaction. Results obtained show speedups from several factors to two orders of magnitude for incremental mining when compared with re-executing a state of the art sequence mining algorithm. Results for interactive approaches are even better. At the cost of maintaining a summary data structure, the interactive approach performs several orders of magnitude faster than any current sequence mining algorithm. One of the limitations of the incremental approach proposed in this paper is the size of the negative border, and the resulting memory utilization. We are currently investigating methods to eliminate part or all of this problem, either by refining the algorithm or by performing out of core computation.

References

[1] C. Aggarwal and P. Yu. Online generation of associations. In *14th ICDE Conf.*, Feb. 1998.

[2] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: an incremental updating technique. In *12th ICDE Conf.*, February 1996.

[3] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *2nd SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[4] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *3rd Intl. Conf. Information and Knowledge Management*, November 1994.

[5] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery: An International Journal*, 1(3):259–289, 1997.

[6] R. T. Ng, et al. Exploratory mining and pruning optimizations of constrained association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

[7] T. Oates, et al. A family of algorithms for finding temporal structure in data. In *6th Intl. Workshop on AI and Statistics*, March 1997.

[8] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.

[9] R. Srikant, Q. Vu, and R. Agrawal. Mining Association Rules with Item Constraints. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, August 1997.

[10] S. Thomas, S. Bodgala, K. Alsabti, and S. Ranka. An efficient algorithm for incremental updation of association rules in large databases. In *3rd KDD Conf.*, August 1997.

[11] K. Wang. Discovering patterns from large and dynamic sequential data. *J. Intelligent Information Systems*, 9(1), August 1997.

[12] M. J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.