Efficient Enumeration of Frequent Sequences *

Mohammed J. Zaki
Computer Science Department, Rensselaer Polytechnic Institute, Troy NY 12180

Abstract

In this paper we present SPADE, a new algorithm for fast discovery of *Sequential Patterns*. The existing solutions to this problem make repeated database scans, and use complex hash structures which have poor locality. SPADE utilizes combinatorial properties to decompose the original problem into smaller sub-problems, that can be independently solved in main-memory using efficient lattice search techniques, and using simple join operations. All sequences are discovered in only three database scans. Experiments show that SPADE outperforms the best previous algorithm by a factor of two, and by an order of magnitude with some pre-processed data. It also has linear scalability with respect to the number of customers, and a number of other database parameters.

1 Introduction

The sequence mining task is to discover a set of attributes, shared across time among a large number of objects in a given database. For example, consider the sales database of a bookstore, where the objects represent customers and the attributes represent authors or books. Let's say that the database records the books bought by each customer over a period of time. The discovered patterns are the sequences of books most frequently bought by the customers. An example could be that "70% of the people who buy Jane Austen's *Pride and Prejudice* also buy *Emma* within a month." Stores can use these patterns for promotions, shelf placement, etc. Consider another example of a web access database at a popular site, where an object is a web user and an attribute is a web page. The discovered patterns are the sequences of most frequently accessed pages at that site. This kind of information can be used to restructure the web-site, or to dynamically insert relevant links in web pages based on user access patterns. There are many other domains where sequence mining has been applied, which include identifying plan failures [12], finding network alarm patterns [4], and so on.

The task of discovering all frequent sequences in large databases is quite challenging. The search space is extremely large. For example, with m attributes there are $O(m^k)$ potentially frequent sequences of length k. With millions of objects in the database the problem of I/O minimization becomes paramount. However, most current algorithms are iterative in nature, requiring as many full database scans as the longest frequent sequence, which is clearly very expensive. Some of the methods, especially those using some form of sampling, can be sensitive to the data-skew, which can adversely effect performance. Furthermore, most approaches use very complicated internal data structures which have poor locality [8], and add additional space and computation overheads. Our goal is to overcome all of these limitations.

In this paper we present SPADE (Sequential **PA**ttern **D**iscovery using **E**quivalence classes), a new algorithm for discovering the set of all frequent sequences. The key features of our approach are as follows: 1) We use a *vertical id-list* database format, where we associate with each sequence a list of objects in which it occurs, along with the time-stamps. We show that all frequent sequences can be enumerated via simple id-list intersections. 2) We use a lattice-theoretic approach to decompose the original search space (lattice) into smaller pieces (sub-lattices) which can be processed independently in main-memory. Our approach usually requires three database scans, or only a single scan with some pre-processed information, thus minimizing the I/O costs. 3) We decouple the problem decomposition from the pattern search. We propose two different search strategies for enumerating the frequent sequences within each sub-lattice: breadth-first and depth-first search.

SPADE not only minimizes I/O costs by reducing database scans, but also minimizes computational costs by using efficient search schemes. The vertical id-list based approach is also insensitive to data-skew. An extensive set of experiments shows that SPADE outperforms previous approaches by a factor of two, and by an order of magnitude if we have some additional off-line information. Furthermore, SPADE scales linearly in the database size, and a number of other database parameters.

The rest of the paper is organized as follows: In Section 2 we describe the sequence discovery problem and look at related work. In Section 3 we develop our lattice-based approach for problem decomposition, and for pattern search. Section 4 describes our new algorithm, and an experimental study is presented in Section 6. Finally, we conclude in Section 7.

2 Sequence Mining

The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. An *itemset* is a non-empty unordered collection of items (without loss of generality, we assume that items of an itemset are sorted in increasing order). A *sequence* is an ordered list of itemsets. An itemset i is denoted as $(i_1 i_2 \cdots i_k)$, where i_j is an item. An itemset with k items is called a *k-itemset*. A sequence α is denoted as $(\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_q)$, where the sequence *element* α_j is an itemset. A sequence with k items ($k = \sum_j |\alpha_j|$)

* This work was performed while the author was at the University of Rochester, and was supported in part by a NSF Research Initiation Award (CCR-9409120), an ARPA contract F19628-94-C-0057, and a NSF research grant CCR-9705594.

CIKM 98 Bethesda MD USA

is called a *k-sequence*. For example, $(B \mapsto AC)$ is a 3-sequence. An item can occur only once in an itemset, but it can occur multiple times in different itemsets of a sequence.

A sequence $\alpha = (\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_n)$ is a *subsequence* of another sequence $\beta = (\beta_1 \mapsto \beta_2 \mapsto \cdots \mapsto \beta_m)$, denoted as $\alpha \preceq \beta$, if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_j \subseteq b_{i_j}$ for all a_j. For example the sequence $(B \mapsto AC)$ is a subsequence of $(AB \mapsto E \mapsto ACD)$, since the sequence elements $B \subseteq AB$, and $AC \subseteq ACD$. On the other hand the sequence $(AB \mapsto E)$ is not a subsequence of (ABE), and vice versa. We say that α is a proper subsequence of β, denoted $\alpha \prec \beta$, if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$. A sequence is *maximal* if it is not a subsequence of any other sequence.

A *transaction* $\mathcal{T}$ has a unique identifier and *contains* a set of items, i.e., $\mathcal{T} \subseteq \mathcal{I}$. A *customer*, $\mathcal{C}$, has a unique identifier and has associated with it a list of transactions $\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$. Without loss of generality, we assume that no customer has more than one transaction with the same time-stamp, so that we can use the transaction-time as the transaction identifier. We also assume that the list of customer transactions is sorted by the transaction-time. Thus the list of transactions of a customer is itself a sequence $\mathcal{T}_1 \mapsto \mathcal{T}_2 \mapsto \cdots \mapsto \mathcal{T}_n$, called the *customer-sequence*. The database, $\mathcal{D}$, consists of a number of such customer-sequences.

A customer-sequence, $\mathcal{C}$, is said to *contain* a sequence α, if $\alpha \preceq \mathcal{C}$, i.e., if α is a subsequence of the customer-sequence $\mathcal{C}$. The *support* or *frequency* of a sequence, denoted $\sigma(\alpha)$, is the the total number of customers that contain this sequence. Given a user-specified threshold called the *minimum support* (denoted *min_sup*), we say that a sequence is *frequent* if occurs more than *min_sup* times. The set of frequent k-sequences is denoted as $\mathcal{F}_k$.

Given a database $\mathcal{D}$ of customer sequences and *min_sup*, the problem of mining sequential patterns is to find all frequent sequences in the database. For example, consider the customer database shown in figure 1 (used as a running example throughout this paper). The database has eight items (A to H), four customers, and ten transactions in all. The figure also shows all the frequent sequences with a minimum support of 50% or 2 customers. This example has a unique maximal frequent sequence $D \mapsto BF \mapsto A$.

DATABASE

Customer-Id	Transaction-Time	Items
1	10	C D
1	15	A B C
1	20	A B F
1	25	A C D F
2	15	A B F
2	20	E
3	10	A B F
4	10	D G H
4	20	B F
4	25	A G H

FREQUENT SEQUENCES

Frequent 1-Sequences

A	4
B	4
D	2
F	4

Frequent 2-Sequences

AB	3
AF	3
B->A	2
BF	4
D->A	2
D->B	2
D->F	2
F->A	2

Frequent 3-Sequences

ABF	3
BF->A	2
D->BF	2
D->B->A	2
D->F->A	2

Frequent 4-Sequences

D->BF->A	2

Figure 1: Original Database

2.1 Related Work

The problem of mining sequential patterns was introduced in [2]. They also presented three algorithms for solving this problem. The *AprioriAll* algorithm was shown to perform equal to or better than the other two approaches. In subsequent work [11], the same authors proposed the GSP algorithm that outperformed *AprioriAll* by up to 20 times. They also introduced maximum gap, minimum gap, and sliding window constraints on the discovered sequences.

The problem of finding *frequent episodes* in a sequence of events was presented in [6]. An episode consists of a set of events and an associated partial order over the events. Our definition of a sequence can be expressed as an episode, however their work is targeted to discover the frequent episodes in a single long event sequence, while we are interested in finding frequent sequences across many different customer-sequences. They further extended their framework in [5] to discover *generalized episodes*, which allows one to express arbitrary unary conditions on individual episode events, or binary conditions on event pairs. The MEDD and MSDD algorithms [7] discover patterns in multiple event sequences. However, they only find sequences of length 2 with a given window size and a time-gap.

Sequence discovery can essentially be thought of as association discovery [1] over a temporal database. 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. Due to this similarity sequence mining algorithms like *AprioriAll*, GSP, etc., utilize some of the ideas initially proposed for the discovery of association rules [1, 10]. Our new algorithm is based on the fast association mining techniques presented by us in [13]. Nevertheless, the sequence search space is much more complex and challenging than the itemset space, and thus warrants specific algorithms.

3 Sequence Enumeration: Lattice-based Approach

We assume that the reader is familiar with basic concepts of lattice theory (see [3] for a good introduction). Let P be a set. A *partial order* on P is a binary relation $\leq$ on P that is 1) reflexive: $X \leq X$, 2) anti-symmetric: $X \leq Y$ and $Y \leq X$ imply $X = Y$, and 3) transitive: $X \leq Y$ and $Y \leq Z$ imply $X \leq Z$, for all $X, Y, Z \in P$. A partially ordered set L is called a *lattice* if the two binary operations 1) *join*, denoted as $X \vee Y$, and 2) *meet*, denoted as $X \wedge Y$, exist of all $X, Y \in L$. L is a *complete lattice* if the join and meet exist for arbitrary subsets of L. Any finite lattice is thus complete. M is a sub-lattice of L if $X, Y \in M$ implies $X \vee Y \in M$ and $X \wedge Y \in M$.

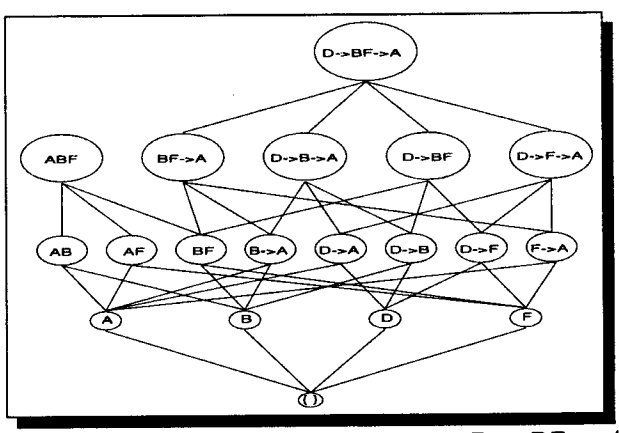

Figure 2: Lattice Induced by Maximal Sequence $D \mapsto BF \mapsto A$

Theorem 1 *Given a set $\mathcal{I}$ of items, the ordered set $\mathcal{S}$ of all possible sequences on the items, is a complete lattice in which join and meet are given by union and intersection, respectively:*

$$\bigvee\{A_i \mid i \in I\} = \bigcup_{i \in I} A_i \qquad \bigwedge\{A_i \mid i \in I\} = \bigcap_{i \in I} A_i$$

The *bottom* element $\perp$ of the sequence lattice $\mathcal{S}$ is $\perp = \{\}$, but the *top* element is undefined, since, in the abstract, the sequence lattice is infinite. However, in all practical cases it is bounded and sparse. The set of *atoms* of lattice L are defined to be the immediate upper neighbors of the bottom element, given as $\mathcal{A}(L) = \{X \in L \mid \perp < X$, and $\perp \leq Y \leq X$ implies $Y = X\}$. For example, consider Figure 2 which shows the sequence lattice induced

by the maximal frequent sequence $D \mapsto BF \mapsto A$ for our example database. The set of atoms $\mathcal{A}$ is given by the frequent items $\{A, B, D, F\}$. It is obvious that the set of all frequent sequences forms a *meet-semilattice*, because it is closed under the meet operation, i.e., if X and Y are frequent sequences, then the meet $X \cap Y$ is also frequent. However, it is not a *join-semilattice*, since it is not closed under joins, i.e., X and Y being frequent, doesn't imply that $X \cup Y$ is frequent. The closure under meet leads to the well known observation on sequence frequency:

Lemma 1 *All subsequences of a frequent sequence are frequent.*

The above lemma leads very naturally to a bottom-up search procedure for enumerating frequent sequences, which has been leveraged in many sequence mining algorithms [11, 6, 7]. However, the lattice formulation makes it apparent that we need not restrict ourselves to a purely bottom-up search. We can employ a number of different search procedures, which we will discuss later.

3.1 Support Counting

Lets associate with each atom X in the sequence lattice its *id-list*, denoted $\mathcal{L}(X)$, which is a list of all customer (*cid*) and transaction identifiers (*tid*) pairs containing the atom. Figure 3 shows the id-lists for the atoms in our example database. For example consider the atom D. In our original database in Figure 1, we see that D occurs in the following customer and transaction identifier pairs $\{(1, 10), (1, 25), (4, 10)\}$. This forms the id-list for item D.

A	
CID	TID
1	15
1	20
1	25
2	15
3	10
4	25

B	
CID	TID
1	15
1	20
2	15
3	10
4	20

D	
CID	TID
1	10
1	25
4	10

F	
CID	TID
1	20
1	25
2	15
3	10
4	20

Figure 3: Id-Lists for the Atoms

Lemma 2 *For any $X \in \mathcal{S}$, let $J = \{Y \in \mathcal{A}(\mathcal{S}) \mid Y \preceq X\}$. Then $X = \bigcup_{Y \in J} Y$, and $\sigma(X) = |\bigcap_{Y \in J} \mathcal{L}(Y)|$.*

The above lemma states that any sequence in $\mathcal{S}$ can be obtained as a union or join of some atoms of the lattice, and the support of the sequence can be obtained by intersecting the id-list of the atoms. This lemma is applied only to the atoms of the lattice. We generalize this for a set of sequences in the next lemma.

Lemma 3 *For any $X \in \mathcal{S}$, if $X = \bigcup Y$, then $\sigma(X) = |\bigcap \mathcal{L}(Y)|$.*

This lemma says that if a sequence is given as a union of a set of sequences in J, then its support is given as the intersection of id-lists of elements in J. In particular we can determine the support of any k-sequence by simply intersecting the id-lists of any two of its $(k-1)$ length subsequences. A simple check on the cardinality of the resulting id-list tells us whether the new sequence is frequent or not. Figure 4 shows this process pictorially. It shows the initial vertical database with the id-list for each atom. The intermediate id-list for $D \mapsto A$ is obtained by intersecting the lists of atoms D and A, i.e., $\mathcal{L}(D \mapsto A) = \mathcal{L}(D) \cap \mathcal{L}(A)$. Similarly, $\mathcal{L}(D \mapsto BF \mapsto A) = \mathcal{L}(D \mapsto BF) \cap \mathcal{L}(D \mapsto B \mapsto A)$, and so on. Thus, only the lexicographically first two subsequences at the last level are required to compute the support of a sequence at a given level.

Lemma 4 *If $X \preceq Y$, then $\mathcal{L}(X) \supseteq \mathcal{L}(Y)$.*

This lemma says that if X is a subsequence of Y, then the cardinality of the id-list of Y (i.e., support) must be equal to or less than the cardinality of the id-list of X. A practical and important consequence of this lemma is that the cardinalities of intermediate id-lists shrink as we move up the lattice. This results in very fast intersection and support counting.

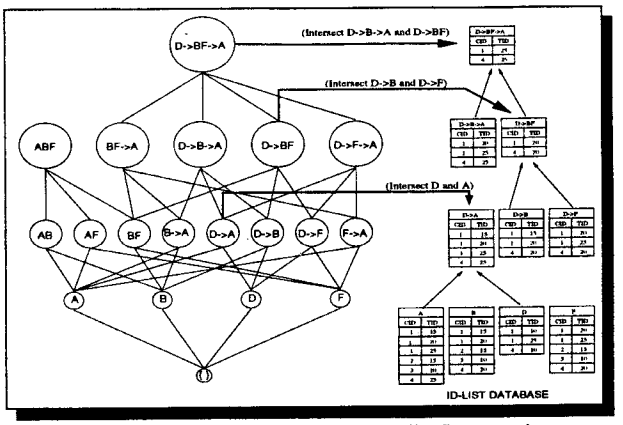

Figure 4: Computing Support via Id-list Intersections

3.2 Lattice Decomposition: Prefix-Based Classes

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 id-lists will not fit in memory. This brings up a natural question: can we decompose the original lattice into smaller pieces such that each piece can be solved independently in main-memory. We address this question below.

An equivalence relation on a set is a reflexive, symmetric and transitive binary relation. An equivalence relation partitions the set into disjoint subsets, called equivalence classes. Define a function $p : \mathcal{S} \mapsto \mathcal{S}$ where $p(X, k) = X[1 : k]$. In other words, $p(X, k)$ returns the k length prefix of X. Define an equivalence relation θ_k on the lattice $\mathcal{S}$ as follows: $\forall X, Y \in \mathcal{S}$, we say that X is related to Y under θ_k, denoted as $X \equiv Y \ (mod\ \theta_k)$ if and only if $p(X, k) = p(Y, k)$. That is, two sequences are in the same class if they share a common k length prefix. We therefore call θ_k a *prefix-based* equivalence relation.

Figure 5 shows the lattice induced by the equivalence relation θ_k where we collapse all sequences with a common k length prefix into an equivalence class. Figure 5a shows the equivalences classes induced by θ_1 on $\mathcal{S}$, namely, $\{[A]_{\theta_1}, [B]_{\theta_1}, [D]_{\theta_1}, [F]_{\theta_1}\}$. At the bottom of the figure, it also shows the links among the four classes. These links carry pruning information. In other words if we want to prune a sequence (if it has at least one infrequent subsequence) then we may need some cross-class information. We will have more to say about this later.

Lemma 5 *Each equivalence class $[X]_{\theta_k}$ induced by the equivalence relation θ_k is a sub-lattice of $\mathcal{S}$.*

Each $[X]_{\theta_1}$ is thus a lattice with its own set of atoms. For example, the atoms of $[D]_{\theta_1}$ are $\{D \mapsto A, D \mapsto B, D \mapsto F\}$, and the bottom element is $\perp = D$. By the application of Lemmas 2, and 3, we can generate all the supports of the sequences in each class (sub-lattice) by intersecting the id-list of atoms or any two subsequences at the previous level. If there is enough main-memory to hold temporary id-lists for each class, then we can solve each $[X]_{\theta_1}$ independently.

In practice we have found that the one level decomposition induced by θ_1 is sufficient. However, in some cases, a class may still be too large to be solved in main-memory. In this scenario, we apply recursive class decomposition. Lets assume that $[D]$ is too large to fit in main-memory. Since $[D]$ is itself a lattice, it can be decomposed using θ_2. Figure 5b shows the classes induced by applying θ_2 on $[D]$ (after applying θ_1 on $\mathcal{S}$). Each of the resulting six classes, $[A]$, $[B]$, $[D \mapsto A]$, $[D \mapsto B]$, $[D \mapsto F]$, and $[F]$, can be solved independently. Thus depending on the amount of main-memory available, we can recursively partition large classes into smaller ones, until each class is small enough to be solved independently in main-memory.

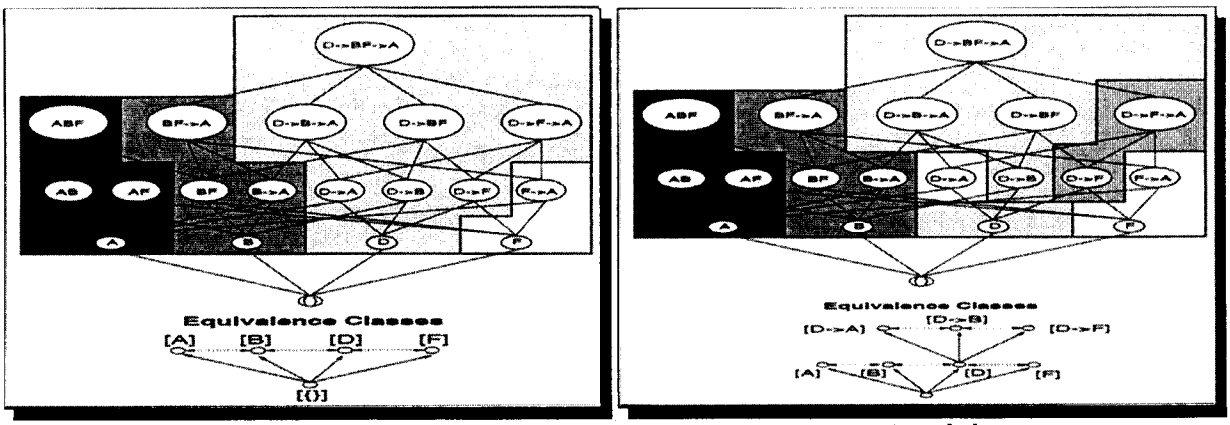

Figure 5: Equivalence Classes Induced by a) θ_1 on $\mathcal{S}$, b) θ_1 on $\mathcal{S}$ and θ_2 on $[D]_{\theta_1}$

3.3 Search for Frequent Sequences

In this section we discuss efficient search strategies for enumerating the frequent sequences within each class. We will discuss two main strategies: breadth-first and depth-first search. Both these methods are based on a recursive decomposition of each class into smaller classes induced by the equivalence relation θ_k. Figure 6 shows the decomposition of $[D]_{\theta_1}$ into smaller and smaller classes, and the resulting lattice of equivalence classes.

Breadth-First Search (BFS) In a breadth-first search the lattice of equivalence classes generated by the recursive application of θ_k is explored in a bottom-up manner. We process all the child classes at each level before moving on to the next level. For example, in Figure 6 we process the equivalence classes $\{[D \mapsto A], [D \mapsto B], [D \mapsto F]\}$, before moving on to the classes $\{[D \mapsto B \mapsto A], [D \mapsto BF], [D \mapsto F \mapsto A]\}$, and so on.

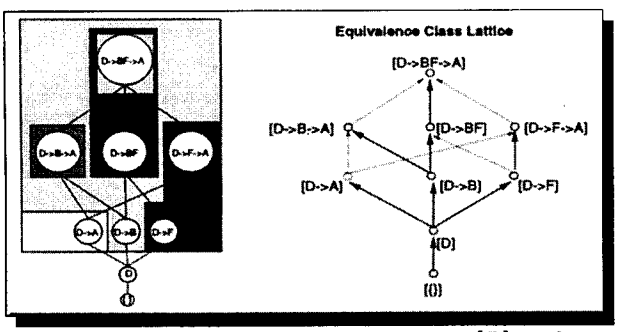

Figure 6: Recursive Decomposition of Class $[D]$ via θ_k

Depth-First Search (DFS) In a depth-first search, we completely solve all child equivalence classes along one path before moving on to the next path. For example, we process the classes in the following order $[D \mapsto A]$, $[D \mapsto B]$, $[D \mapsto B \mapsto A]$, $[D \mapsto BF]$, $[D \mapsto BF \mapsto A]$, and so on.

The advantage of BFS over DFS is that we have more information available for pruning. For example, we know the set of 2-sequences before constructing the 3-sequences, while this information is not available in DFS. On the other hand DFS requires less main-memory than BFS. DFS only needs to keep the intermediate id-lists for classes along a single path, while BFS must keep track of id-lists for all the classes in the current level.

Besides BFS and DFS search, there are many other search possibilities. For example, in the DFS scheme, if we determine that $D \mapsto BF \mapsto A$ is frequent, then we do not have to process the classes $[D \mapsto F]$, and $[D \mapsto F \mapsto A]$, since they must necessarily be frequent. We are currently investigating such schemes for efficient enumeration of only the maximal frequent sequences.

4 SPADE: Algorithm Design and Implementation

In this section we describe the design and implementation of SPADE. Figure 7 shows the high level structure of the algorithm. The main steps include the computation of the frequent 1-sequences and 2-sequences, the decomposition into prefix-based equivalence classes, and the enumeration of all other frequent sequences via BFS or DFS search within each class. We will now describe each step in some more detail.

SPADE $(min\_sup, \mathcal{D})$:
  $\mathcal{F}_1 =$ { frequent items or 1-sequences };
  $\mathcal{F}_2 =$ { frequent 2-sequences };
  $\mathcal{E} =$ { equivalence classes $[X]_{\theta_1}$ };
  for all $[X] \in \mathcal{E}$ **do** *Enumerate-Frequent-Seq*([X]);

Figure 7: The SPADE Algorithm

4.1 Computing Frequent 1-Sequences and 2-Sequences

Most of the current sequence mining algorithms [2, 11] 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. Each customer has a set of transactions, along with the items contained in the transaction. In contrast our algorithms use a *vertical* database format, where we maintain a disk-based id-list for each item. Each entry of the id-list is a (cid, tid) pair where the item occurs. This enables us to check support via simple id-list intersections.

Computing $\mathcal{F}_1$: Given the vertical id-list database, all frequent 1-sequences can be computed in a single database scan. For each database item, we read its id-list from the disk into memory. We then scan the id-list, incrementing the support for each new cid encountered.

Computing $\mathcal{F}_2$: Let $N = |\mathcal{I}|$ be the number of frequent items, and A the average id-list size in bytes. A naive implementation for computing the frequent 2-sequences requires $\binom{N}{2}$ id-list intersections for all pairs of items. The amount of data read is $A \cdot N \cdot (N-1)/2$, which corresponds to around $N/2$ data scans. This is clearly inefficient. Instead of the naive method we propose two alternate solutions: 1) Use a preprocessing step to gather the counts of all 2-sequences above a user specified lower bound. Since this information is invariant, it has to be computed once, and the cost can be amortized over the number of times the data is mined. 2) Perform a vertical-to-horizontal transformation on-the-fly. This can be done quite easily, with very little overhead. For each item i, we scan its id-list into memory. For each customer and transaction id pair, say (c, t) in $\mathcal{L}(i)$, we insert (i, t) in the list for customer c. For example, consider the id-list for item A, shown in Figure 3. We scan the first pair $(1, 15)$, and then insert $(A, 15)$ in the list for customer 1. Figure 8 shows the complete horizontal database recovered from the vertical item id-lists. Computing $\mathcal{F}_2$ from the recovered horizontal database is straight-forward. We form a list of

all 2-sequences in each customer sequence, and update counts in a 2-dimensional array indexed by the frequent items.

cid	(*item*, *tid*) pairs
1	(A 15) (A 20) (A 25) (B 15) (B 20) (C 10) (C 15) (C 25) (D 10) (D 25) (F 20) (F 25)
2	(A 15) (B 15) (E 20) (F 15)
3	(A 10) (B 10) (F 10)
4	(A 25) (B 20) (D 10) (F 20) (G 10) (G 25) (H 10) (H 25)

Figure 8: Vertical-to-Horizontal Database Recovery

Enumerate-Frequent-Seq(S):
  for all atoms $A_i \in S$ **do**
    $T_i = \emptyset$;
    for all atoms $A_j \in S$, 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 $\sigma(R) \geq min\_sup$ **then**
          $T_i = T_i \cup \{R\}$; $\mathcal{F}_{|R|} = \mathcal{F}_{|R|} \cup \{R\}$;
    end
    if (Depth-First-Search) **then** *Enumerate-Frequent-Seq*(T_i);
  end
  if (Breadth-First-Search) **then**
    for all $T_i \neq \emptyset$ **do** *Enumerate-Frequent-Seq*(T_i);

Figure 9: Pseudo-code for Breadth-First and Depth-First Search

4.2 Enumerating Frequent Sequences of a Class

Figure 9 shows the pseudo-code for the breadth-first and depth-first search. The input to the procedure is a set of atoms of a sub-lattice S, along with their id-lists. Frequent sequences are generated by intersecting the id-lists of all distinct pairs of atoms and checking the cardinality of the resulting id-list against *min_sup*. Before intersecting the id-lists a pruning step is inserted to ensure that all subsequences of the resulting sequence are frequent. If this is true, then we go ahead with the id-list intersection, otherwise we can avoid the intersection. The sequences found to be frequent at the current level form the atoms of classes for the next level. This recursive 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 id-lists for at most two consecutive levels. The depth-first search requires memory for two classes on the two levels. The breadth-first search requires memory of all the classes on the two levels. Once all the frequent sequences for the next level have been generated, the sequences at the current level can be deleted.

Disk Scans: Before processing each of equivalence classes from the initial decomposition, all the relevant item id-lists for that class are scanned into from disk into memory. The id-lists for the atoms of each initial class are constructed by intersecting the item id-lists. All the other frequent sequences are enumerated as described above. If all the initial classes have disjoint set of items, then each item's id-list is scanned from disk only once during the entire frequent sequence enumeration process over all sub-lattices. In the general case there will be some degree of overlap of items among the different sub-lattices. However only the database portion corresponding to the frequent items will need to be scanned, which can be a lot smaller than the entire database. Furthermore, sub-lattices sharing many common items can be processed in a batch mode to minimize disk access. Thus we claim that our algorithms will usually require a single database scan after computing $\mathcal{F}_2$, in contrast to the current approaches which require multiple scans.

4.3 Id-List Intersection

We now describe how we perform the id-list intersections for two sequences. Consider an equivalence class $[B \mapsto A]$ with the atom set $\{B \mapsto AB, B \mapsto AD, B \mapsto A \mapsto A, B \mapsto A \mapsto D, B \mapsto A \mapsto F\}$. If we let P stand for the prefix $B \mapsto A$, then we can rewrite the class to get $[P] = \{PB, PD, P \mapsto A, P \mapsto D, P \mapsto F\}$. One can observe the class has two kinds of atoms: the itemset atoms $\{PB, PD\}$, and the sequence atoms $\{P \mapsto A, P \mapsto D, P \mapsto F\}$. We assume without loss of generality that the itemset atoms of a class always precede the sequence atoms. To extend the class it is sufficient to intersect the id-lists of all pairs of atoms. However, depending on the atom pairs being intersected, there can be upto three possible resulting frequent sequences:

1. **Itemset Atom vs Itemset Atom**: If we are intersecting PB with PD, then we get a new itemset atom PDB.
2. **Itemset Atom vs Sequence Atom**: If we are intersecting PB with $P \mapsto A$, then the only possible outcome is new sequence atom $PB \mapsto A$.
3. **Sequence Atom vs Sequence Atom**: If we are intersecting $P \mapsto A$ with $P \mapsto F$, then there are three possible outcomes: a new itemset atom $P \mapsto AF$, and two new sequence atoms $P \mapsto A \mapsto F$ and $P \mapsto F \mapsto A$. A special case arises when we intersect $P \mapsto A$ with itself, which can only produce the new sequence atom $P \mapsto A \mapsto A$.

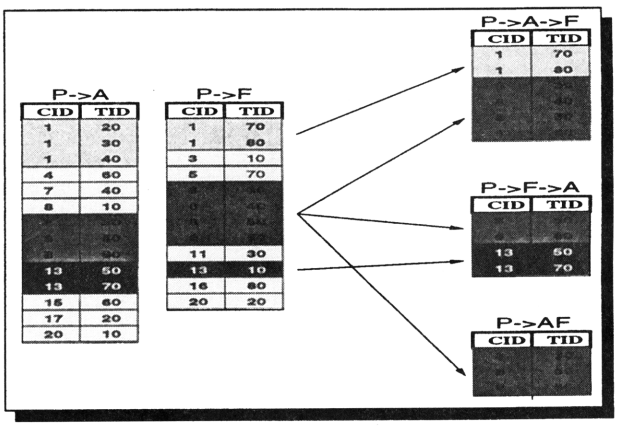

Figure 10: Id-List Intersection

We now describe how the actual id-list intersection is performed. Consider Figure 10, which shows the hypothetical id-lists for the sequence atoms $P \mapsto A$ and $P \mapsto F$. To compute the new id-list for the resulting itemset atom $P \mapsto AF$, we simply need to check for *equality* of (cid,tid) pairs. In our example, the only matching pairs are $\{(8, 30), (8, 50), (8, 80)\}$. This forms the id-list for $P \mapsto AF$. To compute the id-list for the new sequence atom $P \mapsto A \mapsto F$, we need to check for a *follows* relationship, i.e., for a given pair (c, t_1) in $\mathcal{L}(P \mapsto A)$, we check whether there exists a pair (c, t_2) in $\mathcal{L}(P \mapsto F)$ with the same cid c, but with $t_2 > t_1$. If this is true, it means that the item F follows the item A for customer c. In other words, the customer c contains the pattern $P \mapsto A \mapsto F$, and the pair (c, t_2) is added to its id-list. Finally, the id-list for $P \mapsto F \mapsto A$ can be obtained in a similar manner by reversing the roles of $P \mapsto A$ and $P \mapsto F$. The final id-lists for the three new sequences are shown in Figure 10. Since we only intersect sequences within a class, which have the same prefix, we only need to keep track of the last tid for determining the equality and follows relationships. As a further optimization, we generate the id-lists of all the three possible new sequences in just one join.

4.4 Pruning Sequences

The pruning algorithm is shown in Figure 11. Let α_1 denote the first item of sequence α. Before generating the id-list for a new k-sequence β, we check whether all the k subsequences of length $k-1$ are frequent. If they all are frequent then we perform the id-list intersection. Otherwise, β is dropped from consideration.

```
Prune (β):
  for all (k − 1)-subsequences, α ≺ β do
    if ([α1] has been processed, and α ∉ F_{k−1}) then
      return TRUE;
  return FALSE;
```

Figure 11: Sequence Pruning

Note that all subsequences except the last are within the current class. For example consider a sequence $\beta = (D \mapsto BF \mapsto A)$. The first three subsequences, $(D \mapsto BF)$, $(D \mapsto B \mapsto A)$, and $(D \mapsto F \mapsto A)$ are all lie in the class $[D]$. However, the last subsequence $(BF \mapsto A)$ belongs to the class $[B]$. If $[B]$ has already been processed then we have complete subsequence information for pruning. Otherwise, if $[B]$ has not been processed, then we cannot determine whether $(BF \mapsto A)$ is frequent or not. Nevertheless, partial pruning based on the members of the same class is still possible. It is generally better to process the classes in lexicographically descending order, since in this case at least for itemsets all information is available for pruning. This is because items of an itemset are kept sorted in increasing order. For example, if we wanted to test $\beta = ABDF$, then we would first check within its class $[A]$ if ADF is frequent, and since $[B]$ will have been processed if we solve the classes in reverse lexicographic order, we can also check if BDF is frequent.

5 The GSP Algorithm

Below we describe the GSP algorithm [11] in some more detail, since we use it as a base against which we compare SPADE, and it is one of the best current algorithms.

```
F1 = { frequent 1-sequences };
for (k = 2; F_{k−1} ≠ ∅; k = k + 1) do
  C_k = Set of candidate k-sequences;
  for all customer-sequences E in the database do
    Increment count of all α ∈ C_k contained in E
  F_k = {α ∈ C_k | α.sup ≥ min_sup};
Set of all frequent sequences = ⋃_k F_k;
```

Figure 12: The GSP Algorithm

GSP makes multiple passes over the database. In the first pass, all single items (1-sequences) are counted. From the frequent items a set of *candidate* 2-sequences are formed. Another pass is made to gather their support. The frequent 2-sequences are used to generate the candidate 3-sequences, and this process is repeated until no more frequent sequences are found. There are two main steps in GSP, shown in Figure 12 (see [11] for more details). 1) **Candidate Generation**: Given the set of frequent $(k-1)$-sequences, $\mathcal{F}_{k-1}$, the candidates for the next pass are generated by joining $\mathcal{F}_{k-1}$ with itself. A pruning phase eliminates any sequence at least one of whose subsequences is not frequent. For fast counting, the candidate sequences are stored in a *hash-tree*. 2) **Support Counting**: To find all candidates contained in a customer-sequence $\mathcal{E}$, all k-subsequences of $\mathcal{E}$ are generated. For each such subsequence a search is made in the hash-tree. If a candidate in the hash-tree matches the subsequence, its count is incremented.

6 Experimental Results

In this section we compare the performance of SPADE with the GSP algorithm. The GSP algorithm was implemented as described in [11]. For SPADE results are shown only for the BFS search. Experiments were performed on a 100MHz MIPS processor with 256MB main memory running IRIX 6.2, with non-local 2GB disk.
Synthetic Datasets: The synthetic datasets are the same as those used in [11], albeit with twice as many customers. We used the publicly available dataset generation code from the IBM Quest data mining project [9]. These datasets mimic real-world transactions,

Dataset	Size (MB)
C10-T2.5-S4-I1.25-D200K	36.8
C10-T2.5-S4-I1.25-D500K	92.0
C10-T2.5-S4-I1.25-D1000K	184.0
C10-T5-S4-I1.25-D200K	56.5
C10-T5-S4-I2.5-D200K	54.3
C20-T2.5-S4-I1.25-D200K	76.7
C20-T2.5-S4-I2.5-D200K	66.5
C20-T2.5-S8-I1.25-D200K	76.4

Table 1: Synthetic Datasets

where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The customer sequence size and transaction size are clustered around a mean and a few of them may have many elements. 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 average size S are created by assigning itemsets from N_I to each sequence. Next a customer of average C 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 D customers have been generated. Like [11] we set $N_S = 5000$, $N_I = 25000$ and $N = 10000$. The number of data-sequences was set to $D = 200,000$. Table 1 shows the datasets with their parameter settings. We refer the reader to [2] for additional details on the dataset generation.
Plan Dataset: The real-life dataset was obtained from a Natural Language Planning domain. The planner generates plans for routing commodities from one city to another. A "customer" corresponds to a plan identifier, while a "transaction" corresponds to an event in a plan. An event consists of an event identifier, an outcome (such as "success", "late", or "failure"), an action name (such as "move", or "load"), and a set of additional parameters specifying things such as origin, destination, vehicle type ("truck", or "helicopter"), weather conditions, and so on. The data mining goal is to identify the causes of plan failures. There are 77 items, 202071 plans (customers), and 829236 events (transactions). The average plan length is 4.1, and the average event length is 7.6.

6.1 Comparison of SPADE with GSP

Figure 13 compares our SPADE algorithm with GSP, on different synthetic datasets. Each graph shows the results as the minimum support is changed from 1% to 0.25%. Two sets of experiments are reported for each value of support. The bar labeled SPADE corresponds to the case where we computed $\mathcal{F}_2$ via the vertical-to-horizontal transformation method described in Section 4.1. The times for GSP and SPADE include the cost of computing $\mathcal{F}_2$. The bars labeled SPADE-F2 and GSP-F2 correspond to the case where $\mathcal{F}_2$ was computed in a pre-processing step, and the times shown don't include the pre-processing cost.

The figures clearly indicate that the performance gap increases with decreasing minimum support. SPADE is about twice as fast as GSP at lower values of support. In addition we see that SPADE-F2 outperforms GSP-F2 by an order of magnitude in most cases. There are several reasons why SPADE outperforms GSP: 1) SPADE uses only simple join operation on tid-lists. As the length of the frequent sequences increases, the size of the tid-lists decreases, resulting in very fast joins. 2) No complicated hash-tree structure is used, and no overhead of generating and searching of customer subsequences is incurred. These structures typically have very poor locality [8]. On the other hand SPADE has excellent locality, since a join requires only a linear scan of two lists. 3) As the minimum support is lowered, more and larger frequent sequences are found. GSP makes a complete dataset scan for each iteration. SPADE on the other hand restricts itself to usually only three scans. It thus cuts down the I/O costs.

Another conclusion that can be drawn from the SPADE-F2 and GSP-F2 comparison is that nearly all the benefit of SPADE comes

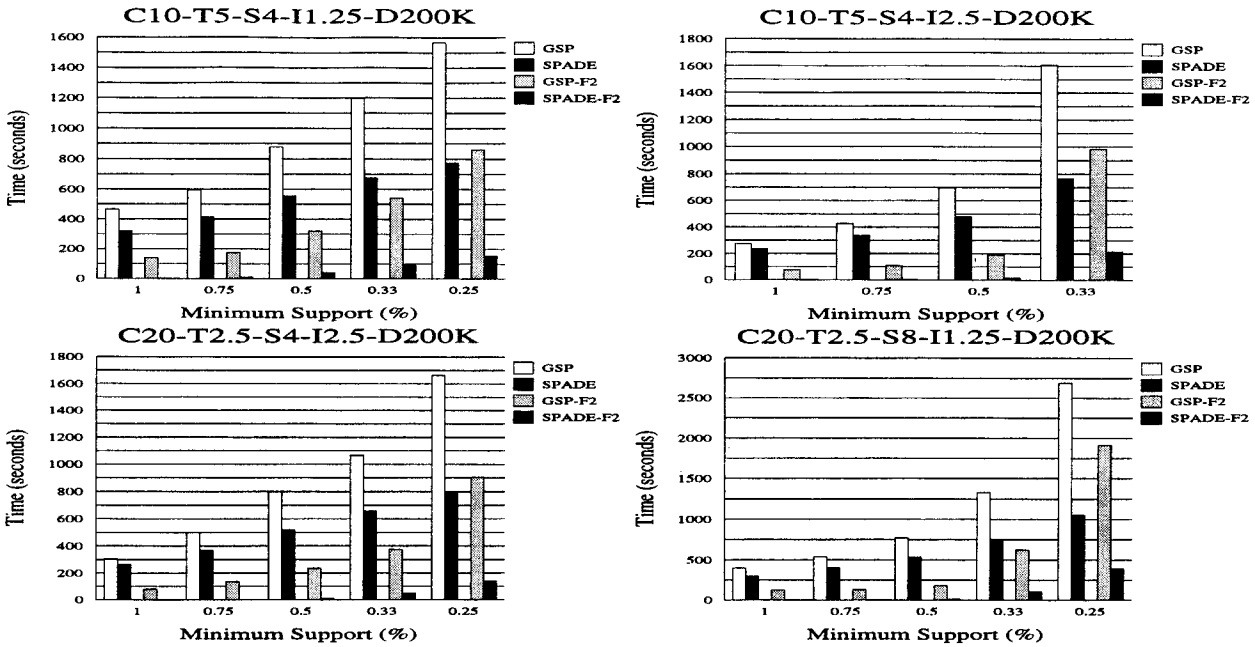

Figure 13: Performance Comparison: Synthetic Datasets

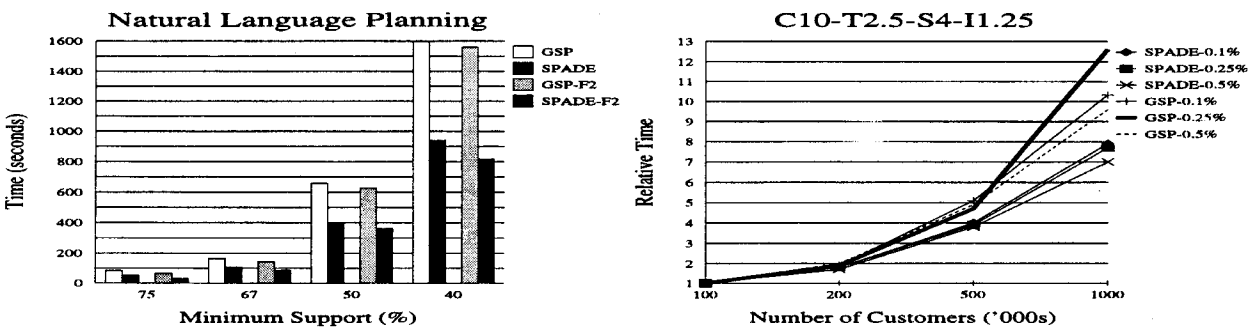

Figure 14: a) Performance Comparison: Planning Dataset; b) Scale-up: Number of Customers

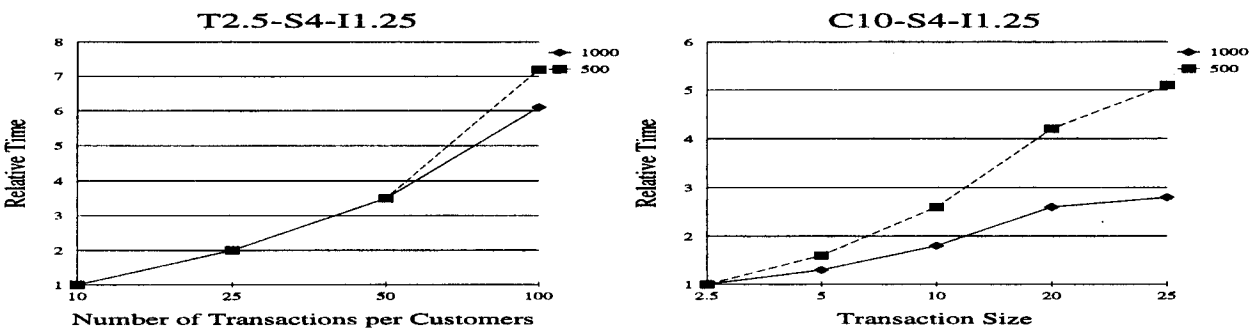

Figure 15: Scale-up: a) # of Transactions/Customer; b) Transaction Size

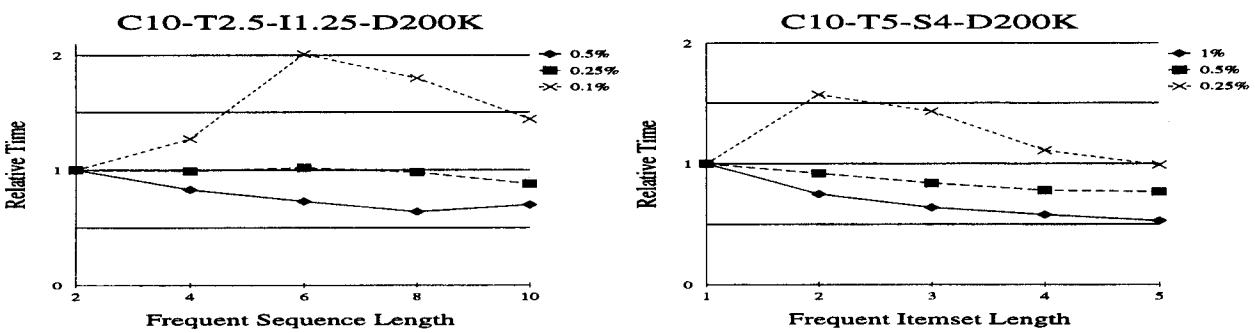

Figure 16: Scale-up: a) Frequent Sequence Length; b) Frequent Itemset Length

from the improvement in the running time after the $\mathcal{F}_2$ pass since both algorithms spend roughly the same time in computing $\mathcal{F}_2$. Between $\mathcal{F}_3$ and $\mathcal{F}_k$, SPADE outperforms GSP anywhere from a factor of three to an order of magnitude.

We also compared the performance of the two algorithms on the plan database. The results are shown in Figure 14 a). As in the case of synthetic databases, the SPADE algorithm outperforms GSP by a factor of two.

6.2 Scaleup

Figure 14 b) shows how SPADE scales up as the number of customers is increased ten-fold, from 0.1 million to 1 million (the number of transactions is increased from 1 million to 10 million, respectively). All the experiments were performed on the *C10-T2.5-S4-I1.25* dataset with different minimum support levels ranging from 0.5% to 0.1%. The execution times are normalized with respect to the time for the 0.1 million customer dataset. It can be observed that SPADE scales quite linearly.

We next study the scale-up as we increase the dataset parameters in two ways: 1) keeping the average number of items per transaction constant, we increase the average number of transactions per customer; and 2) keeping the average number of transactions per customer constant, we increase the average number of items per transaction. The size of the datasets is kept nearly constant by ensuring that the product of the average transaction size, the average number of transactions per customer, and the number of customers ($T \cdot C \cdot D$) remains the same. The aim of these experiments is to gauge the scalability with respect to the two test parameters, and independent of factors like data size or number of frequent sequences.

Figure 15 shows the scalability results. To ensure that the number of frequent sequences doesn't increase by a great amount, we used an absolute minimum support value instead of using percentages (the graph legends indicate the value used). For both the graphs, we used *S4-I1.25*, and the database size was kept a constant at $T \cdot C \cdot D = 500K$. For the first graph we used $T = 2.5$, and varied C from 10 to 100 (D varied from 200K to 20K), and for the second graph we set $C = 10$, and varied T from 2.5 to 25 (D varied from 200K to 20K). It can be easily observed the the algorithm scales linearly with the two varying parameters. The scalability is also dependent on the minimum support value used, since for a lower minimum support relatively more frequent sequences are generated with increase in both the number of transactions, and the transaction size, and thus it takes more time for pattern discovery.

We further study the scalability as we change the size of the maximal elements in two ways: i) keeping all other parameters constant, we increase the average length of maximal potential frequent sequences; and ii) keeping all other parameters constant, we increase the average length of maximal potential frequent itemsets. The constant parameters for the first experiment were *C10-T2.5-I1.25-D200K*, and S was varied from 2 to 10. For the second experiment, the constant parameters were *C10-T5-S4-D200K*, and I was varied from 1 to 5.

Figure 16 shows how the algorithm scales with the two test parameters. For higher values of support the time starts to decrease with increasing maximal element size. This is because of the fact that the average transaction size and average number of customer transactions remains fixed, and increasing the maximal frequent sequence or itemset size means that fewer of these will fit in a customer-sequence, and thus fewer frequent sequences will be discovered. For lower values of support, however, a larger sequence will introduce many more subsequences, thus the time starts to increase initially, but then decreases again due to the same reasons given above. The peak occurs at roughly the median values of $C10$ (at $S6$) for the sequences experiment, and of $T5$ (at $I2$) for the itemsets experiment.

7 Conclusions

In this paper we presented SPADE, a new algorithm for fast mining of sequential patterns in large databases. Unlike previous approaches which make multiple database scans and use complex hash-tree structures that tend to have sub-optimal locality, SPADE decomposes the original problem into smaller sub-problems using equivalence classes on frequent sequences. Not only can each equivalence class be solved independently, but it is also very likely that it can be processed in main-memory. Thus SPADE usually makes only three database scans – one for frequent 1-sequences, another for frequent 2-sequences, and one more for generating all frequent k-sequences ($k \geq 3$). SPADE uses only simple intersection operations, and is thus ideally suited for direct integration with a DBMS. An extensive set of experiments has been conducted to show that SPADE outperforms the best previous algorithm, GSP, by a factor of two, and by an order of magnitude with precomputed support of 2-sequences. It also has excellent scaleup properties with respect to a number of parameters such as the number of customers, the number of transactions per customer, transaction size, and size of potential maximal frequent itemsets and sequences.

This work opens several research opportunities, which we plan to address in the future: 1) Implementation of SPADE directly on top of a DBMS. 2) Parallel discovery of sequences. 3) Discovery of *quantitative* sequences – where the quantity of items bought is also considered. 4) Enumerating *generalized* sequences using the SPADE approach – introducing minimum and maximum time gap constraints, incorporating sliding windows, and imposing a taxonomy on the items.

References

[1] R. Agrawal et al. Fast discovery of association rules. In U. Fayyad, et al (eds.) *Advances in KDD*, AAAI Press, 1996.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th ICDE Conf.*, 1995.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[4] K. Hatonen, et al. Knowledge discovery from telecom. network alarm databases. In *12th ICDE Conf.*, Feb 1996.

[5] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurences. In *2nd Intl. Conf. Knowledge Discovery and Data Mining*, 1996.

[6] H. Mannila, H. Toivonen, and I. Verkamo. Discovering frequent episodes in sequences. In *1st Intl. Conf. KDD*, 1995.

[7] T. Oates, et al. Algorithms for finding temporal structure in data. In *6th Intl. Wkshp. AI and Statistics*, Mar 1997.

[8] S. Parthasarathy, M. J. Zaki, and W. Li. Memory placement techniques for parallel association mining. In *4th Intl. Conf. KDD*, Aug 1998.

[9] http://www.almaden.ibm.com/cs/quest/syndata.html. Quest Project. IBM Almaden Research Center, San Jose, CA 95120.

[10] A. Savasere, et al. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

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

[12] M. J. Zaki, et al. PLANMINE: Sequence mining for plan failures. In *4th Intl. Conf. KDD*, Aug 1998.

[13] M. J. Zaki, et al. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. KDD*, Aug 1997.