Finding Frequent Patterns Using Length-Decreasing Support Constraints *

Masakazu Seno and George Karypis

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455
Technical Report 03–0004

{seno, karypis}@cs.umn.edu

Abstract

Finding prevalent patterns in large amount of data has been one of the major problems in the area of data mining. Particularly, the problem of finding frequent itemset or sequential patterns in very large databases has been studied extensively over the years, and a variety of algorithms have been developed for each problem. The key feature in most of these algorithms is that they use a constant support constraint to control the inherently exponential complexity of these two problems. In general, patterns that contain only a few items will tend to be interesting if they have a high support, whereas long patterns can still be interesting even if their support is relatively small. Ideally, we want to find all the frequent patterns whose support decreases as a function of their length without having to find many uninteresting infrequent short patterns. Developing such algorithms is particularly challenging because the downward closure property of the constant support constraint cannot be used to prune short infrequent patterns.

In this paper we present two algorithms, LPMiner and SLPMiner. Given a length-decreasing support constraint, LPMiner finds all the frequent itemset patterns from an itemset database, and SLPMiner finds all the frequent sequential patterns from a sequential database. Each of these two algorithms combines a well-studied efficient algorithm for constant-support-based pattern discovery with three effective database pruning methods that dramatically reduce the runtime. Our experimental evaluations show that both LPMiner and SLPMiner, by effectively exploiting the length-decreasing support constraint, are up to two orders of magnitude faster, and their runtime increases gradually as the average length of the input patterns increases.

Keywords: frequent pattern discovery, data-mining, association rules, scalability

*This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, ACI-0133464, and by Army High Performance Computing Research Center contract number DA/DAAG55-98-1-0441. Access to computing facilities was provided by the Minnesota Supercomputing Institute.

1 Introduction

Finding prevalent patterns in a large amount of data is one of the important problems in the area of data mining. The problem of finding itemset patterns from large itemset databases has been studied extensively during the last nine years and has led to the development of a variety of algorithms [3, 1, 8, 27]. The itemset model regards a set of objects (or items) as a pattern. By generalizing the itemset model, the sequence model was introduced, and various algorithms have been developed [21, 25, 18]. The sequence model introduces ordering among a set of itemsets and regards those ordered itemsets as a pattern. Each of these two types of patterns can be effectively used to find association rules or extract prevalent patterns from itemset or sequential databases in various domains and applications.

The key feature in most of these algorithms is that they control the inherently exponential complexity of the problem by finding only the patterns that occur in a sufficiently large fraction of the input itemsets or sequences, called the *support*. A limitation of this paradigm for generating frequent patterns is that it uses a constant support value, irrespective of the length of the discovered patterns. In general, patterns that contain only a few items will tend to be interesting if they have a high support, whereas long patterns can still be interesting even if their support is relatively small. Unfortunately, if constant-support-based frequent pattern discovery algorithms are used to find some of the longer but infrequent patterns, they will end up generating an exponentially large number of short patterns. Ideally, we would like to find frequent patterns whose support decreases as a function of their length without having to find many uninteresting infrequent short patterns.

In this paper, we introduce the problem of finding frequent patterns whose support satisfies a non-increasing function of their length, for both itemset and sequential patterns. A pattern is frequent if and only if its support is greater than or equal to the minimum support value determined by the length of the pattern. We refer to such a support constraint as the ***length-decreasing support constraint***. Unfortunately, the downward closure property does not hold for the problem of finding the complete set of frequent patterns that satisfy a length-decreasing support constraint as a pattern can be frequent even if its sub-patterns are infrequent. To overcome this problem, we identified a new property that an infrequent pattern must have in order to grow to longer frequent patterns under a length-decreasing support constraint. This property, called the ***smallest valid extension*** or ***SVE*** for short allows us to prune large portions of the input database that are irrelevant for finding frequent patterns.

We evaluate the effectiveness of the SVE property, by developing two algorithms, LPMiner and SLPMiner. Given a length-decreasing support constraint, LPMiner finds all the frequent itemsets from an itemset database, and SLPMiner finds all the frequent sequential patterns from a sequential database. We conducted a series of experiments on these algorithms to evaluate the effectiveness of pruning methods that we incorporated into these algorithms to exploit the length-decreasing support constraint. Both of LPMiner and SLPMiner achieve up to two orders of magnitude of speedup by effectively exploiting the SVE property, and their runtime increases gradually as the average length of the input itemsets or sequences (and the discovered patterns) increases.

The rest of this paper is organized as follows. Section 2 provides some background definitions and introduces the notation that we will be using through-out the paper. Section 3 provides an overview of related research in this area. Section 4 introduces the notion of the length-decreasing support constraint and discusses the smallest valid extension property that can be used to potentially prune the search space. Section 5 focuses on the problem of finding frequent itemsets that satisfy a length-decreasing support constraint and describes the LPMiner algorithm that we developed for solving this problem. Similarly, Section 6 focuses on the corresponding problem for sequential patterns and describes the SLPMiner algorithm that finds frequent sequential patterns. The experimental evaluations of LPMiner and SLPMiner are shown in Section 7. Finally, Section 8 provides some concluding remarks and future research directions.

2 Definitions and Notation

We will use the itemset model [3] and sequence model [21], both of which were introduced by Agrawal and Srikant. These two models are defined as follows. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. An ***itemset*** is a subset of I. A ***sequence*** $s = \langle t_1, t_2, \ldots, t_l \rangle$ is an ordered list of itemsets, where $t_j \subseteq I$ for $1 \leq j \leq l$. A sequential database D is a set of sequences. The length of a sequence s is defined to be the number of items in s and denoted by $|s|$. Similarly, the length of an itemset t is defined to be the number of items in t and denoted by $|t|$; thus $|s| = |t|$ if s consists of only one itemset t. Given a sequential database D, $|D|$ denotes the number of sequences in D. We assume that the items in I can be arranged in a lexicographic order, and we will use consecutive integers starting from one to represent the items according to that ordering.

Sequence $s = \langle t_1, t_2, \ldots, t_l \rangle$ is called a ***sub-sequence*** of sequence $s' = \langle t'_1, t'_2, \ldots, t'_m \rangle$ ($l \leq m$) if there exist l integers $i_1, i_2, \ldots i_l$ such that $1 \leq i_1 < i_2 < \ldots < i_l \leq m$ and $t_j \subseteq t'_{i_j}$ ($j = 1, 2, \ldots, l$). If s is a sub-sequence of s', then we write $s \subseteq s'$ and say sequence s' ***supports*** s. In our algorithms, we often consider a sequential database D' that is derived from the input sequential database D by eliminating certain sequences or certain items in D. The ***support*** of a sequence s in a sequential database D' derived from the input sequential database D is defined to be $|D'_s|/|D|$ and denoted by $\sigma_{D'}(s)$, where $D'_s = \{s_i | s \subseteq s_i \wedge s_i \in D'\}$. If a sequence s has only one itemset t, we use $\sigma_{D'}(t) = \sigma_{D'}(\langle t \rangle)$. Furthermore, if the itemset t has only one item i, we use $\sigma_{D'}(i) = \sigma_{D'}(\langle t \rangle)$. From the definition, it always holds that $0 \leq \sigma_{D'}(s) \leq 1$ for any sequence $s \in D'$ and any sequential database D' derived from the input sequential database D (including the case in which $D' = D$). We use the term ***sequential pattern*** to refer to a sequence when we want to emphasize that the sequence is supported by many sequences in a sequential database.

We will use the traditional method for writing out sequences, in which each itemset is represented as a list of items ordered according to their lexicographical order and enclosed within matched parentheses (), and the sequence of these itemsets is written one-after-the-other enclosed within matched angled parentheses $\langle\rangle$.

To illustrate the above definitions consider the set of items $I = \{1, 2, 3\}$. This set can generate seven possible itemsets and each of them is represented as (1),(2),(3),(1, 2), (1, 3),(2, 3),(1, 2, 3). Let $t_1 = (1, 2)$, $t_2 = (1, 2, 3)$, and $t_3 = (3)$, be three itemsets of sizes two, three, and one, respectively. Sequence $s = \langle t_1, t_2, t_3 \rangle = \langle (1, 2), (1, 2, 3), (3) \rangle$ has three itemsets and has length $|s| = 2 + 3 + 1 = 6$. Sequence $s' = \langle (1, 3), (1, 2, 3), (1, 2, 3), (2), (2, 3) \rangle$ supports s, or in other words s is a sub-sequence of s'.

If every sequence in a sequential database D has exactly one itemset, we refer to such a sequential database as an ***itemset database*** and regard each sequence $s = \langle t \rangle$, where t is an itemset, as just an itemset t. If an itemset is supported by many itemsets in an itemset database, we call it ***itemset pattern***. Since itemset databases are special case of sequential databases, any frequent sequential pattern mining algorithm is applicable to itemset databases for frequent itemset pattern mining. There exist, however, many algorithms that are designed specific to itemset pattern mining and in general much more efficient than sequential pattern mining algorithms. When we discuss both sequential patterns and itemset patterns, we use the term ***pattern*** to refer to both of them.

The problem of finding frequent sequential patterns given a constant minimum support constraint [21] is formally defined as follows:

> **Definition 1 (Sequence Mining with Constant Support)** *Given a sequential database D and a minimum support σ ($0 \leq \sigma \leq 1$), find all sequences each of which is supported by at least $\lceil \sigma |D| \rceil$ sequences in D.*

Such sub-sequences are called ***frequent sequential patterns***. If every sequence consists of exactly one itemset, the problem of finding frequent sequential patterns degenerates to the problem of finding ***frequent itemset patterns*** in an itemset database [3] that is defined as follows:

> **Definition 2 (Itemset Mining with Constant Support)** *Given an itemset database D and a minimum support σ ($0 \leq \sigma \leq 1$), find all itemsets each of which is supported by at least $\lceil \sigma |D| \rceil$ itemsets in D.*

3 Related Research

Efficient algorithms for finding frequent itemsets or sequences in very large itemset or sequence databases have been one of the key success stories of data mining research. One of the early computationally efficient algorithm was Apriori [3], which finds frequent itemsets of length l based on previously generated $(l-1)$-length frequent itemsets. The GSP [20] algorithm extended the Apriori-like level-wise mining method to find frequent patterns in sequential databases. The basic level-wise algorithm has been extended in a number of different ways leading to more efficient algorithms such as DHP [14, 13], Partition [19], SEAR and Spear [12], and DIC [5]. An entirely different approach for finding frequent itemsets and sequences are the equivalence class-based algorithms Eclat [26] and SPADE [24] that break the large search space of frequent patterns into small and independent chunks and use a vertical database format that allows them to determine the frequency by computing set intersections. More recently, a set of database-projection-based methods has been developed that significantly reduce the complexity of finding frequent patterns [1, 8, 7, 18]. The key idea behind these methods is to find the patterns by growing them one item at a time, and simultaneously partitioning (*i.e.*, projecting) the original database into pattern-specific sub-databases (which in general overlap). The process of pattern-growth and database-projection is repeated recursively until all frequent patterns are discovered. Prototypical examples of such algorithms are the tree-projection algorithm [1] that constructs a lexicographical tree and projects a large database into a set of reduced, item-based sub-databases based on the frequent patterns mined so far. The original algorithm was developed for finding non-sequential patterns, but it has been extended to find sequential patterns as well [7]. Another similar algorithm is the FP-growth algorithm [8] that combines projection with the use of the FP-tree data structure to compactly store in memory the itemsets of the original database. The basic ideas in this algorithm were recently used to develop a similar algorithm for finding sequential patterns [18].

The problem of finding frequent patterns has been extended to that of finding frequent maximal patterns [4, 10, 2, 27] and finding frequent closed patterns [15, 23, 17]. Both of these problem formulations can be used to reduce the number of patterns that gets discovered and help in finding long patterns present in the data. However, both of these problem formulations can still generate a very large number of short infrequent itemsets if these itemsets are maximal or closed.

Even though to our knowledge no work has been published for finding frequent itemsets in which the support decreases as a function of the length of the itemset, there has been some work in developing itemset discovery algorithms that use multiple support constraints. Liu *et al* [11] presented an algorithm in which each item has its own minimum item support (or MIS). The minimum support of an itemset is the lowest MIS among those items in the itemset. By sorting items in ascending order of their MIS values, the minimum support of the itemset never decreases as the length of itemset grows, making the support of itemsets downward closed. Thus an Apriori-based algorithm can be applied. Wang *et al* [22] allow a set of more general support constraints. In particular, they associate a support constraint for each one of the itemsets. By introducing a new function called Pminsup that has an "Apriori-like" property, they proposed an Apriori-based algorithm for finding the frequent itemsets. It is possible to represent a length-decreasing support constraint by using the formulation in [22]. However, the "*pushed*" minimum support of each itemset is forced to be equal to the support value corresponding to the longest itemset. Thus, it cannot prune the search space. Cohen *et al* [6] adopt a different approach in that they do not use any support constraint. Instead, they search for similar itemsets using probabilistic algorithms, that do not guarantee that all frequent itemsets can be found.

In [9] a FP-tree based algorithm called TFP was introduced for finding top-k frequent closed patterns of length no less than a given value. TFP does not input a predefined minimum support. Instead, starting from 0, the minimum support is raised based on candidates top-k frequent closed patterns found so far. Similar to LPMiner, TFP is effective for finding long frequent itemsets. Furthermore, TFP allows users to input the number of patterns to be discovered rather than less intuitive minimum support. One drawback with TFP is that any patterns shorter than a given minimum length are never discovered, which might not be appropriate for some applications.

Finally, a good discussion of the different type of constraints that have been used in the context of pattern discovery

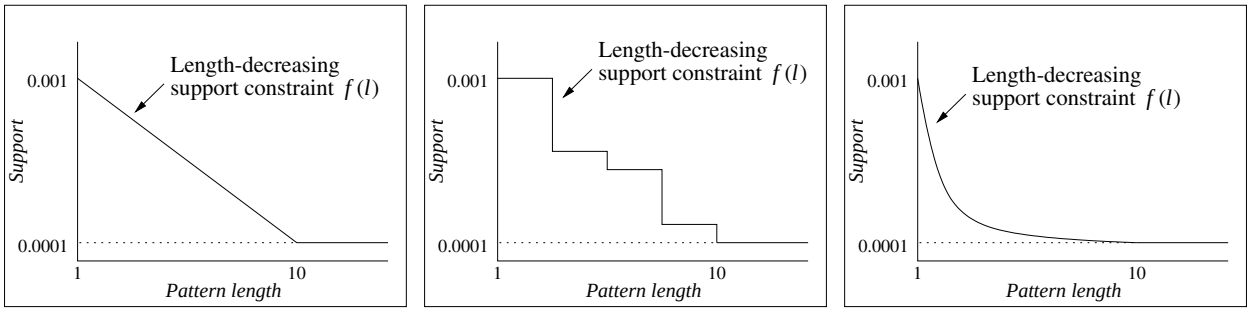

Figure 1: Typical length-decreasing support constraint functions.

is presented in [16]. In particular, the authors classify the various constraints into four categories, namely, anti-monotone, monotone, succinct, and convertible. For these constraints it has been shown that efficient mining can be achieved by pushing the constraints into mining process. However, the length-decreasing support constraint does not fall into any of these four categories.

4 Pattern Mining with Length-Decreasing Support Constraint

A limitation of the above paradigm for generating frequent patterns is that it uses a constant value of support, irrespective of the length of the discovered patterns. In general, patterns that contain only a few items will tend to be interesting if they have a high support, whereas long patterns can still be interesting even if their support is relatively small. Unfortunately, if constant-support-based frequent pattern discovery algorithms are used to find some of the longer but infrequent patterns, they will end up generating an exponentially large number of short patterns.

In order to solve this problem, we introduce the idea of length-decreasing support constraint that helps us to find long patterns with low support as well as short patterns with high support. A length-decreasing support constraint is given as a non-increasing function $f(l)$ of the pattern length l. The idea of introducing this kind of support constraint is that by using a support function that decreases as the length of the pattern increases, we may be able to find long patterns that may be of interest without generating an exponentially large number of shorter patterns. Formally, the length-decreasing support constraint is defined as follows:

Definition 3 (Length-Decreasing Support Constraint) *Given an itemset or sequential database D and a function $f(l)$ that satisfies $1 \geq f(l) \geq f(l+1) \geq 0$ for any positive integer l, a pattern s is frequent if and only if $\sigma_D(s) \geq f(|s|)$.*

Figure 1 shows some typical length-decreasing support constraints functions. For example, Figure 1(a) shows a support constraint function that decreases linearly from 0.001 to 0.0001 as the length of the pattern increases from one to ten, and then remains constant at 0.0001 for patterns that are longer than ten items. Similarly, Figures 1(b) and 1(c) show a stair-case-type function and an exponential decreasing function, respectively. Given a length-decreasing support constraint $f(l)$, we can define the inverse function of $f(l)$ as follows:

Definition 4 (The Inverse Function of Length-Decreasing Support Constraint) *Given a length-decreasing support constraint $f(l)$, its inverse function is defined as $f^{-1}(\sigma) = \min(\{l | f(l) \leq \sigma\})$ for $0 \leq \sigma \leq 1$.*

Essentially, for a given support value σ, $f^{-1}(\sigma)$ is the minimum length l that a pattern must have in order to satisfy the support constraint (*i.e.*, be considered frequent).

Formally, the problems of finding frequent itemsets and sequences that satisfy a length-decreasing support constraint are defined as follows:

Definition 5 (Itemset Mining with Length-Decreasing Support) *Given an itemset database D and a length-decreasing support constraint $f(l)$, find all itemsets t such that $\sigma_D(t) \geq f(|t|)$.*

Definition 6 (Sequence Mining with Length-Decreasing Support) *Given a sequential database D and a length-decreasing support constraint $f(l)$, find all sequential patterns s such that $\sigma_D(s) \geq f(|s|)$.*

4.1 Smallest Valid Extension Property

The key feature of existing algorithms for finding frequently occurring patterns that satisfy a constant minimum support constraint is that they control the inherently exponential complexity of the problem by using the downward closure property [3, 21]. This property states that in order for a pattern of length l to be frequent, all of its sub-patterns (*i.e.*, sub-itemsets or sub-sequences) must be frequent as well. As a result, once these algorithms determine that a pattern s of length l is infrequent, then any longer pattern s' that includes s cannot be frequent, and thus eliminate such patterns from further consideration.

Unfortunately, the downward closure property does not hold for the problem of finding the complete set of frequent patterns that satisfy a length-decreasing support constraint. This is because under this types of constraints, a pattern can be frequent even if its sub-patterns are infrequent since the minimum support value decreases as the length of a pattern increases. The only way that existing, constant-support, frequent pattern discovery algorithms can be used to find the desired set of patterns is to set $\sigma = \min_{l \geq 1} f(l)$ (*i.e.*, the smallest possible support value allowed by the length-decreasing support constraint) and discard any discovered patterns that do not satisfy the length-decreasing support constraint. However, this approach does not reduce the number of infrequent patterns that is being discovered, and as our experiments will show, requires a large amount of time. Thus, the fact that the downward closure property does not hold, makes the task of developing computationally efficient algorithms for discovering such patterns particularly challenging.

A key property regarding patterns whose support decreases as a function of their length is encapsulated in the following lemma.

Lemma 1 (Smallest Valid Extension) *Given an itemset or sequential database D and an itemset or sequence pattern $s \in D$, if s is currently infrequent ($\sigma_D(s) < f(|s|)$), then $f^{-1}(\sigma_D(s))$ is the minimum length that a pattern $s' \supset s$ must have before it can potentially become frequent.*

The best way to understand (and prove) this lemma is to look at Figure 2 that graphically illustrates the relation between s and s' and their respective lengths. Given a pattern s, the length of s' (*i.e.*, $f^{-1}(\sigma_D(s))$) corresponds to the x-coordinate of the point at which a line parallel to the x-axis at $y = \sigma_D(s)$ intersects the length-decreasing support curve. This relation between the lengths of s and s' essentially assumes that pattern s can be extended to form pattern s' in a way such that every database itemset or sequence that supported s (*i.e.*, $\sigma_D(s)$) will also support s'. Note that $\sigma_D(s)$ represents an upper-bound on the support of s' because every database itemset or sequence that supports s' will also support s, but there may be some database itemsets or sequences that support s but not s'. We will refer to this relation between the current length of an infrequent pattern s and the minimum length that a superset pattern s' must have before it can potentially become frequent as the ***smallest valid extension*** property or ***SVE*** property for short.

The SVE property is important as it allow us to prune both the space of candidate patterns as well as the database that can potentially support any supersets of a particular pattern. For example, if s is infrequent, then from the SVE property we know that any superset pattern s' whose length is smaller than $f^{-1}(\sigma_D(s))$ will also be infrequent. Thus, we can dramatically reduce the set of patterns that we need to consider. Similarly, if s is infrequent, then because $f^{-1}(\sigma_D(s))$ is the minimum length of a potential frequent pattern derived from s, any database itemset or sequence that supported s but whose length is smaller than $f^{-1}(\sigma_D(s))$ does not need to be considered as it can never contribute to a frequent pattern that is a superset of s. Thus, we can dramatically reduce the size of the database in which we search for frequent patterns.

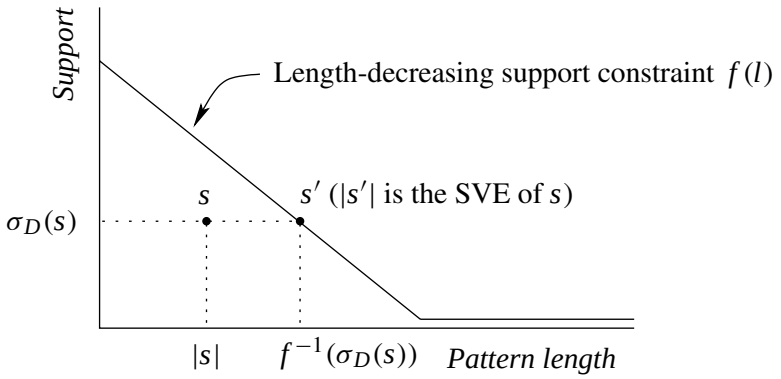

Figure 2: Smallest valid extension property.

5 Itemset Mining with Length-Decreasing Support Constraint

We developed an algorithm called LPMiner that finds all frequent itemsets that satisfy a given length-decreasing support constraint function. The underlying structure of LPMiner's frequent pattern discovery algorithm follows the general framework of the recently introduced FP-growth algorithm [8]. However, LPMiner extends FP-growth by incorporating three different and complementary pruning methods that rely on the SVE property to substantially prune the database and the space of patterns that are being considered. In the rest of this section we briefly review the key elements of the FP-growth algorithm and describe the pruning methods that we developed.

5.1 FP-growth Algorithm

The key idea behind the FP-growth algorithm [8] is to use a data structure called FP-tree to compactly store the database so that it can fit in the main memory. As a result, any subsequent operations that are required to find the frequent itemset patterns can be performed quickly, without accessing the disks.

The FP-tree itself can be built efficiently by requiring only two passes over the input database. Figure 3 shows how the FP-tree generation algorithm works given an input database D that has five itemsets $(t_1, t_2, \ldots, t_5)$ with items $I = \{1, 2, 3, 4, 5, 6\}$, and a constant minimum support constraint of $\sigma = 0.4$. First, it scans D to find the number of times that each item occurs in the various itemsets and uses this information to build the *Item Support Table*. This table consists of a set of (item-ID, support) pairs. For example, item 1 occurs twice in the itemset database (in itemsets t_1 and t_5); therefore its support is $2/5 = 0.4$. Then, any items whose support is smaller than the minimum support are eliminated (*e.g.*, item 6 in our example), and the remaining items are sorted in non-increasing order according to their support. The resulting ordering is stored in an array called the "Node-Link" header table or NL for short. Finally, the FP-tree is generated by reading the itemsets from the database and inserting them one-by-one in the tree. Initially the FP-tree has only a root node called the *null node*. Each (non-root) node of the FP-tree contains three fields. The first field corresponds to the item-ID of the item for which this node was created, the second field represents a counter that is set to one the moment a node is created, and the third field is used to form a link-list of all the nodes corresponding to the same item. Note that the FP-tree of Figure 3 uses a two-element array to represent each node in which the first element corresponds to the item-ID and the second element corresponds to the counter.

The actual process of building the FP-tree is better explained by going through the steps involved in inserting some of the transactions of Figure 3. When the first itemset $t_1 = (1, 2, 3, 4, 5)$ is read from the disk, its items are first re-ordered to match the ordering in the NL-table (*i.e.*, $t'_1 = (2, 3, 5, 1, 4)$) and are then inserted into the tree according to that order. That is, the algorithm creates nodes that represent items 2, 3, 5, 1, and 4 along a path from the root to the node corresponding to item 4. Next, when the second itemset ($t_2 = (2, 5)$) is read, its items re-ordered according to the NL-table ($t'_2 = (2, 5)$) and is inserted in the FP-tree, a node representing item 2 is *not* generated; instead, the

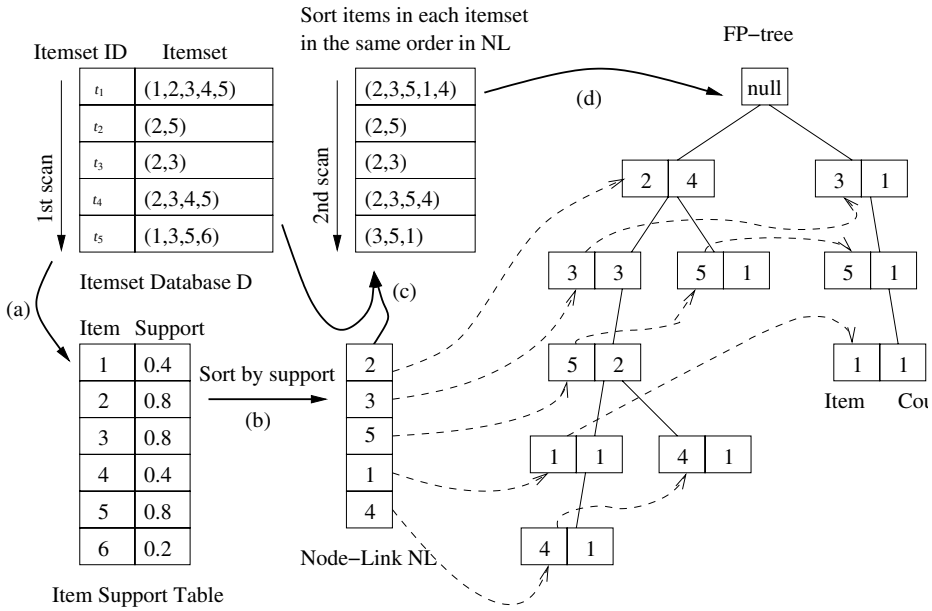

Figure 3: Flow of FP-tree generation.

node already generated is reused. In this case, because the root node has a child that represents item 2, the count of the node is incremented by one. As for item 5, since there is no child node representing item 5 under the current node, a new node with item-name 5 is generated as a child of the current node. Similar processes are repeated until all the sorted itemsets are inserted into the FP-tree. Note that as the itemsets are being inserted in the FP-tree, if they contain any items that have already been eliminated due to the support constraint (and are not in the NL-table), they are also eliminated from the itemsets. Also, in addition to the tree-structure itself, the FP-tree data-structure also maintains for each item a link-list that connects all the nodes of the tree that store the same item. The header pointer of these link-lists is the NL array as illustrated in Figure 3.

Once an FP-tree is generated from the input database D, FP-growth mines frequent patterns by only accessing this FP-tree. The algorithm generates patterns from shorter to longer ones by adding items one-by-one to those patterns already generated. It divides mining the FP-tree into mining smaller FP-trees, each of which is based on an item of the Node-Link header table in Figure 3. To illustrate this process consider the patterns that are generated from item 4. Initially, the FP-growth algorithm generates a new itemset database called *conditional pattern base*. Each itemset in the conditional pattern base consists of items on the paths from parent nodes whose child nodes have item 4 to the root node. The conditional pattern base for item 4 is shown in Figure 4. Each itemset in the conditional pattern base also has its frequency of occurrence corresponding to the counter of the node with item 4 in the original FP-tree. Note that item 4 itself is a frequent itemset pattern consisting of one item, and in the FP-growth terminology it is called a *conditional pattern*. A conditional pattern base is a set of itemsets each of which includes the conditional pattern. Then, the algorithm proceeds to generate frequent itemset patterns that include the conditional pattern (4) by using only its conditional pattern base. For this purpose, a smaller FP-tree is generated based on the conditional pattern (4). This new FP-tree, called *conditional FP-tree*, is generated from the conditional pattern base using the FP-tree generation algorithm again. If the conditional FP-tree is not a single path tree, the process of mining this conditional FP-tree is recursively decomposed to that of mining even smaller conditional FP-trees. This is repeated until a conditional FP-tree with only a single path is obtained. During those recursively repeated processes, all selected items are added to the conditional pattern. Once a single path conditional FP-tree like the one in Figure 4 is obtained, all possible combinations of the items along the path are generated and are combined with the conditional pattern. For example, from those three nodes in the conditional FP-tree in Figure 4, we have $2^3 = 8$ combinations of items 2, 3, and 5: ()

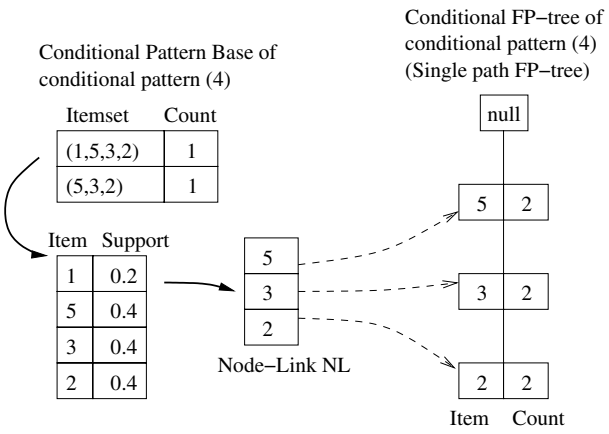

Figure 4: Conditional FP-tree.

(no item), (2), (3), (5), (2,3), (3,5), (5,2), and (2,3,5). Then we obtain frequent itemset patterns based on conditional pattern base (4): (4), (4,2), (4,3), (4,5), (4,2,3), (4,3,5), (4,5,2), and (4,2,3,5).

5.2 Pruning Methods of LPMiner

In this subsection, we introduce three pruning methods that use the SVE property to substantially reduce the search space of frequent itemset patterns and allow LPMiner to efficiently find all itemset patterns that satisfy a length-decreasing support constraint.

5.2.1 Transaction Pruning (TP)

The first pruning method is used to eliminate certain itemsets from conditional pattern bases. Recall that, during frequent itemset pattern generation, the FP-growth algorithm builds a separate FP-tree for all the itemsets that contain the conditional pattern under consideration. Let p be that conditional pattern. If p is infrequent, we know from the SVE property that in order for this conditional pattern to grow to something indeed frequent, it must have a length of at least $f^{-1}(\sigma_D(p))$. Using this requirement, before building the FP-tree corresponding to this conditional pattern, we can eliminate any itemsets whose length is shorter than $f^{-1}(\sigma_D(p)) - |p|$, as such itemsets cannot contribute to a valid frequent itemset pattern in which p is part of it. Since itemsets are sometimes called transactions, we will refer to this as the ***transaction pruning*** method or ***TP*** for short. This pruning method is formally defined as follows:

> **Definition 7 (Transaction Pruning)** *Given a length-decreasing support constraint $f(l)$ and a conditional pattern base D' with regard to a conditional pattern p, an itemset $t \in D'$ can be pruned from D' if $f(|t|+|p|) > \sigma_D(p)$.*

We evaluated the complexity of this method in comparison with the complexity of inserting an itemset to a conditional pattern base. There are three parameters that we need to know in order to prune an itemset t: $|t|$, $|p|$, and $\sigma_D(p)$. We can calculate $|t|$ and $|p|$ in a constant time since the length of an itemset is in the itemset data structure of LPMiner. As for $\sigma_D(p)$, we know this value when we generated the conditional pattern base for the pattern p. Evaluating function f takes a constant time because LPMiner has a lookup table that contains all possible $(l, f(l))$ pairs. Thus, the complexity of this method is just a constant time per inserting an itemset.

5.2.2 Node Pruning (NP)

The second pruning method is used to eliminate certain nodes from each conditional FP-tree. Let us consider a node v of the FP-tree. Let $I(v)$ be the item stored at this node, $\sigma_{D'}(I(v))$ be the support of the item in the conditional pattern base D', and $h(v)$ be the height of the longest path from the root through v to a leaf node. From the SVE property we

know that the node v will contribute to a valid frequent itemset pattern only if

$$h(v) + |p| \geq f^{-1}(\sigma_{D'}(I(v))), \tag{1}$$

where $|p|$ is the length of the conditional pattern of the current conditional FP-tree. Equation (1) is correct because of the following. Any itemset that goes through node v has a length of up to $h(v) + |p|$. From the SVE property, such itemsets must have support at least $f(h(v) + |p|)$ in order to be frequent. Now, if the support of item $I(v)$ is smaller than $f(h(v) + |p|)$, item $I(v)$ cannot contribute any frequent itemsets that go through node v. Thus, if Equation 1 does not hold, node v can be pruned from the FP-tree. Once node v is pruned, then $\sigma_{D'}(I(v))$ will decrease as well as the height of the nodes whose longest path goes through v, possibly allowing further pruning. We will refer to this as the ***node pruning*** method or ***NP*** for short and is formally defined as follows:

> **Definition 8 (Node Pruning)** *Given a length-decreasing support constraint $f(l)$, a conditional pattern base D' with regard to a conditional pattern p, and the FP-tree T built from D', a node v in T can be pruned from T if $h(v) + |p| < f^{-1}(\sigma_{D'}(I(v)))$.*

A key observation to make is that both the TP and NP methods can be used together, as each one of them prunes portions of the FP-tree that the other one does not. In particular, the NP methods can prune a node in a path that is longer than $f^{-1}(\sigma_D(p)) - |p|$, because the item of that node may have lower support than p. On the other hand, TP reduces the frequency of some itemsets in the FP-tree by removing entire short itemsets. For example, consider two itemsets (1, 2, 3, 4) and (1, 2). Let us assume that $f^{-1}(\sigma_D(p)) - |p| = 4$, and each one of the items 1, 2, 3, and 4 has a support equal to that of p. In this case, NP will not remove any nodes, whereas TP will eliminate the second itemset.

Practical Considerations In order to perform the node pruning, we need to compute the height of each node and then traverse each node v to see if it violates Equation 1. If it does, then the node v can be pruned. The height of all the nodes whose longest path goes through v must be decremented by one, and the support of $I(v)$ needs to be decremented to take account of the removal of v. Every time we make such changes in the tree, nodes that could not have been pruned before may now become eligible for pruning. In particular, all the rest of the nodes that have the same item $I(v)$ needs to be rechecked, as well as, all the nodes whose height was decremented upon the removal of v. Our initial experiments with such an implementation showed that the cost of performing the pruning was often quite higher than the saving we achieved when used in conjunction with TP. For this reason we implemented an approximate but fast version of this method that achieves a comparable degree of pruning.

Our approximate NP algorithm initially sorts the itemsets of the conditional pattern base in decreasing itemset length, then traverses each itemset in that order, and tries to insert these itemsets in the FP-tree. Let t be one such itemset. When t is inserted into the FP-tree it may share a prefix with some itemsets already in the FP-tree. However, as soon as the insertion of t results in a new node being created, we check to see if we can prune it using Equation 1. In particular, if v is that newly created node, then $h(v) = |t|$, because the itemsets are inserted into the FP-tree in the decreasing length order. Thus v can be pruned if

$$|t| + |p| < f^{-1}(\sigma(I(v))). \tag{2}$$

If that can be done, the new node is eliminated and the insertion of t continues to the next item. Now if one of the next items inserts a new node u, then that one may be pruned using Equation 2. In Equation 2, we use the original length of the itemset $|t|$, not the length after the removal of the item previously pruned. The reason is that $|t|$ is the correct upper bound of $h(u)$, because one of the itemsets inserted later may have a length of at most $|t|$, the same as the length of the current itemset, and may increase $h(u)$ to $|t|$ at most.

The above approach is approximate because (i) the elimination of a node affects only the nodes that can be eliminated in the subsequent itemsets, not the nodes already in the tree; (ii) we use pessimistic bounds on the height of a

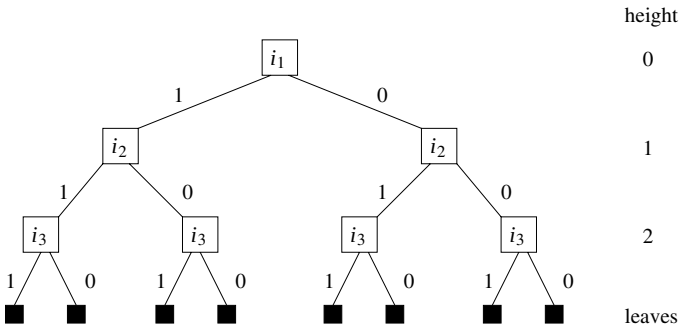

Figure 5: Binary tree when $k = 3$.

node (as discussed in the previous paragraph). This approximate approach, however, does not increase the complexity of generating the conditional FP-tree, beyond the sorting of the itemsets in the conditional pattern base. Since the length of itemsets falls within a small range, they can be sorted in linear time using bucket sort.

5.2.3 Path Pruning (PP)

Once an FP-tree becomes a single path, the original FP-growth algorithm generates all possible combinations of items along the path and concatenates each of those combinations with its conditional pattern. If the path contains k items, there exist a total of 2^k such combinations. Using the SVE property, however, we can limit the number of combinations that we may need to consider.

Let $\{i_1, i_2, \ldots, i_k\}$ be the k items such that $\sigma_{D'}(i_j) \geq \sigma_{D'}(i_{j+1})$, where D' is the conditional pattern base of the single path FP-tree. One way of generating all possible 2^k combinations is to grow them incrementally as follows. First, we create two sets, one that contains i_1, and the other that does not. Next, for each of these sets, we generate two new sets such that, in each pair of them, one contains i_2 and the other does not, leading to four different sets. By continuing this process a total of k times, we will obtain all possible 2^k combinations of items. This approach essentially builds a binary tree with k levels of edges, in which the leaves correspond to the possible combinations. One such binary tree for $k = 3$ is shown in Figure 5.

To see how the SVE property can be used to prune certain subgraphs of this binary tree (and hence combinations to be explored), consider a particular internal node v of that binary tree. Let p be the current conditional pattern, $h(v)$ be the height of node v (the root has a height of zero), and $\beta(v)$ be the number of edges that were one on the path from the root to v. In other words, $\beta(v)$ is the number of items that have been included so far in the set. Using the SVE property we can stop expanding the binary tree under node v if

$$\beta(v) + (k - h(v)) + |p| < f^{-1}(\sigma(I_{h(v)})).$$

Essentially, the above formula states that, based on the frequency of the current item, the set must have a sufficiently large number of items before it can be frequent. If the number of items that were already inserted in the set ($\beta(v)$) plus the number of items that are left for possible insertion ($k - h(v)$) is not sufficiently large, then no frequent itemsets can be generated from this branch of the binary tree, and hence it can be pruned. We will refer to this method as ***path pruning*** method or ***PP*** for short, and is formally defined as follows:

> **Definition 9 (Path Pruning)** *Given k items on a single-path conditional FP-tree with regard to a conditional pattern p and a node v in the binary tree T representing all the combinations of these k items, all the nodes in the subtree rooted at v can be pruned from T if $f(\beta(v) + (k - h(v)) + |p|) > \sigma(I_{h(v)})$.*

The complexity of applying the path pruning scheme for a particular binary tree is constant because (i) for each

node v of the tree we can update in constant times its $\beta(v)$ and $h(v)$ values; (ii) the length of the conditional pattern (*i.e.*, $|p|$) is already known as part of the data structure used to store the pattern; and (iii) $\sigma(I_{h(v)})$ can be calculated using the already computed frequency of v in the current conditional FP-tree.

6 Sequence Mining with Length-Decreasing Support Constraint

To solve the problem of finding frequent sequential patterns that satisfy a length-decreasing support constraint we developed an algorithm called SLPMiner. Our design goals for SLPMiner were to make it both efficient and at the same time sufficiently generic so that any conclusions drawn from our experiments can carry through other database-projection-based sequential pattern mining algorithms [7, 18].

6.1 Sequential Database-Projection-based Algorithm

SLPMiner finds frequent sequential patterns using a database-projection-based approach that was derived from the sequential version [7] of the tree-projection algorithm of Agarwal *et al* [1] for finding frequent itemsets. This algorithm [7] shares the same overall structure with the PrefixSpan [18] algorithm that was independently developed at the same time frame.

Key to this algorithm is the use of a tree to both organize the process of sequential pattern discovery and to represent the patterns that have been discovered thus far. Each node in the tree represents a frequent sequential pattern. The relation between the sequential pattern represented at a particular node at level k and that of its parent at level $k-1$ is that they share the same $k-1$ prefix. That is, the child's pattern is obtained from that of the parent by adding one item at the end. For example, if a node represents a pattern $\langle(1), (2, 3)\rangle$, its parent node represents $\langle(1), (2)\rangle$. The root node of the tree represents the null sequence with no itemset. From the above definition it is easy to see that given a particular node corresponding to pattern p, all the patterns represented in the nodes of the subtree rooted at that node will have p as a prefix. For this reason, we will refer to this tree as the ***prefix tree***.

SLPMiner finds the frequent sequential patterns by growing this tree as follows. It starts from the root node and expands it to create the children nodes that correspond to the frequent items. Then it recursively visits each child node in a depth-first order and expands it into children nodes that represent frequent sequential patterns. SLPMiner grows each pattern in two different ways, namely, ***itemset extension*** and ***sequence extension***. Itemset extension grows a pattern by adding an item to the last itemset of the pattern, where the added item must be lexicographically larger than any item in the last itemset of the original pattern. For example, $\langle(1), (2)\rangle$ is extended to $\langle(1), (2, 3)\rangle$ by itemset extension, but cannot be extended to $\langle(1), (2, 1)\rangle$ or $\langle(1), (2, 2)\rangle$. Sequence extension grows a pattern by adding an item as a new itemset next to the last itemset of the pattern. For example, $\langle(1), (2)\rangle$ is extended to $\langle(1), (2), (2)\rangle$ by sequence extension.

Figure 6 shows a sequential database D and its prefix tree that contains all the frequent sequential patterns given minimum support 0.5. Since D contains a total of four sequences, a pattern is frequent if and only if at least two sequences in D support the pattern. The root of the tree represents the null sequence. At each node of the tree in the figure, its pattern and its supporting sequences in D are depicted together with symbol SE or IE on each edge representing itemset extension or sequence extension respectively.

The key computational step in SLPMiner is that of counting the frequency of the various itemset and sequence extensions at each node of the tree. In principle, these frequencies can be computed by scanning the original database for each one of the nodes. However, this is not cost-effective, especially when the support for each of those extensions is very small. For this reason, SLPMiner creates a projected database for each node of the tree, and uses this projected database (which is usually much smaller) to determine its frequent extensions. The ***projected database*** of a sequential pattern p has only those sequences in D that support p. For example, at the node $\langle(2, 3)\rangle$ in Figure 6, its projected database needs to contain only s_1, s_2, s_4 since s_3 does not support this pattern. Furthermore, we can eliminate preceding items in each sequence that will never be used to extend the current pattern. For example, at the node $\langle(2)\rangle$ in Figure 6,

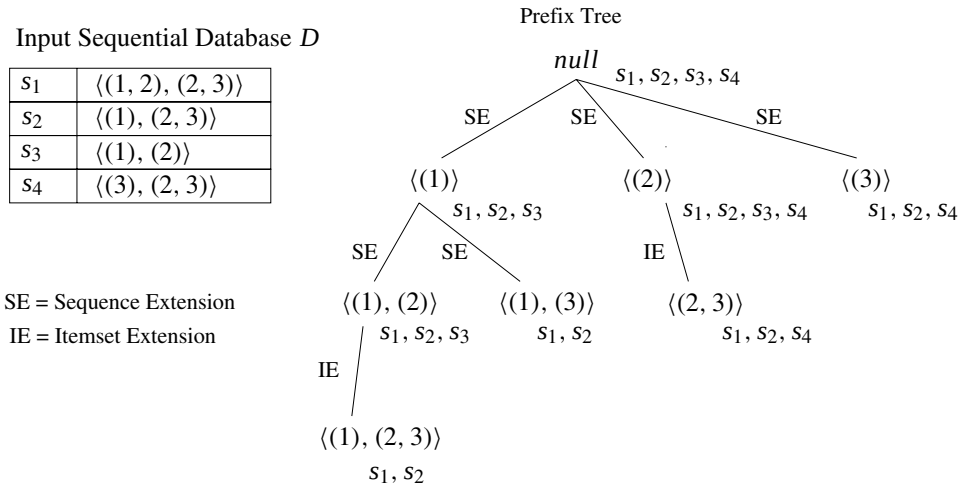

Figure 6: The prefix tree of a sequential database.

we can store sequence $s_1' = \langle(2, 3)\rangle$ instead of s_1 itself in its projected database. Also note that items that do not contribute to a frequent sequence or itemset extension get pruned from all projected databases under that node of the tree. Overall, database projection reduces the amount of sequences that need to be processed at each node and promotes efficient pattern discovery.

6.1.1 Performance Optimizations for Disk I/O

While expanding each node of the tree, SLPMiner performs the following two steps. First, it calculates the support of each item that can be used for itemset extension and each item that can be used for sequence extension by scanning the projected database D' once. Second, SLPMiner projects D' into a projected database for each frequent extension found in the previous step.

Since we want SLPMiner to be able to run against large input sequential databases, the access to the input database and all projected databases is disk-based. To facilitate this, SLPMiner uses two kinds of buffers: a read-buffer and a write-buffer. The read-buffer is used to load a projected database from disk. If the size of a projected database does not fit in the read-buffer, SLPMiner reads part of the database from disk several times. The write-buffer is used to temporally store several projected databases that are generated at a node by scanning the current projected database once using the read-buffer. There are two conflicting requirements concerning how many projected databases we should generate at a time. In order to reduce the number of database scans, we want to generate as many projected databases as possible in one scan. On the other hand, if we keep small buffers for many projected databases simultaneously within the write-buffer, it will reduce the size of the buffer assigned to each projected database, leading to expensive frequent I/O between the write-buffer and disk. In order to balance these two conflicting requirements, SLPMiner calculates the size of each projected database when calculating the support of every item in the current projected database before it actually generates new projected databases. Then, SLPMiner performs a number of database scan, and in each scan, it generates as many projected databases as they can fit in the write-buffer and then writes the entire buffer to the disk. The number of scans depends on the database size and the size of the write buffer. This method also facilitates storing each projected database in a contiguous segment on the disk, allowing us to use fast sequential disk operations which dramatically improve the efficiency of disk I/O.

6.2 Pruning Methods of SLPMiner

In this subsection, we introduce three pruning methods that use the SVE property to substantially reduce the size of the projected databases and allow SLPMiner to efficiently find all sequential patterns that satisfy a length-decreasing

support constraint.

6.2.1 Sequence Pruning (SP)

The first pruning method is used to eliminate certain sequences from the projected databases and is analogous to the transaction pruning method used by LPMiner (Section 5.2.1). Recall from Section 6.1 that SLPMiner generates a projected database at every node. Let us assume that we have a projected database D' at a node N that represents a sequential pattern p. Each sequence in D' has p as its prefix. If p is infrequent, we know from the SVE property that in order for this pattern to grow to something indeed frequent, it must have a length of at least $f^{-1}(\sigma_D(p))$. Now consider a sequence s that is in the projected database at node N, *i.e.*, $s \in D'$. The largest sequential pattern that s can support is of length $|s| + |p|$. Now if $|s| + |p| < f^{-1}(\sigma_D(p))$, then s is too short to support any frequent patterns that have p as prefix. Consequently, s does not need to be considered any further and can be pruned. We will refer to this pruning method as the ***sequence pruning*** method or ***SP*** for short and is formally defined as follows:

> **Definition 10 (Sequence Pruning)** *Given a length-decreasing support constraint $f(l)$ and a projected database D' at a node representing a sequential pattern p, a sequence $s \in D'$ can be pruned from D' if $f(|s| + |p|) > \sigma_D(p)$.*

SLPMiner checks if a sequence needs to be inserted to a projected database just before inserting it onto the write-buffer. We evaluated the complexity of this method in comparison with the complexity of inserting a sequence to a projected database. There are three parameters we need to know to prune a sequence: $|s|$, $|p|$, and $\sigma_D(p)$. As the length of each sequence is part of the sequence data structure in SLPMiner, it takes a constant time to calculate $|s|$ and $|p|$. As for $\sigma_D(p)$, we know this value when we generated the projected database for the pattern p. Evaluating function f takes a constant time because SLPMiner has a lookup table that contains all possible $(l,\ f(l))$ pairs. Thus, the complexity of this method is just a constant time per inserting a sequence.

6.2.2 Item Pruning (IP)

The second pruning method is used to eliminate certain items from each sequence in each projected database. Let us assume that we have a projected database D' at a node v that represents sequential pattern p and consider an item i in a sequence $s \in D'$. From the SVE property we know that the item i will contribute to a valid frequent sequential pattern only if

$$|s| + |p| \geq f^{-1}(\sigma_{D'}(i)), \tag{3}$$

where $\sigma_{D'}(i)$ is the support of item i in D'. This is because of the following. The longest sequential pattern that s can participate in is $|s| + |p|$, and we know that, in the subtree rooted at v, sequential patterns that extend p with item i have support at most $\sigma_{D'}(i)$. Now, from the SVE property, such sequential patterns must have length at least $f^{-1}(\sigma_{D'}(i))$ in order to be frequent. As a result, if Equation 3 does not hold, item i can be pruned from the sequence s. Once item i is pruned, then $\sigma_{D'}(i)$ and $|s|$ decrease, possibly allowing further pruning. Essentially, this pruning method eliminates some of the infrequent items from the short sequences. We will refer to this method as the ***item pruning*** method, or ***IP*** for short and is formally defined as follows:

> **Definition 11 (Item Pruning)** *Given a length-decreasing support constraint $f(l)$ and a projected database D' at a node representing a sequential pattern p, an item i in a sequence $s \in D'$ can be pruned from s if $|s| + |p| < f^{-1}(\sigma_{D'}(i))$.*

Note that item pruning is similar in nature to the node pruning scheme used by LPMiner (Section 5.2.2). However, SLPMiner's version of this pruning method is somewhat less efficient than the corresponding method of LPMiner as the latter by pruning nodes from the FP-tree it prunes items from multiple transactions.

Practical Considerations A simple way to implement this pruning method is as follows: for each projected database D', repeat scanning D' to collect support values of items and scanning D' again to prune items from each sequence until no more items can be pruned. After that, we can project the database into a projected database for each frequent item in the pruned projected database. This algorithm, however, requires multiple scans of the projected database and hence will be too costly as a pruning method.

Instead, we can scan a projected database once to collect support values and use those support values for pruning items as well as for projecting each sequence. Notice that we are using approximate support values that might be higher than the real values since the support values of some items might decrease during the pruning process. SLPMiner applies IP before generating a projected sequence s' of s and after generating s' just before inserting s' into the write-buffer. By applying IP before projecting the sequences, we can reduce the computation of sequence projection. By applying IP once again for the projected sequence s', we can exploit the reduction of length $|s| - |s'|$ to further prune items in s'. Pruning items from each sequence is repeated until no more items can be pruned or the sequence becomes short enough to be pruned by SP.

IP can potentially prune a larger portion of the projected database than SP since it always holds that $\sigma_D(p) \geq \sigma_{D'}(i)$ and hence $f^{-1}(\sigma_D(p)) \leq f^{-1}(\sigma_{D'}(i))$. However, the pruning overhead of IP is much larger than that of SP. Given a sequence s, in the worst case, only one item will be pruned during each iteration over the items in s. Since this can be repeated as many as the number of items in the sequence, the worst case complexity for one sequence is $O(n^2)$ where n is the number of items in the sequence. In our experimental evaluation (Section 7.2) we will see how this overhead affects the total runtime of SLPMiner.

6.2.3 Structure-based Pruning

The sequence and item-pruning schemes that we described so far, do not take advantage of the fact that sequences having the same overall length, can potentially support disjoint sets of sequential patterns depending on size of the itemsets that they contain. For example, consider two size-four sequences $s_1 = \langle(1, 2, 3, 4)\rangle$ and $s_2 = \langle(1), (2), (3), (4)\rangle$. Even though both of them support the same size-one sequences $\langle(1)\rangle$, $\langle(2)\rangle$, $\langle(3)\rangle$, and $\langle(4)\rangle$ they will never simultaneously support any sequences whose size is greater or equal to two. For instance sequence $\langle(1, 2)\rangle$ will only be supported by s_1 whereas sequence $\langle(1), (2)\rangle$ will only be supported by s_2. Motivated by this observation, we considered ways to split a projected database into smaller equivalent classes, such that each class supports a disjoint set of sequential patterns. The advantage of such an approach is that by having smaller databases we may be able to reduce the depth of a certain path from the root to a leaf node of the tree.

As a structure-based pruning, we developed the min-max pruning method. Let p be a sequential pattern at a particular node, D' be its projected database, and assume that p is infrequent (*i.e.*, $\sigma_D(p) < f(|p|)$). From the SVE property, in order for p to become frequent, we need to grow p by adding at least $f^{-1}(\sigma_D(p)) - |p|$ items. Now, consider the following two values that are defined for each sequence $s \in D'$.

1. $a(s)$ = the smallest number of itemsets in s that need to be used to grow p by $f^{-1}(\sigma_D(p)) - |p|$ items.
2. $b(s)$ = the number of itemsets in s.

These two values define an interval $[a(s), b(s)]$, that we call the ***min-max interval*** of sequence s. If two sequences $s, s' \in D'$ satisfy $[a(s), b(s)] \cap [a(s'), b(s')] = \emptyset$, then s and s' cannot support any common sequential pattern since their min-max intervals are disjoint. Motivated by this observation, the basic idea of the min-max pruning is to split the projected database D' into two databases D'_1 and D'_2 such that they contribute to two disjoint sets of frequent sequential patterns.

If there exists D'_1 and D'_2 that satisfy $\cup_{s \in D'_1}[a(s), b(s)] \cap \cup_{s \in D'_2}[a(s), b(s)] = \emptyset$, then D'_1 and D'_2 support distinct sets of frequent sequential patterns. However, in general,this is impossible. Instead, D' will be split into three sets A, B, C of sequences as shown in Figure 7. More precisely, these three sets are defined for some positive integer k as

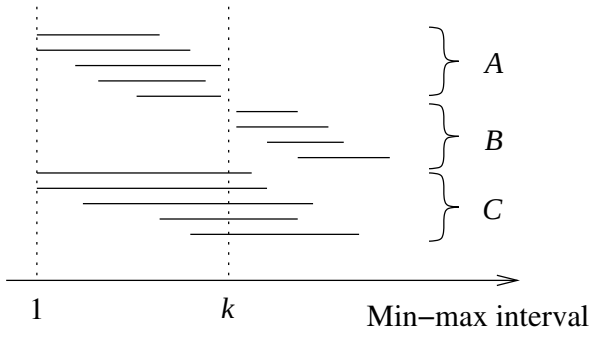

Figure 7: Min-max intervals of a set of sequences.

follows.

$$\begin{aligned} A(k) &= \{s | s \in D' \wedge b(s) < k\} \\ B(k) &= \{s | s \in D' \wedge a(s) \geq k\} \\ C(k) &= D' - (A \cup B) \end{aligned}$$

$A(k)$ and $B(k)$ support distinct sets of frequent sequential patterns, whereas $A(k)$ and $C(k)$ as well as $B(k)$ and $C(k)$ support overlapping sets of frequent sequential patterns. From these three sets, we form $D'_1 = A(k) \cup C(k)$ and $D'_2 = B(k) \cup C(k)$. If we mine frequent sequential patterns of length up to $k-1$ from D'_1 and patterns of length no less than k from D'_2, we will obtain the same patterns as we would from original D'.

Throughout our experiments, we observed that $|C|$ is usually close to $|D'|$; thus, mining D'_1 and D'_2 separately will cost more than mining the original database D'. We can, however, prune the entire D' if both $|D'_1|$ and $|D'_2|$ are smaller than $\min_{l \geq 1} f(l)$. Furthermore, we can increase this minimum support by the fact that any sequential patterns that the current pattern p can extend to is of length at most $\max_{s \in D'}(|s|) + |p|$. Now, from the SVE property, we know that if both $|D'_1|$ and $|D'_2|$ are smaller than $f^{-1}(\max_{s \in D'}(|s|) + |p|)|D|$, we can eliminate the entire D'. Essentially, this means that if we can split a projected database into two subsets each of which is too small to be able to support any frequent sequential pattern, then we can eliminate the entire original projected database. We will refer to this method as the ***min-max pruning*** method or ***MP*** for short and is formally defined as follows:

> **Definition 12 (Min-Max Pruning)** *Given a length-decreasing support constraint $f(l)$ and a projected database D' at a node representing a sequential pattern p, the entire D' can be pruned if there exists a positive integer k such that*
>
> $$\begin{aligned} |D'_1| &= |A(k)| + |C(k)| < f(\max_{s \in D'}(|s|) + |p|)|D|, \text{ and} \\ |D'_2| &= |B(k)| + |C(k)| < f(\max_{s \in D'}(|s|) + |p|)|D|. \end{aligned}$$

Practical Considerations We apply MP just after a new projected database D' is generated if the entire sequences in D' is still kept in the write-buffer and if $|D'| \leq 1.2 f(\max_{s \in D'}(|s|) + |p|)|D|$. The first condition is necessary to avoid costly disk I/O and the second condition is necessary to increase the probability of successfully eliminating the projected database.

The algorithm for MP consists of two parts. The first part calculates the distribution of the number of sequences over possible min-max intervals. The second part finds a positive integer k that satisfies the above two equations. The first part requires scanning D' once and finding the min-max interval for each sequence. For each sequence s, SLPMiner determines $a(s)$ as the smallest number of the largest itemsets whose sizes add up to at least $f^{-1}(\sigma_D(p)) - |p|$. The

other value $b(s)$ is simply the number of itemsets in s. This part requires $O(m)$ where m is the total number of itemsets in D'. The second part uses an $n \times n$ upper triangular matrix $Q = (q_{ij})$ where $q_{ij} = |\{s | a(s) = i \wedge b(s) = j \wedge s \in D'\}|$ and n is the maximum number of itemsets in a sequence in D'. Matrix Q is generated during the database scan of the first part. Given matrix Q, we have

$$\begin{aligned} |A(k)| + |C(k)| &= \sum_{i=1}^{k-1} \sum_{j=i}^{n} q_{ij} \\ |B(k)| + |C(k)| &= \sum_{j=k}^{n} \sum_{i=1}^{j} q_{ij} \end{aligned}$$

Using the relations

$$\begin{aligned} (|A(k+1)| + |C(k+1)|) - (|A(k)| + |C(k)|) &= \sum_{j=k}^{n} q_{kj} \\ (|B(k+1)| + |C(k+1)|) - (|B(k)| + |C(k)|) &= -\sum_{i=1}^{k} q_{ik} \end{aligned}$$

we can calculate $|A(k)| + |C(k)|$ and $|B(k)| + |C(k)|$ incrementally for all k in $O(n^2)$. So the overall complexity of the min-max pruning for one projected database is $O(m + n^2)$. In some cases, this complexity may be much larger than the runtime reduction achieved by eliminating the projected database. However, our experimental results show that the min-max pruning method alone can substantially reduce the total runtime.

7 Experimental Evaluation

We experimentally evaluated the performance and effectiveness of the LPMiner and the SLPMiner algorithms for finding frequent patterns that satisfy a length-decreasing support constraint on a variety of synthetic datasets. These datasets were generated by the widely used itemset and sequence generators that are provided by the IBM Quest group and were initially used to evaluate the Apriori [3] and AprioriAll [21] algorithms.

These experiments were performed on two classes of workstations. In particular, the experiments with LPMiner were performed on Intel-based Linux workstations with Pentium III at 600MHz and 1GB of main memory, whereas the experiments with SLPMiner were performed on Linux workstations with AMD Athlon at 1.5GHz and 3GB of main memory. All the reported runtimes are in seconds.

7.1 Experimental Evaluation of LPMiner

We used two classes of datasets DS1 and DS2. Both of them contained 100K itemsets. For each of the two classes we generated different problem instances as follows. For DS1, we varied the average size of the itemsets from 10 to 34 in increments of two, obtaining a total of 13 different datasets, DS1-10, DS1-12, $\cdots$, DS1-34. For DS2, we varied the average size of the itemsets from 10 to 28 in increments of two, obtaining a total of 10 different datasets, DS2-10, DS2-12, $\cdots$, DS2-28. For each problem instance in both of DS1-x and DS2-x, we set the average size of the potentially maximal long itemset to be $x/2$, so as x increases, the dataset contains potentially longer frequent itemsets. The difference between DS1 and DS2 is that each DS1-x dataset contains 1000 distinct items, whereas each DS2-x dataset contains 5000 distinct items. Consequently, for sufficiently small values of support, the expected number of frequent itemsets for DS2-x will be greater than that for DS1-x. The characteristics of these datasets are summarized in Table 1.

In all of our experiments we used minimum support constraints that decrease linearly with the length of the frequent

Parameter	Description	DS1	DS2
$\|D\|$	Number of itemsets	100K	100K
$\|T\|$	Average size of the itemsets	3 to 34	3 to 28
$\|I\|$	Average size of the maximal potentially long itemsets	$\|T\|/2$	$\|T\|/2$
$\|L\|$	Number of maximal potentially large itemsets	10000	10000
N	Number of distinct items	1000	5000

Table 1: Parameters of datasets for LPMiner.

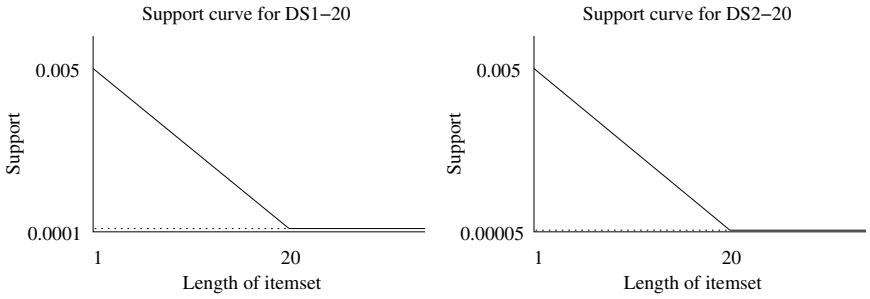

Figure 8: Support curve for DS1-20 and DS2-20.

itemsets. In particular, for each of the DS1-x datasets, the initial value of support was set to 0.005 (representing a minimum occurrence frequency of 500) and it was decreased linearly down to 0.0001 (representing a minimum occurrence frequency of 10) for itemsets of length up to x. For longer itemsets the support was kept fixed at 0.0001. In the case of the DS2 class of datasets, we used a similar approach to generate the constraint; however, instead of using 0.0001 as the minimum support, we used 0.00005 (representing a minimum occurrence frequency of 5). Figure 8 graphically illustrates these two length-decreasing support constraints for DS1-20 and DS2-20.

7.1.1 Results

Table 2 shows the experimental results that we obtained for the DS1 and DS2 datasets. Each row of the table shows the results obtained for a different DS1-x or DS2-x dataset, specified on the first column. The remaining columns show the amount of time required by different itemset discovery algorithms. The column labeled "FP-growth" shows the amount of time taken by the original FP-growth algorithm using a constant support constraint that corresponds to the smallest support of the support curve, *i.e.*, 0.0001 for DS1, and 0.00005 for DS2. The other columns show the amount of time required by LPMiner that uses the length-decreasing support constraint and a total of seven different combinations of pruning methods. For example, the column label "NP" corresponds to the scheme that uses only node pruning, whereas the column labeled "NP+TP+PP" corresponds to the scheme that uses all the three pruning methods (*i.e.*, node, transaction, and path pruning). Note that values with a "–" correspond to experiments that were aborted because they were taking too long time.

A number of observations can be made from the results in these tables. First, either one of the LPMiner methods performs better than the FP-growth algorithm. In particular, the LPMiner that uses all three pruning methods does the best, requiring substantially smaller time than the FP-growth algorithm. For DS1, it is about 2.2 times faster for DS1-10, 8.2 times faster for DS1-20, 33.4 times faster for DS1-30, and 105 times faster for DS1-34. Similar trends can be observed for DS2, in which the performance of LPMiner is 4.2 times faster for DS2-10, 21.0 times faster for DS2-20, and 58.5 times faster for DS2-26. Second, the performance gap between FP-growth and LPMiner increases as the length of the discovered frequent itemset increases (recall that for both DS1-x and DS2-x, the length of the frequent itemsets tend to increase with x). This is due to the fact that the overall number of patterns that LPMiner can prune becomes larger, leading to improved relative performance. Third, comparing the different pruning methods in isolation, we can see that NP and TP lead to the largest runtime reduction and PP achieves the smallest reduction. This

Dataset	FP-growth	LPMiner						
		NP	TP	PP	NP+TP	NP+PP	TP+PP	NP+TP+PP
DS1-10	40.42	21.36	25.03	40.38	18.32	21.29	25.03	18.34
DS1-12	71.09	32.80	35.72	70.92	27.88	32.64	35.59	27.87
DS1-14	130.60	47.86	48.31	125.70	40.55	47.59	48.26	40.38
DS1-16	255.52	67.79	68.52	253.89	56.46	67.16	64.06	60.44
DS1-18	409.96	85.85	80.07	404.51	71.29	84.64	79.17	71.04
DS1-20	730.39	113.98	105.25	711.94	93.82	110.49	101.09	89.35
DS1-22	1224.41	145.25	141.53	1180.97	117.60	137.18	133.18	109.37
DS1-24	1840.37	183.51	191.36	1739.31	142.72	174.06	184.60	134.17
DS1-26	3465.20	287.81	306.50	3134.16	212.42	226.66	259.93	166.95
DS1-28	7512.34	2142.16	1911.44	4646.93	1733.97	300.82	362.57	210.95
DS1-30	8884.68	431.02	534.11	7370.50	338.81	397.33	489.12	266.11
DS1-32	31063.53	11001.17	8289.84	12143.14	7943.06	547.12	676.44	361.11
DS1-34	51420.51	16214.51	10990.93	18027.93	10446.44	751.23	905.89	487.83
DS2-10	146.91	37.37	47.84	147.16	34.55	37.15	47.75	35.10
DS2-12	275.83	47.29	66.48	274.81	43.65	46.97	66.04	43.28
DS2-14	475.75	58.44	90.50	471.67	53.75	56.39	88.98	52.97
DS2-16	812.48	80.11	125.09	798.52	72.07	77.16	122.50	70.39
DS2-18	1280.64	100.25	165.01	1252.84	93.05	91.61	160.60	86.79
DS2-20	2359.50	143.24	223.24	2282.95	125.45	116.60	207.55	112.24
DS2-22	3592.04	229.33	315.03	3388.12	186.41	150.00	267.46	139.40
DS2-24	5137.13	313.26	373.71	4676.63	208.68	186.62	336.51	173.38
DS2-26	12974.80	2297.73	2094.52	10022.34	1884.83	241.35	426.62	221.59
DS2-28	–	8431.51	7149.52	–	6977.56	328.28	551.04	296.88

Table 2: Comparison of pruning methods of LPMiner using DS1 and DS2.

is not surprising as PP can only prune itemsets during the late stages of itemset generation. Finally, the runtime with three pruning methods increases gradually as the average length of the itemsets (and the discovered itemset patterns) increases, whereas the runtime of the original FP-growth algorithm increases exponentially.

To evaluate how the performance of LPMiner scales with the size of the database we performed an experiment in which we varied the number of itemsets from 50K to 200K for the DS1-26 and DS2-26 datasets. The amount of time required by LPMiner on these datasets is shown in Figure 9. From these results we can see that LPMiner scales sub-linearly with the size of the database. For instance, for DS1-26, the runtime increases by a factor of 3.15 when the database increases by a factor of 4. The reason for that is that FP-tree is able to achieve a somewhat higher compression rate; thus, speeding up the overall computations. Note that the small variability in the performance obtained for DS2-26 as we increase the number of itemsets is due to the fact that the number of frequent patterns discovered in each distance was somewhat different.

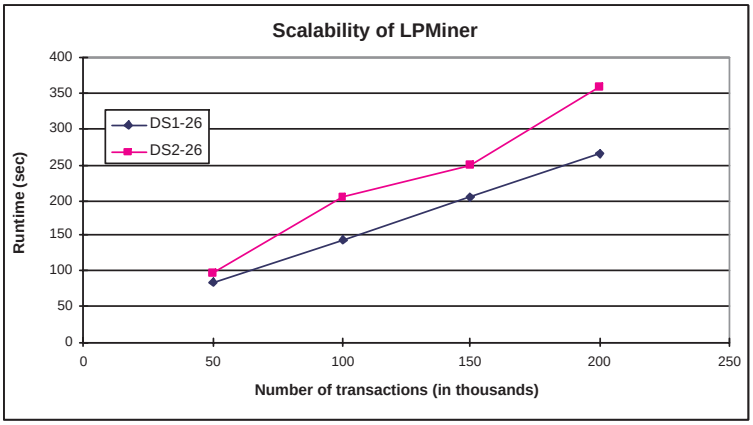

Figure 9: Runtime performance of LPMiner as the number of transactions increases from 50K to 200K.

7.2 Experimental Evaluation of SLPMiner

We primarily used two classes of datasets DS3 and DS4, each of which contained 25K sequences. For each of the two classes we generated different problem instances as follows. For DS3 we varied the average number of itemsets in a sequence from 10 to 30 in increments of two, obtaining a total of 11 different datasets, DS3-10, DS3-12, $\cdots$, DS3-30. For DS4 we varied the average number of items in an itemset from 2.5 to 7.0 in increments of 0.5, obtaining a total of 10 different datasets, DS4-2.5, DS4-3.0, $\cdots$, DS4-7.0. For DS3-x we set the average size of maximal potentially frequent sequences to be $x/2$. For DS4-x we set the average size of maximal potentially frequent itemsets to be $x/2$. Thus, the dataset contains longer frequent patterns as x increases. The characteristics of these datasets are summarized in Table 3.

Parameter	Description	DS3	DS4
$\|D\|$	Number of sequences	25000	25000
$\|C\|$	Average number of itemsets per sequence	$x = 10, 12, \cdots, 30$	10
$\|T\|$	Average number of items per itemset	2.5	$x = 2.5, 3.0, \cdots, 7.0$
N	Number of items	10000	10000
$\|S\|$	Average size of maximal potentially frequent sequences	$x/2$	5
$\|I\|$	Average size of maximal potentially frequent itemsets	1.25	$x/2$

Table 3: Parameters for datasets for SLPMiner.

In addition to the above datasets, we also performed a limited number of experiments with another dataset DS5, for which the parameters were set as follows: $|D| = 25000$, $|C| = 20$, $|T| = 10$, $N = 10000$, $|S| = 10$, and $|I| = 5$. This dataset contains much longer sequences than DS3 and DS4 and was used to evaluate the overheads associated with the various pruning methods.

In all of our experiments, we used a minimum support constraint that decreases linearly with the length of the frequent sequential pattern. In particular, the initial value of support was set to 0.01 (representing a minimum occurrence frequency of 25) and it was decreased linearly down to 0.001 (representing a minimum occurrence frequency of 3) for sequences of up to length $\lfloor |C||T|/2 \rfloor$. For longer sequences, the support was kept fixed at 0.001. Note that $|C||T|$ represents the average size of each sequence.

We also ran SPADE [25] to compare runtime values with SLPMiner. When running SPADE, we used the depth first search option, which leads to better performance than the breadth first search option on our datasets. We set the minimum support value to be equal to 0.0001 (*i.e.*, the smallest support value of the length-decreasing support constraint curve).

For SLPMiner, we set the size of the read-buffer to 10MB and the write-buffer to 300MB. Similarly, we set the available memory size to 310MB for SPADE.

7.2.1 Results

Table 4 shows the experimental results that we obtained for the DS3 and DS4 datasets. Each row of the table shows the results obtained for a different DS3-x or DS4-x dataset, specified on the first column. The column labeled "SPADE" shows the amount of time taken by SPADE. The column labeled "None" shows the amount of time taken by SLPMiner using a constant support constraint that corresponds to the smallest support of the support curve, that is 0.0001 for all datasets. The other columns show the amount of time required by SLPMiner that uses the length-decreasing support constraint and a total of five different combinations of pruning methods. For example, the column label "SP" corresponds to the pruning scheme that uses only sequence pruning, whereas the column labeled "SP+IP+MP" corresponds to the scheme that uses the sequence, the item, and the min-max pruning methods. Note that values with a "–" correspond to experiments that were aborted because they were taking too long time.

A number of interesting observations can be made from the results in these tables. First, even though SLPMiner

Dataset	SPADE	SLPMiner					
		None	SP	IP	MP	SP+IP	SP+IP+MP
DS3-10	10.56	20.21	11.51	11.57	12.64	12.00	11.83
DS3-12	18.24	41.42	15.31	15.43	17.80	15.35	15.93
DS3-14	46.21	98.35	21.29	21.58	24.45	21.42	21.29
DS3-16	87.28	208.18	27.34	26.63	31.23	26.18	27.38
DS3-18	273.32	592.88	39.22	39.03	43.49	38.79	40.17
DS3-20	594.77	1438.93	46.14	48.44	54.72	47.86	47.72
DS3-22	4702.69	8942.94	63.35	65.12	74.90	65.23	65.90
DS3-24	–	–	82.75	85.62	94.64	82.37	83.14
DS3-26	–	–	106.98	112.18	126.64	111.69	106.56
DS3-28	–	–	139.36	142.76	162.06	137.95	138.41
DS3-30	–	–	180.71	189.02	212.84	185.60	184.10
DS4-2.5	10.56	20.21	11.51	11.57	12.64	12.00	11.83
DS4-3.0	21.15	45.88	16.62	16.94	18.71	15.87	15.90
DS4-3.5	117.48	279.61	31.85	35.31	43.26	31.44	31.69
DS4-4.0	333.78	899.02	32.78	32.48	39.80	31.94	32.10
DS4-4.5	731.40	1784.57	35.87	37.95	43.13	38.03	36.53
DS4-5.0	6460.64	17106.37	57.67	61.65	77.83	59.11	59.09
DS4-5.5	–	–	59.50	62.61	73.75	61.18	61.79
DS4-6.0	–	–	77.75	78.68	96.95	77.92	75.18
DS4-6.5	–	–	98.06	105.47	144.38	101.21	102.18
DS4-7.0	–	–	116.98	119.90	136.51	113.44	117.60

Table 4: Comparison of pruning methods of SLPMiner using DS3 and DS4.

without any pruning method is slower than SPADE, the relative performance difference remains stable ranging from 1.9 to 2.7 with an average value of 2.3. This shows that the performance of SLPMiner is comparable to SPADE and a reasonably good platform for evaluating our pruning methods. Second, either one of pruning methods performs better than SLPMiner without any pruning method. In particular, SP, IP, SP+IP, and SP+IP+MP have almost the same speedup. For DS3, the speedup by SP is about 1.8 times faster for DS3-10, 7.6 times faster for DS3-16, and 141.2 times faster for DS3-22. Similar trends can be observed for DS4, in which the performance of SLPMiner with SP is 1.8 times faster for DS4-2.5, 8.8 times faster for DS4-3.5, and 296.6 times faster for DS4-5.0.

Third, comparing the different pruning methods in isolation, we can see that SP leads to the largest runtime reduction, IP leads to the second largest runtime reduction, and MP achieves the smallest reduction. The reason for this somewhat worse performance of MP is primarily due to the overhead of splitting a database into two subsets. Despite that, it seems surprising to gain such a great speedup by MP alone. This shows a large part of the runtime of SLPMiner without any pruning method is accounted for by many small projected databases that never contribute to any frequent patterns. As for SP and IP, SP is slightly better than IP because IP and SP prune almost the same amount of projected databases for those datasets but IP has much larger overhead than SP. Fourth, the runtime with the three pruning methods increases gradually as the average length of the sequences (and the discovered patterns) increases, whereas the runtime of SLPMiner without any pruning increases exponentially.

	SP	IP	SP+IP	SP+IP+MP
Runtime	15939.38	16019.34	15103.93	15205.96
Projected Database Size (GB)	65.99	47.50	43.20	41.35

Table 5: Comparison of pruning methods of SLPMiner using DS5.

Table 5 shows the runtime and projected database size for the DS5 dataset. We tested SP, IP, SP+IP, SP+IP+MP for DS5 since they were the best when applied to DS3 and DS4 datasets. Even though the projected database size of IP is 1.5 times smaller than that of SP, SP and IP achieve almost the same runtime again because of the large overhead of IP. These two methods, however, can achieve the best runtime when combined as SP+IP because IP does not have to

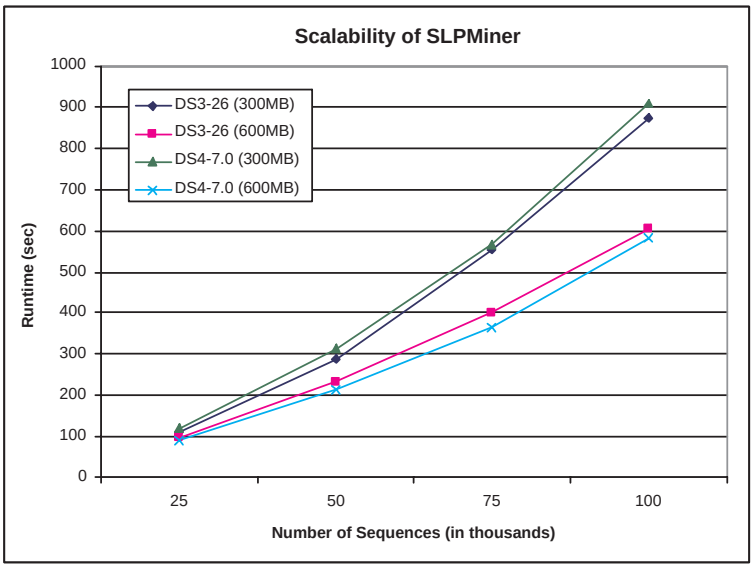

Figure 10: Runtime performance of SLPMiner as the number of transactions increases from 25K to 100K.

prune the part of projected databases for which SP can prune. Since DS5 contains much longer sequences than DS3 and DS4 datasets, there are more opportunities for IP to prune where SP does not work.

Finally, to evaluate how the performance of SLPMiner scales with the size of the database we performed an experiment in which we varied the number of sequences from 25K to 100K for the DS3-26 and DS4-7.0 datasets. The amount of time required by SLPMiner on these datasets is shown in Figure 10. Specifically, Figure 10 shows two sets of results. The first set was obtained using a 300MB buffer of disk I/O operations, whereas the second set was obtained using a 600MB buffer. From these results we can see that SLPMiner scales somewhat worse than linearly with the size of the database. For instance, for "DS3-26 (300MB)", the runtime increases by a factor of 7.85 when the database increases by a factor of 4, whereas in the case of "DS3-26 (600MB)", the corresponding runtime increases by a factor of 6.32. These results suggest that the reason for the worse than linear performance is due to the fact that we are using a fixed-size buffer for disk I/O operations. As the database increases, this buffer is not sufficient to store the databases (as they are being projected), leading to multiple passes and somewhat lower performance.

8 Conclusion

In this paper, we presented two algorithms, LPMiner and SLPMiner, that can efficiently find all frequent itemset or sequential patterns that satisfy a length-decreasing support constraint. For each of these two algorithms, we developed three pruning methods that improved the performance by up to two orders of magnitude. The key insight that allowed us to achieve such high performance was the introduction of length-decreasing support constraints and the smallest valid extension property that frequent patterns must have in order to satisfy such support constraints.

The pruning methods are not specific to these two algorithms but almost all of them can be incorporated into other algorithms for itemset or sequential pattern discovery. For example, the tree-projection algorithm [1] can use the transaction pruning and item pruning in a straight forward way since the basic approach of this algorithm is very close to that of FP-growth [8] except that FP-growth uses the FP-tree data structure. It is straight-forward to implement all the three pruning methods of SLPMiner in PrefixSpan [18] with disk-based projection. PrefixSpan with pseudo-projection can use the sequence pruning method. Even SPADE [25], which has no explicit sequence representation during pattern mining, can use the sequence pruning method by adding the length of a sequence to each record in the vertical database representation.

Acknowledgments

We will like to thank Dr. Ramesh Agarwal from IBM research for introducing us to the problem of pattern mining under length-decreasing support constraints. We also will like to thank Prof. Mohammed Zaki from RPI for providing us with an implementation of the SPADE algorithm used in the experimental evaluation of SLPMiner.

References

[1] R. Agarwal, C. Aggarwal, V. Prasad, and V. Crestana. A tree projection algorithm for generation of large itemsets for association rules. *IBM Research Report*, RC21341, November 1998.

[2] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, pages 85–93, Seattle, Washington, June 1998.

[5] S. Brin, R. Motwani, and C. Silversteim. Beyond market baskets: Generalizing association rules to correlations. In *Proc. of 1997 ACM-SIGMOD Int. Conf. on Management of Data*, Tucson, Arizona, 1997.

[6] Edith Cohen, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D. Ullman, and Cheng Yang. Finding interesting associations without support pruning. In *ICDE*, pages 489–499, 2000.

[7] Valerie Guralnik, Nivea Garg, and George Karypis. Parallel tree projection algorithm for sequence mining. In *European Conference on Parallel Processing*, pages 310–320, 2001.

[8] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

[9] Jiawei Han, Jianyong Wang, Ying Lu, and Petre Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *IEEE International Conference on Data Mining*, 2002.

[10] D. Lin and Z. Kedem. Princer-Search: An efficient algorithm for discovering the maximum frequent set. In *Proceedings of the 6th International Conference on Extending Database Technology*, 1998.

[11] Bing Liu, Wynne Hsu, and Yiming Ma. Mining association rules with multiple minimum supports. In *SIGKDD 1999*, 1999.

[12] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, College Park, MD, 1995.

[13] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. of 1995 ACM-SIGMOD Int. Conf. on Management of Data*, 1995.

[14] J.S. Park, M.S. Chen, and P.S. Yu. Efficient parallel data mining for association rules. In *Proceedings of the 4th Int'l Conf. on Information and Knowledge Management*, 1995.

[15] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discoverying frequent closed itemsets for association rules. In *7th International Conference on Database Theory*, pages 398–416, 1999.

[16] Jain Pei and Jiawei Han. Can we push more constraints into frequent pattern mining? In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2000.

[17] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 of ACM-SIGMOD Int. Workshop on Data Mining and Knowledge Discovery*, 2000.

[18] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *ICDE*, pages 215–224, 2001.

[19] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st VLDB Conference*, pages 432–443, Zurich, Switzerland, 1995.

[20] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[21] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns. In *11th Int. Conf. Data Engineering*, 1995.

[22] Ke Wang, Yu He, and Jiawei Han. Mining frequent itemsets using support constraints. In *The VLDB Journal*, pages 43–52, 2000.

[23] M. Zaki. Generating non-redundant association rules. In *6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 34–43, 2000.

[24] Mohammed J. Zaki. SPADE: an efficient algorithms for mining frequent sequences. *Machine Learning Journal*, 42:31–60, 2001.

[25] Mohammed Javeed Zaki. Fast mining of sequential patterns in very large databases. Technical Report 668, Department of Computer Science, Rensselaer Polytechnic Institute, 1997.

[26] Mohammed Javeed Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[27] Mohammed Javeed Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, RPI, 2001.