Sequence Mining in Dynamic and Interactive Environments *

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 execution time improvements of up to several orders of magnitude in practice.

1 Introduction

Dynamism and interactivity are essential features of all human endeavors, be they in a scientific or business enterprise. The collection of information and data in various fields serves as an exemplar, where every moment we are faced with new content (dynamism) and are required to manipulate it (interactivity). For example, consider a large retail store like Walmart, which has a data-warehouse more than a terabyte in size. In addition, Walmart collects approximately 20 million customer transactions every day. It is simply infeasible to mine the entire database (the original terabyte data-warehouse, and the new transactions) each time an update occurs. As another example consider

*This work was supported in part by NSF grants CDA–9401142, CCR–9702466, CCR–9705594, CCR–9701911, CCR–9725021, INT-9726724, and a DARPA grant F30602-98-2-0133; and an external research grant from Digital Equipment Corporation.

Web Mining. Let's assume we have mined interesting browsing patterns at a popular portal site like Yahoo! that receives millions of hits every day. Once again it is not practical to re-mine the site logs each time an update occurs.

Given the inherently dynamic nature of data collection, it is somewhat surprising that incremental techniques have received little to no attention within knowledge discovery and data mining. It is worth noting that without incrementality, interactivity also remains a distant goal. True interactivity is not possible if once is forced to re-mine the entire database from scratch each time. This paper seeks to address the problem of mining frequent sequences in dynamic and interactive environments. For example, incremental updates of the most frequent sequence of items purchased by customers, or real-time mining of browsing patterns (i.e., sequences of web-pages) on the Internet.

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 retail industry (e.g. the Walmart example from above), including attached mailing, add-on sales and customer satisfaction. Besides the retail and Internet examples, it applies to many other 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, many domains such as electronic commerce, stock analysis, collaborative surgery, etc., impose soft real-time constraints on the mining process. In such domains, where the databases are updated on a regular basis and user interactions modify the search parameters, running the discovery program all over again is infeasible. Hence, there is a need for algorithms

that maintain valid mined information across i) database updates, and ii) user interactions (modifying/constraining the search space).

In this paper, we present a method for incremental and interactive 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 up to 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.

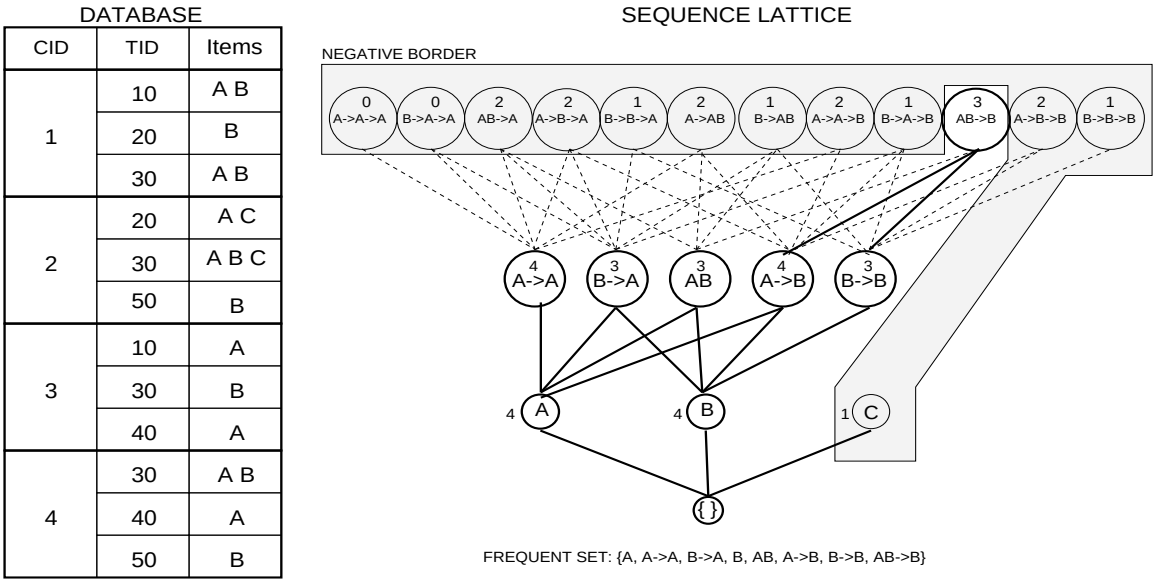
2 Problem Formulation

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. 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 (ordered in time) of non-empty itemsets. A sequence of itemsets $\alpha_1, \ldots, \alpha_n$ is denoted by $(\alpha_1 \rightarrow \cdots \rightarrow \alpha_n)$. 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 $\emptyset \rightarrow$ and $\rightarrow \emptyset$ one at a time. For example, $B \rightarrow AC$ is a subsequence of $AB \rightarrow E \rightarrow ACD$. 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 $AB \rightarrow CD \rightarrow E$ are $A \rightarrow CD \rightarrow E$ and $B \rightarrow CD \rightarrow E$, and they share the common suffix $CD \rightarrow E$. A sequence is *maximal* in a collection $\mathcal{C}$ of sequences if the sequence is not a subsequence of any other sequence in $\mathcal{C}$.

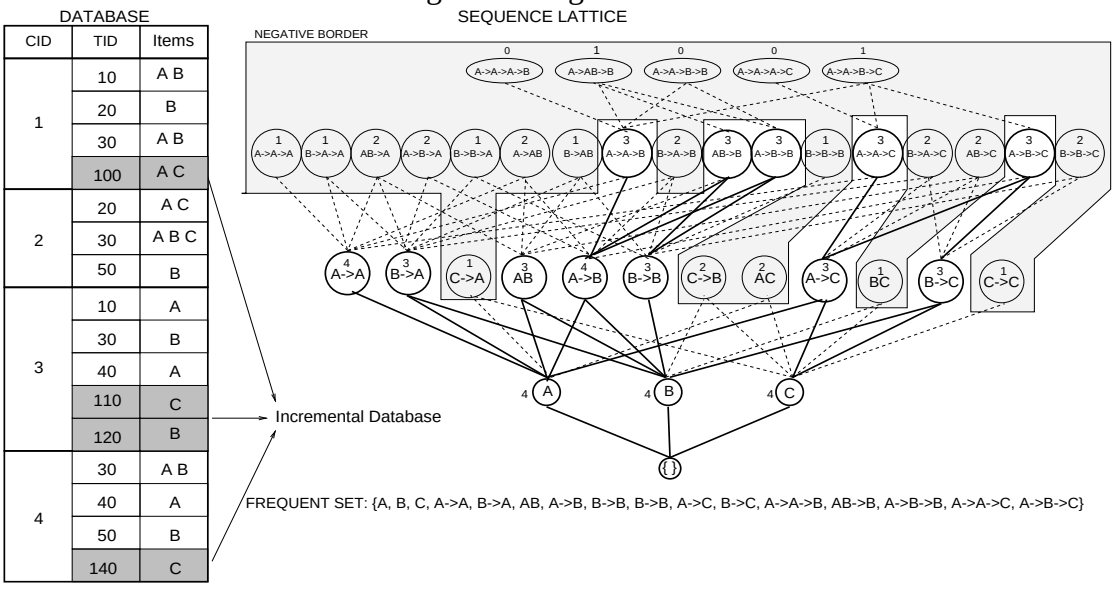
Our database is a collection of *customers*, each with a sequence of *transactions*, each of which is an itemset. For a database D and a sequence α, the *support* or *frequency* of α in D, denoted by $support_D(\alpha)$, is the number of customers in D whose sequences contain α as a subsequence. The *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 $A \Rightarrow B$ 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 δ is to be added to a database D. Then we call D the *original database* and δ 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 in the updated database. Also FS denotes the set of all frequent sequences in the updated database. 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.



CID	TID	Items
1	10	A B
	20	B
	30	A B
2	20	A C
	30	A B C
	50	B
3	10	A
	30	B
	40	A
4	30	A B
	40	A
	50	B

Figure 1: Original Database



CID	TID	Items
1	10	A B
	20	B
	30	A B
	100	A C
2	20	A C
	30	A B C
	50	B
3	10	A
	30	B
	40	A
	110	C
	120	B
4	30	A B
	40	A
	50	B
	140	C

Figure 2: Original plus Incremental Database and Lattice

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 the Increment Sequence Lattice (ISL) with 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 (highlighted in dark grey). For example, C is not frequent in the original database D, but C (along with some of its super-sequences) becomes frequent after the update δ. 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.

Some comments are in order to see the generality of our problem formulation: 1) We discover sequences of *subsets* of items, and not just single item sequences. For example, the itemset CD in $(CD \rightarrow E)$. 2) We discover sequences with arbitrary *gaps* among events, and not just the consecutive subsequences. For example, the sequence $(A \rightarrow A)$ is a subsequence of customer 1 (see Figure 1), even though there is an intervening transaction. The sequence symbol $\rightarrow$ simply denotes a *happens-after* relationship. 3) Our formulation is general enough to encompass almost any categorical sequential domain. For example, if the input-sequences are DNA strings, then an event consists of a single item (one of A, C, G, T). If input-sequences represent text documents, then each word (along with any other attributes of that word, e.g., noun, position, etc.) would comprise an event. Even continuous domains can be represented after a suitable discretization step.

3 The SPADE Algorithm

In this section we describe SPADE [11], 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, 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 (single items) to the most specific frequent sequences (maximal sequences) in a depth-first manner. For instance, in Figure 1, the bold lines correspond to the lattice for the example dataset.

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 item. For example, the idlist for the item C in the original database (Figure 1) would consist of the tuples $\{< 2, 20 >, < 2, 30 >\}$.

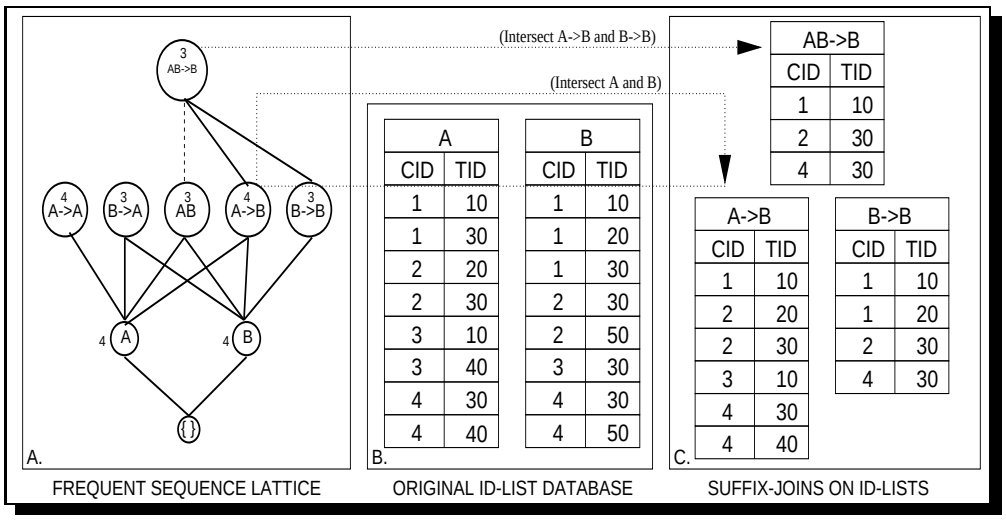


Figure 3: A. Frequent Sequence Lattice; B. Initial Idlist Database; C. idlist Intersections

Given the per item idlists, we can iteratively determine the support of any k-sequence by performing a temporal join on the idlists of its two generating sequences (i.e., its $(k-1)$ length subsequences that share a common suffix sequences). A simple check on the support (i.e., the number of distinct cids) 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 $A \rightarrow B$ is obtained by a temporal join on the lists of A and B. Since the symbol $\rightarrow$ represents a temporal relationship, we find all occurrences of A before a B in a customer's transaction sequence, and store the corresponding time-stamps or tids, to obtain $\mathcal{L}(A \rightarrow B)$. We obtain the idlist for $AB \rightarrow B$ by intersecting the idlist of its two generating sequences, $A \rightarrow B$ and $B \rightarrow B$, but this time we are looking for *equality join*, i.e., instances where A and B co-occur before a B. Since we always join two sequences that share a common suffix, it suffices to keep track of only the tid of the first item, as the tids of the suffix remain fixed. Please see [11] for exact details on how the temporal and equality joins are implemented.

If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing intersections 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 that can be processed *independently* in memory. This is accomplished via *suffix-based equivalence classes* (henceforth denoted as a class). 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. Each suffix 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 $Y \rightarrow X$, and $Z \rightarrow X$ as the only sequences, the only possible frequent sequences at the next step can be $Y \rightarrow Z \rightarrow X$, $Z \rightarrow Y \rightarrow X$, and $(YZ) \rightarrow X$. It should be obvious that no other item Q can lead to a frequent sequence with the suffix X, unless (QX) or $Q \rightarrow X$ is also in $[X]$.

SPADE recursively decomposes the sequences at each new level into even smaller independent classes. For instance, at level one it uses suffix classes of length one (X,Y), at level two it uses suffix classes of length two (X $\rightarrow$ Y, XY) and so on. We refer to level one suffix classes as *parent* classes. These suffix classes are processed one-by-one. Figure 4 shows the pseudo-code (simplified for

BEGIN Enumerate-Frequent-Seq($[S]$**):**
  for all elements $A_i \in [S]$ **do**
    $[A_i] = \emptyset$;
    for all elements $A_j \in [S]$ **do**
      $R = A_j \cup A_i$; /*sequences R formed by generating
        subsequences A_j and A_i with A_i as a suffix*/
      $idlist(R) = idlist(A_j) \cap idlist(A_i)$;
      if $support(idlist(R)) \geq min\_sup$ **then**
        $[A_i] = [A_i] \cup \{R\}$;
  for all $[A_i] \neq \emptyset$ **do** *Enumerate-Frequent-Seq*($[A_i]$);
END Enumerate-Frequent-Seq($[S]$**):**

Figure 4: Enumerating Frequent Sequences

exposition, see [11] for exact details) for the main procedure of the SPADE algorithm. The input to the procedure is a class, along with the idlist for each of its elements. Frequent sequences are generated by intersecting [11] the idlists of all distinct pairs of sequences in each class and 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 recursively 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 [11].

4 Incremental Mining Algorithm

Our purpose is to minimize re-computation 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 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 SPADE. 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.

Theory of Incremental Sequences Let C', T' and I' be the set of all cid's, tid's and items, respectively, that appear in the incremental part δ. Define D' to be the set of all records (in $D \cup \delta$) with cid in C' and $D'' = D' \setminus \delta$. For the sake of simplicity assume that there is no new customer added to the database. This implies that infrequent sequences can become frequent but not the other way around. We use the following properties of the lattice to efficiently perform incremental sequence mining.

By set inclusion-exclusion, we can update the support of a sequence in FS or NB based on its support in D' and D''.

$$\begin{aligned} & support_{D+\delta}(X) \\ = \; & support_D(X) + support_{D'}(X) - support_{D''}(X). \end{aligned} \tag{1}$$

This allows us to compute the support at any node in the lattice quickly, by limiting re-access to the original database to D''.

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

This allows us to limit the nodes in the ISL that are re-examined to those with descendants in I'.

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}. We show that 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 $X_2 \notin FS_D \cup NB_D$. ∎

Proposition 2 limits the additional sequences (not found in the original ISL) that need to be examined to update the ISL.

Memory Management SPADE simply requires per item idlists. For incremental mining, in order to limit accesses to customers and items in the increment, we use a two level disk-file indexing scheme. However, since the number of customers is unbounded, we use a hashing mechanism described below.

The vertical database is partitioned into a number of blocks such that each individual block fits in memory. Each block contains the vertical representation of all transactions involving a set of customers. Within each block there exists an item dereferencing array, pointing to the first entry for each item. Given a customer, and an item, we first identify the block containing the customer's transactions using a first level cid-index (hash function). The second item-index then locates the item within the given block. After this we perform a linear search for the exact 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. This is not possible in the horizontal format, since a given item can appear in any transaction, which is found by scanning the entire data.

PHASE 1:
1. compute $D'(i), D''(i)$ for all items i
2. **for** all items i in I'
3. Q.enqueue(S);
4. **while** (Q is not empty)
5. p = Q.dequeue(); Compute_Support(p);
6. **if** (*support*(p) $\geq$ *min_sup*)
7. $k = length(p)$;
8. **if** (p is in the negative border)
9. *NB-to-FS*[k].enqueue(p);
10. **else if** ($D'(p) \neq \emptyset$)
11. for all $k+1$-sequences S in ISL that are
12. generating ascendents of p
13. 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 ...)
5. **for** each class $\mathcal{C}$ in *NB-to-FS*[k]
6. *Enumerate-Frequent-Seq*($\mathcal{C}$);

Compute_Support(p):
1. A = generating_subsequence1(p);
2. B = generating_subsequence2(p);
3. $support_{D'}(p)$ = intersect($D'(A), D'(B)$);
4. $support_{D''}(p)$ = intersect($D''(A), D''(B)$);
5. $support(p) = support(p) + support_{D'}(p) - support_{D''}(p)$;

Figure 5: The ISM Algorithm

Incremental Sequence Mining (ISM) Algorithm Our incremental algorithm maintains the incremental sequence lattice, ISL, which consists of all the frequent sequences and all sequences in the negative border in the original database. The support of each member is kept in the lattice, too. There are two properties of increments we are concerned with: whether new customers are added

and whether new transactions are added. We first check whether a new customer is added. If so, the minimum support in terms of the number of transactions is raised. We examine the entire ISL from the 1-sequences towards longer and longer sequences to compute where each sequence belongs. More precisely, for each sequence X that has been reclassified from frequent to infrequent, if its two generating sequences are still frequent we make X as a negative border element; otherwise, X is eliminated from the lattice. Then we default to the algorithm described below (see Figure 5).

The algorithm consists of two phases. Phase 1 is for updating the supports of elements in NB and FS and Phase 2 is for adding to NB and FS beyond what was done in Phase 1. To describe the algorithm, for a sequence p we represent by $D'(p)$ and $D''(p)$ the vertical id-list of p in D' and that in D'', respectively.

Phase 1 begins by generating the single item components of ISL: For each single item sequences p, we compute $D'(p)$ and $D''(p)$ and put p into a queue Q, which is empty at the beginning of computation. Then we repeat the following until Q is empty: We dequeue one element p from Q. We update the support of p using the subroutine Compute_Support, which computes the support based on Equation 1. Once the support is updated, if the sequence p (of length k) is in the frequent set (line 10), all length $k+1$ sequences that are already in ISL and that are generating ascendents of p are queued into Q. If the sequence, p, is in the negative border (line 8) and its support suggests it is frequent, then this element is placed in *NB-to-FS*$[k]$.

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*). In the example in Figures 1 and 2, the following elements had supports updated: $A \rightarrow A \rightarrow A, B \rightarrow A \rightarrow A, A \rightarrow A \rightarrow B, B \rightarrow A \rightarrow B, A \rightarrow B \rightarrow B$, and C. Of these, the following moved from the negative border to the frequent set: $A \rightarrow A \rightarrow B, A \rightarrow B \rightarrow B$, and C.

We next describe Phase 2 (see Figure 5). As to Phase 1, at the end of Phase 1 the *NB-to-FS*

is a list (or an array) of hash tables containing elements that have moved from NB to FS. By Proposition 2 these are the only suffix-based classes we need to examine. For all 1-sequences that have moved we intersect it with all possible other frequent 1-sequences. We add all such frequent 2-sequences into the queue *NB-to-FS*[2] for further processing. In our running example in Figures 1 and 2, $A \rightarrow C$ and $B \rightarrow C$ are 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 $NB_{D+\delta}$. Thus, $C \rightarrow A, C \rightarrow B, AC, BC$ and $C \rightarrow C$ are placed in $NB_{D+\delta}$. The next step in Phase 2 is to, starting with the hash table containing length two sequences, 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. The next step is to pass the resulting equivalence class to *Enumerate-Frequent-Set*, which adds any new frequent sequences or new negative border elements and associated elements to the ISL. We repeat this until all the *NB-to-FS* tables are empty. As an example, let us consider the equivalence class associated with $A \rightarrow C$. From Figures 1 and 2 we see that the only other frequent sequence of its suffix class is $B \rightarrow C$. As both the above sequences are frequent, they are placed in $FS_{D+\delta}$. Recursively enumerating the frequent itemsets results in the sequences $A \rightarrow A \rightarrow C$ and $A \rightarrow B \rightarrow C$ being added to $FS_{D+\delta}$. Similarly, the sequences $AB \rightarrow C, B \rightarrow A \rightarrow C, B \rightarrow B \rightarrow C, A \rightarrow A \rightarrow A \rightarrow C$,, and $A \rightarrow A \rightarrow B \rightarrow C$ are added to $NB_{D+\delta}$.

5 Interactive Sequence Mining

The idea in interactive sequence mining is that an end user be allowed to query the database for association rules at differing values of support and confidence. The goal is to allow such interaction without excessive I/O or computation. Interactive usage of the system normally involves a lot of manual tuning of parameters and re-submission of queries that may be very demanding on the memory subsystem of the server. In most current algorithms, multiple passes have to be made over the database for each $< support, confidence >$ pair. This leads to unacceptable response times

for online queries. Our approach to the problem of supporting such queries efficiently is to create pre-processed summaries that can quickly respond to such online queries.

A typical set of queries that such a system could support include:

i) **Simple Queries:** identify the rules for support x%, confidence y%.

ii) **Refined queries:** where the support value is modified ($x + y$ or $x - y$) involves the same procedure.

iii) **Quantified Queries:** identify the k most important rules in terms of support, confidence pairs or find out for what support/confidence values can we generate exactly k rules.

iv) **Including Queries:** find the rules including itemsets $i_1, \ldots, i_n$.

v) **Excluding Queries:** compute the rules excluding itemsets $i_1, \ldots, i_n$.

vi) **Hierarchical Queries:** treat items $i_1(coke), \ldots, i_n$ (pepsi), as one item (cola) and return the new rules.

Our approach to the problem of supporting such queries efficiently is to adapt the Increment Sequence Lattice. The preprocessing step of the algorithm involves computing such a lattice for a small enough support S_{min}, such that all future queries will involve a support S larger than S_{min}. In order to handle certain queries (**Including, Excluding** etc.), 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). Given such a lattice, we can produce answers to all but one (**Hierarchical queries**) of the queries described in the previous section at interactive speeds without going back to the original database. This is easy to see as all of the queries will basically involve a form of pruning over the lattice. A lattice, as opposed to a flat file containing the relevant sequences, is an important data structure as it permits rapid pruning of relevant sequences. Exactly how we do this is discussed in more detail in [7].

Hierarchical queries require the algorithm 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 using a maximum of 256 MB of physical memory. The DECStation 4100 contains 4 600MHz Alpha 21164 processors. 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 itemsets of average size I are generated by choosing from N items. Then N_S sequences of average length 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 were $N = 1000$, $N_I = 25000$, $I = 1.25$, $N_S = 5000$, $S = 4$. Please see [8] for further details on the dataset generation process.

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

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

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).

Database	Simple	Refined(0.005)	Priority(50)	Including	Excluding	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: Time in Seconds

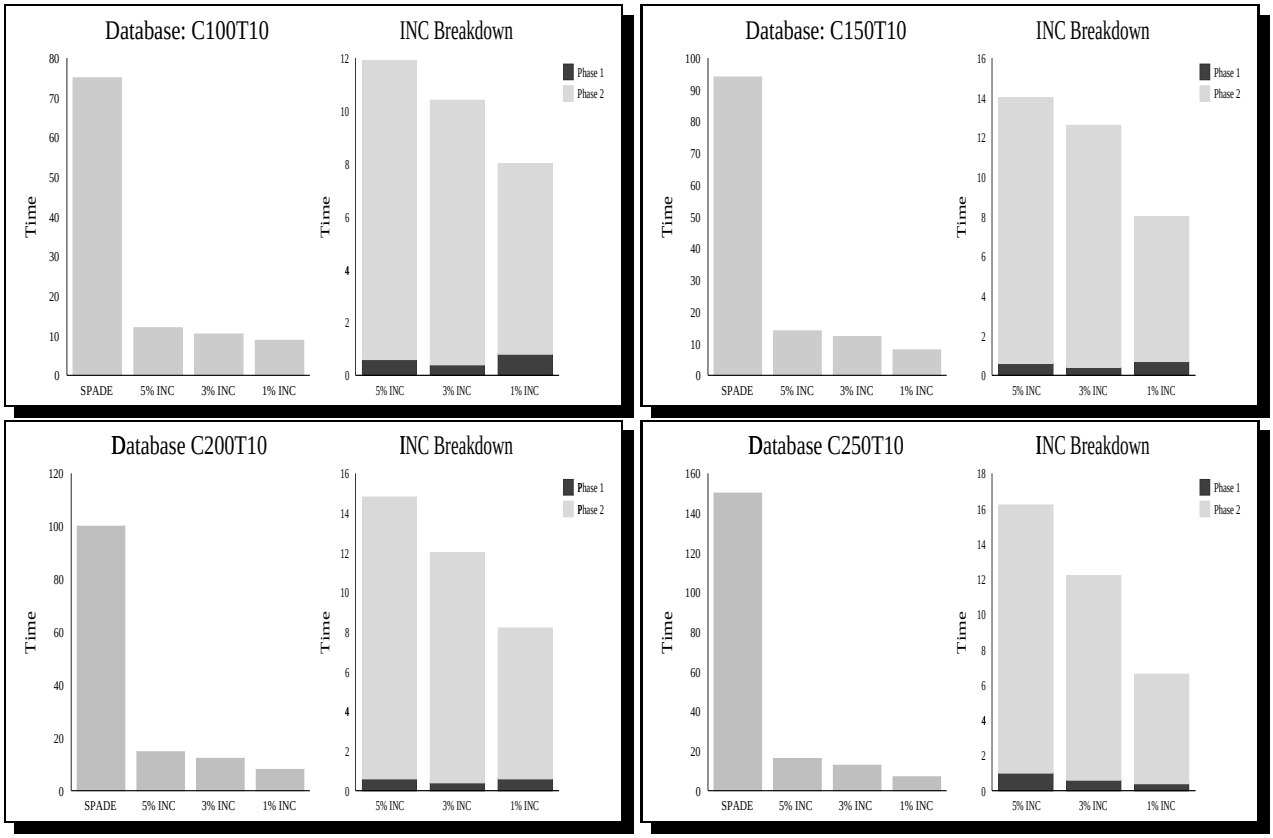


Figure 6: Incremental Algorithm Performance Comparison (Time in Seconds)

Incremental Performance: For the first experiment (see Figure 6), 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 speedups obtained by using the incremental algorithm in comparison to re-running 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

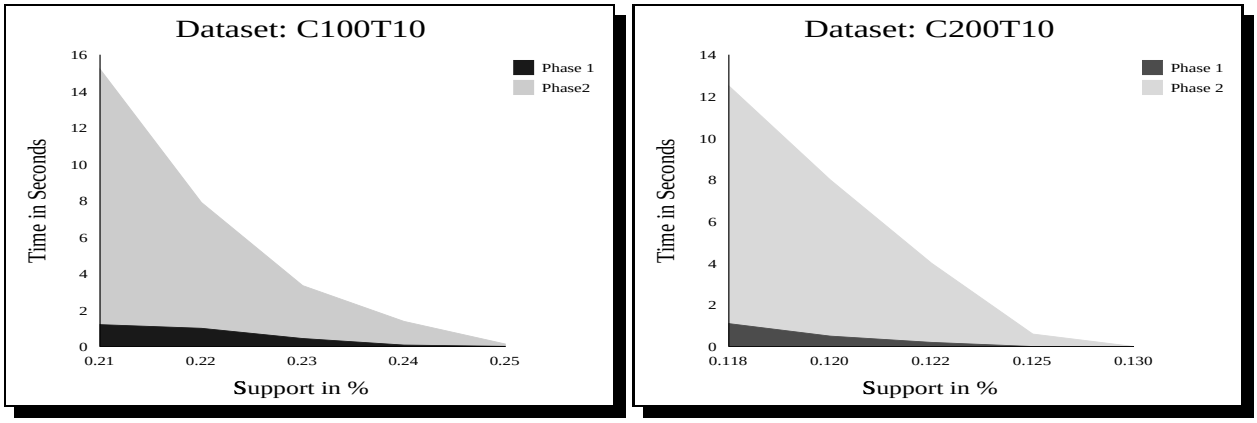


Figure 7: Effect of Varying Support: C100.T10 and C200.T10

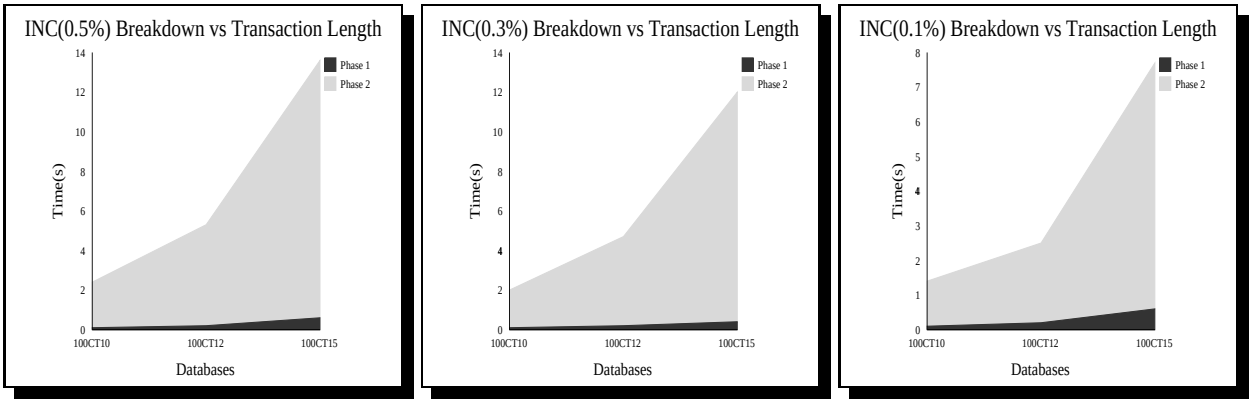


Figure 8: Effect of Varying Transaction Length

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 7. 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 8 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).

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 queries 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: The concept of sequence mining as defined in this paper was first described in [8]. Recently, SPADE [11] was shown to outperform the algorithm presented in [8] 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 problem of discovering patterns in multiple event sequences was studied in [6]; they search the rule space directly instead of searching the sequence space and then forming the rules.

Incremental Sequence Mining: There has been almost no work addressing the incremental mining of sequences. One related proposal in [10] 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 [2, 3, 9] However, there are important differences that make incremental sequence mining a more difficult problem. 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 an unbounded superset of the set of frequent itemsets (bounded). Hence, 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]. Similar paradigms have been proposed exclusively for associations [1]. Our interactive approach tackles a different problem (sequences across different customers) and supports a wider range of interactive querying features.

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 alleviate this problem, either by refining the algorithm or by performing out-of-core computation.

References

[1] C. Aggarwal and P. Yu. Online generation of association rules. In *14th Intl. Conf. on Data Engineering*, February 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 IEEE Intl. Conf. on Data Engineering*, February 1996.

[3] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *2rd ACM 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*, pages 401–407, 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] T. Oates, M. D. Schmill, D. Jensen, and P. R. Cohen. A family of algorithms for finding temporal structure in data. In *6th Intl. Workshop on AI and Statistics*, March 1997.

[7] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *8th Intl. Conf. Information and Knowledge Management*, November 1999. Expanded version available as Technical Report 715, U. of Rochester, June 1999.

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

[9] S. Thomas, S. Bodgala, K. Alsabti, and S. Ranka. An efficient algorithm for incremental updation of association rules in large databases. In *3rd Intl. Conf. Knowledge Discovery and Data Mining*, August 1997.

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

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