HKU CSIS Tech Report TR-2002-06

Efficient Algorithms for Incremental Update of Frequent Sequences

Minghua Zhang Ben Kao David Cheung Chi-Lap Yip

Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong.
{*mhzhang, kao, dcheung, clyip*}*@csis.hku.hk*

Abstract

Agrawal and Srikant first put forward the problem of mining frequently occurring sequences from a customer database [1]. In their model, a customer database consists of a set of sequences. Each sequence is a chronologically ordered set of transactions (or *itemsets*). In real life, the underlying database is changed from time to time due to the insertion and deletion of transactions. This paper studies the problem of incremental maintenance of frequent sequences when the underlying database is modified over time. We propose two *incremental* algorithms `GSP+` and `MFS+` that can efficiently compute the updated set of frequent sequences given the set of frequent sequences obtained from mining the old database. Through experiments, we compare the performance of `GSP+` and `MFS+` with `GSP` and `MFS` — two efficient algorithms for mining frequent sequences. We show that `GSP+` and `MFS+` effectively reduce the CPU costs of their counterparts with only a small or even negative expense on I/O cost.

keywords: data mining, sequence, incremental update

1 Introduction

One of the many data mining problems is mining frequent sequences from transactional databases. The goal is to discover frequent sequences of events. The problem was first introduced by Agrawal and Srikant [1]. In their model, a database is a collection of transactions. Each transaction is a set of items (or an itemset) and is associated with a customer ID and a time ID. If one groups the transactions by their customer IDs, and then sorts the transactions of each group by their time IDs in increasing value, the database is transformed into a number of customer sequences. Each customer sequence shows the order of transactions a customer has conducted. Roughly speaking, the problem of mining frequent sequences is to discover "subsequences" (of itemsets) that occur frequently enough among all the customer sequences.

Sequence mining finds its application in many different areas. For example, an on-line bookstore may find that most customers who have purchased the book "The Gunslinger" are likely to come back again in the future to buy "The Gunslinger II" in another transaction. As another example, a web site manager may mine the web-log sequences to find the visitors' access pattern and improve the site design. A few efficient algorithms for mining frequent sequences have been proposed, notably, `AprioriAll` [1], `GSP` [13], `SPADE` [15], `MFS` [16] and `PrefixSpan` [10]. The above studies assume the database is static, and even a small change in the database

will require the algorithms to run again to get the updated frequent sequences.

In practice, the content of a database changes over time. For example, new customer sequences are added to a bookstore's database as the store recruits new customers. Similarly, every visit to a web site will add a new log to the site's log database. There are also cases in which we have to delete sequences from the database. As an example, when mining current access patterns of a website, we may need to delete some out-of-date logs (say one year before). Since an operational database changes continuously, data mining has to be performed repeatedly. If each time we have to rerun the mining algorithms from scratch, it will be very inefficient.

In this paper we study the problem of incremental maintenance of frequent sequences. We assume that a mining exercise has been performed on an *old* database to obtain a set of frequent sequences. The old database is then updated by inserting new sequences and/or by deleting some old sequences. Our objective is to discover the set of frequent sequences in the *new* database efficiently, taking advantage of information that was obtained in a previous mining exercise. We propose two new algorithms `GSP+` and `MFS+` to solve the problem. The two algorithms are modified versions of `GSP` [13] and `MFS` [16]. The modification is made based on the following observations:

- If one knows about the support of a *frequent* sequence in the old database, then the sequence's support w.r.t. the new database can be deduced by scanning the inserted customer sequences and the deleted customer sequences. The portion (typically the majority) of the database that has not been changed needs not be processed.
- Given that a sequence is infrequent w.r.t. the old database, the sequence cannot become frequent unless its support in the inserted customer sequences is *large enough* and that its support in the deleted customer sequences is *small enough*. This observation allows us to determine whether a candidate sequence should be considered by "looking" at the *small* portion of the database that has been changed.
- If the old database and the new database share a non-trivial set of common sequences, then the set of frequent sequences found in the old database gives a good indication of which sequences in the new database are *likely* to be frequent. As we will see later, this allows us to generate long candidate sequences fairly early on in the mining algorithm. The effect is that fewer passes over the data are required compared with the mine-from-scratch approach. We can thus effectively reduce the I/O cost.

The rest of this paper is organized as follows. In Section 2 we give a formal definition of the maintenance problem. In Section 3, we review some related works. In particular, we briefly review `GSP` and `MFS`, based on which we derive `GSP+` and `MFS+`. The two new algorithms are presented in Section 4. Experiment results comparing the performance of the algorithms are shown in Section 5. Finally, we conclude the paper in Section 6.

2 Model

In this section, we give a formal statement of the maintenance problem. We also define some notations to simplify our discussion.

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called items. An itemset X is a set of items

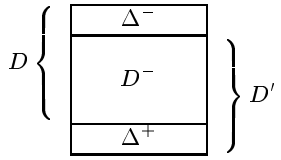


Figure 1: Definitions of D, D', Δ^-, D^- and Δ^+

(hence, $X \subseteq I$). A sequence $s = \langle t_1, t_2, \ldots, t_n \rangle$ is an ordered set of transactions, where each transaction t_i $(i = 1, 2, \ldots, n)$ is an itemset.

The length of a sequence s is defined as the number of items contained in s. (If an item occurs several times in different itemsets of a sequence, the item is counted for each occurrence.) We use $|s|$ to represent the length of s. For example, if $s = \langle \{1\}, \{2, 3\}, \{1, 4\} \rangle$, then $|s| = 5$. A sequence of length k is called a k-sequence.

Given two sequences $s_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $s_2 = \langle b_1, b_2, \ldots, b_l \rangle$, we say s_1 contains s_2 (or equivalently s_2 is a subsequence of s_1) if there exist integers $j_1, j_2, \ldots, j_l$, such that $1 \leq j_1 < j_2 < \ldots < j_l \leq n$ and $b_1 \subseteq a_{j_1}, b_2 \subseteq a_{j_2}, \ldots, b_l \subseteq a_{j_l}$. We represent this relationship by $s_2 \sqsubseteq s_1$.

In a sequence set V, a sequence $s \in V$ is *maximal* if s is not a subsequence of any other sequence in V. That is,

$$s \text{ is maximal if } \nexists s' \text{ s.t. } (s' \in V) \wedge (s' \neq s) \wedge (s \sqsubseteq s').$$

Given a sequence set V and a sequence s, if there exists a sequence $s' \in V$ such that $s \sqsubseteq s'$, we write $s \vdash V$. Given a database D of sequences, the *support count* of a sequence s, denoted by δ_D^s, is defined as the number of sequences in D that contain s. The *fraction* of sequences in D that contain s is called the *support* of s. If we use the symbol $|D|$ to denote the number of sequences in D (or the size of D), we have

$$\text{support of } s = \delta_D^s / |D|.$$

If the support of s is no less than a user specified support threshold ρ_s, s is a frequent sequence. The problem of mining frequent sequences is to find all *maximal* frequent sequences given a database D. We use the symbol L_i to denote the set of all length-i frequent sequences. Also, we use L to denote the set of all frequent sequences. That is $L = \bigcup_{i=1}^{\infty} L_i$.

Given a database D, we assume that a previous mining exercise has been executed to obtain the supports of the frequent sequences. The database D is then updated by deleting a set of sequences Δ^- followed by inserting a set of sequences Δ^+. Let us denote the updated database D'. Note that $D' = (D - \Delta^-) \cup \Delta^+$. We denote the set of unchanged sequences by $D^- = D - \Delta^- = D' - \Delta^+$. Since the relative order of the sequences within a database does not affect the mining results, we may assume (without loss of generality) that all of the deleted sequences are located at the beginning of the database and all of the new sequences are appended at the end, as illustrated in Figure 1.

Our objective is to find all maximal frequent sequences in the database D' given Δ^-, D^-, Δ^+, and the result of mining D. Table 1 summarizes the notations we use in this paper.

Symbol	Description
s	a sequence
δ_X^s	support count of sequence s in database X
b_X^s	an upperbound of δ_X^s
ρ_s	support threshold
$\|X\|$	number of sequences in database X
L_i	the set of length-i frequent sequences
L	the set of all frequent sequences
D	old database
D'	updated database
Δ^-	the set of deleted sequences
Δ^+	the set of inserted sequences
D^-	the set of sequences shared by D and D'

Table 1: Notations

3 Related Works

Agrawal and Srikant [1] first studied the problem of mining frequent sequences. Among three algorithms they proposed, `AprioriAll` has the best overall performance. Later, the same authors proposed `GSP` [13]. Similar to the structure of the Apriori algorithm [11] for mining association rules, `GSP` starts by finding all frequent 1-sequences from the database. A set of candidate 2-sequences are then generated. The support counts of the candidate sequences are then counted by scanning the database once. Those frequent 2-sequences are then used to generate candidate 3-sequences, and so on. In general, `GSP` uses a function to generate candidate $(k+1)$-sequences given the set of all frequent k-sequences. The algorithm terminates when no more frequent sequences are discovered during a database scan. Figure 2 shows the `GSP` algorithm. (In the algorithm, the symbol I denotes the set of all items in the database.)

`GSP` is much faster than `AprioriAll`, and it can also be used to solve other generalized versions of the frequent-sequence mining problem. For example, a user can specify a sliding time window. Items that occur in transactions that are within the sliding time window could be considered as to occur in the same transaction. Also, the problem of mining multi-level frequent sequences is addressed. However, the number of database scans required by `GSP` is determined by the length of the longest frequent sequences in the database. If the database is huge and if it contains very long frequent sequences, the I/O cost of `GSP` is high.

An interesting I/O-efficient algorithm, SPADE, was proposed by Zaki [15]. The idea is to first transform a sequence database from a "horizontal" representation to a "vertical" representation. With the vertical representation, each item is associated with an *id-list*. An id-list of an item is a list of all (customer id, transaction id) pairs such that the corresponding transactions in the database contain the item. The SPADE algorithm works on the transformed database and requires only three database scans to discover frequent sequences. While SPADE is an efficient algorithm, it requires the availability of the transformed database. Such a transformation requires either a high I/O cost or a lot of memory.

Based on SPADE, ISM [9] was put forward as the first algorithm that addresses the maintenance problem of frequent sequences. ISM maintains a *sequence lattice* of an old database on which a previous mining exercise has been performed. The sequence lattice is composed of all frequent sequences (FS) and all the sequences in the *negative border* (NB). The support count of every sequence in the lattice is recorded. ISM considers the case when transactions

```
Function GGen(L_i)
  CandidateSet := ∅
  if (i=1)
    for each pair of s_1, s_2 ∈ L_i (say s_1 = ⟨{i_1}⟩, s_2 = ⟨{i_2}⟩)
      insert ⟨{i_1, i_2}⟩, ⟨{i_1}, {i_2}⟩, and ⟨{i_2}, {i_1}⟩ into CandidateSet
    Return CandidateSet
  for each pair of s_1, s_2 ∈ L_i
    let s be the sequence obtained by deleting the first item (say i_1) from s_1
    let s' be the sequence obtained by deleting the last item (say i_2)from s_2
    if (s = s'), insert ⟨i_1, s, i_2⟩[1] into CandidateSet
    let s be the sequence obtained by deleting the last item (say i_1) from s_1
    let s' be the sequence obtained by deleting the first item (say i_2)from s_2
    if (s = s'), insert ⟨i_2, s, i_1⟩ into CandidateSet
  for each sequence s ∈ CandidateSet
    if ∃s', such that (|s'| = i) ∧ (s' ⊑ s) ∧ (s' ∉ L_i), delete s from CandidateSet
  Return CandidateSet

Algorithm GSP(D, ρ_s, I)
  C_1 := {⟨{i}⟩ | i ∈ I}
  Scan D to compute δ^s_D for every sequence s in C_1
  L_1 := {s | s ∈ C_1, δ^s_D ≥ ρ_s × |D|}
  i := 1
  while (L_i ≠ ∅)
    C_{i+1} := GGen(L_i)
    Scan D to compute δ^s_D for every sequence s in C_{i+1}
    L_{i+1} := {s | s ∈ C_{i+1}, δ^s_D ≥ ρ_s × |D|}
    i := i + 1
  Return L_1 ∪ L_2 ∪ ... ∪ L_{i-1}
```

Figure 2: Algorithm GSP

are inserted to the database. With the help of the sequence lattice, ISM is able to determine the revised set of frequent sequences efficiently.

Our algorithms differ from ISM in the following aspects. First, we consider both sequence insertions and deletions, while ISM handles insertion only. Second, similar to SPADE, ISM also works on the "vertical" representation of the database. Our algorithms do not require the transformation.

In [16], another I/O-efficient algorithm, MFS was proposed. MFS achieves I/O efficiency by generalizing the candidate sequence generation function of GSP. Recall that during each iteration, GSP's candidate generation function generates only candidate sequences of the same length. MFS, on the other hand, allows candidate sequences of various lengths to be generated if given a set of frequent sequences of various lengths. This allows long sequences to be generated early, reducing the number of iterations and hence the I/O cost.

For MFS to be effective, it requires that an initial estimate (S_{est}) of the set of frequent sequences be available. In [16], the initial estimate is obtained by mining a small sample of the database. For the maintenance problem, if the old and the new databases share a significant number of sequences, we can use the frequent sequences of the old database as the estimate for mining the new database using MFS. Hence, MFS is potentially an efficient algorithm for the maintenance problem. Figure 3 shows the MFS algorithm.

[1] $\langle i_1, s, i_2 \rangle$ is the sequence obtained by adding item i_1 to the beginning of sequence s and adding item i_2 to the end of s. Whether i_1 is in a separate itemset in the result sequence is determined by whether i_1 is in a separate itemset in s_1; similarly whether i_2 is in a separate itemset in the result sequence is determined by whether i_2 is in a separate itemset in s_2.

```
Function MGen(MFSS, Iteration, AlreadyCounted)
  CandidateSet := ∅
  for each pair of s1, s2 ∈ MFSS such that |s1| > Iteration-2, |s2| > Iteration-2
  and that s1, s2 share at least one common subsequence of length ≥ Iteration-2
    for each common subsequence s of s1, s2 such that |s| ≥ Iteration-2
      NewCandidate := {⟨i1, s, i2⟩ | ⟨i1, s⟩ ⊑ s1, ⟨s, i2⟩ ⊑ s2}
      CandidateSet := CandidateSet ∪ NewCandidate
      NewCandidate := {⟨i2, s, i1⟩ | ⟨i2, s⟩ ⊑ s2, ⟨s, i1⟩ ⊑ s1}
      CandidateSet := CandidateSet ∪ NewCandidate
  for each sequence s ∈ CandidateSet
    if (s ⊢ MFSS) delete s from CandidateSet
    if s ∈ AlreadyCounted delete s from CandidateSet
    for any subsequence s' of s with length |s| − 1
      if (s' ⊬ MFSS) delete s from CandidateSet
  AlreadyCounted := AlreadyCounted ∪CandidateSet
  for each sequence s ∈ AlreadyCounted
    if (|s| = Iteration) delete s from AlreadyCounted
  Return CandidateSet

Algorithm MFS(D, ρs, I, Sest)
  MFSS := ∅
  CandidateSet := {⟨{i}⟩ | i ∈ I} ∪ {s | s ⊢ Sest, |s| > 1}
  Scan D to get δ^s_D for every sequence s in CandidateSet
  NewFrequentSequences := {s | s ∈ CandidateSet, δ^s_D ≥ ρs × |D|}
  AlreadyCounted := {s | s ⊢ Sest, |s| > 1}
  Iteration := 2
  while (NewFrequentSequences ≠ ∅)
    //Max(S) returns the set of all maximal sequences is S
    MFSS := Max(MFSS ∪NewFrequentSequences)
    CandidateSet := MGen(MFSS, Iteration, AlreadyCounted)
    Scan D to get δ^s_D for every sequence s in CandidateSet
    NewFrequentSequences := {s | s ∈ CandidateSet, δ^s_D ≥ ρs × |D|}
    Iteration := Iteration+1
  Return MFSS
```

Figure 3: Algorithm MFS

PrefixSpan [10] is a newly devised efficient algorithm for mining frequent sequences. It needs to generate a number of intermediate databases during the process. If main memory is large enough, PrefixSpan is very efficient; otherwise it will require a high cost.

Finally, a number of studies have been done on the problem of maintaining discovered association rules including [2, 3, 4, 5, 7, 8, 12, 14].

4 Algorithms

In this section we describe the two incremental update algorithms GSP+ and MFS+. The idea is that, given a sequence s, we use the support count of s in D (if available) to deduce whether s could have enough support in D'. In the deduction process, the portion of the database that has been changed, namely, Δ^- and Δ^+, might have to be scanned. If we deduce that s cannot be frequent in D', s's support in D^- (the portion of the database that has not been changed) is not counted. If D^- is large comparing with Δ^- and Δ^+, the pruning technique saves much CPU cost.

Before we present the algorithms, let us consider a few mathematical equations that allow

us to perform the pruning deductions. (As a reminder, readers are referred to Table 1 for the symbol definitions.)

First of all, since $D' = D - \Delta^- \cup \Delta^+ = D^- \cup \Delta^+$, we have, for any sequence s,

$$\delta^s_D = \delta^s_{D^-} + \delta^s_{\Delta^-}, \quad (1)$$

$$\delta^s_{D'} = \delta^s_{D^-} + \delta^s_{\Delta^+}, \quad (2)$$

$$\delta^s_{D'} = \delta^s_D + \delta^s_{\Delta^+} - \delta^s_{\Delta^-}. \quad (3)$$

Let us define $b^s_X = \min_{s'} \delta^{s'}_X$ for any sequence s and database X, where $(s' \sqsubseteq s) \wedge (|s'| = |s| - 1)$. That is to say, if s is a k-sequence, b^s_X is the smallest support count of the $(k-1)$-subsequences of s in the database X. Since the support count of a sequence s must not be larger than the support count of any subsequence of s, b^s_X is an *upper bound* of δ^s_X.

The reason for considering b^s_X is to allow us to estimate δ^s_X without counting it. As we will see later, under both `GSP+` and `MFS+`, a candidate sequence s is considered (and may have its support counted) only if all of s's subsequences are frequent. Since frequent sequences would already have their supports counted (in order to conclude that they are frequent), we would have the necessary information to deduce b^s_X when we consider s.

To illustrate how the bound is used in the deduction, let us consider the following simple Lemma:

Lemma 1 *If a sequence s is frequent in D', then $\delta^s_D + b^s_{\Delta^+} \geq \delta^s_D + b^s_{\Delta^+} - \delta^s_{\Delta^-} \geq |D'| \times \rho_s$.*

Proof: If s is frequent in D', we have,

$$\begin{aligned} |D'| \times \rho_s &\leq \delta^s_{D'} && \text{(by definition)} \\ &= \delta^s_D + \delta^s_{\Delta^+} - \delta^s_{\Delta^-} && \text{(by Equation 3)} \\ &\leq \delta^s_D + b^s_{\Delta^+} - \delta^s_{\Delta^-} \\ &\leq \delta^s_D + b^s_{\Delta^+}. \end{aligned}$$

Given a sequence s, if s is frequent in D, we know δ^s_D. If $b^s_{\Delta^+}$ is available, we can compute $\delta^s_D + b^s_{\Delta^+}$ and conclude that s is infrequent in D' if the quantity is less than $|D'| \times \rho_s$. Otherwise, we scan Δ^- to find $\delta^s_{\Delta^-}$. We conclude that s is infrequent in D' if $\delta^s_D + b^s_{\Delta^+} - \delta^s_{\Delta^-}$ is less than the required support count ($|D'| \times \rho_s$). Note that in the above cases, the deduction is made without processing D^- or Δ^+.

If a sequence s is not frequent in D, δ^s_D is unavailable. The pruning tricks derived from Lemma 1 are thus not applicable. However, being infrequent in D means that the support of s (in D) is *small*. The following Lemma allows us to prune those sequences.

Lemma 2 *If a sequence s is frequent in D' but not in D, then $b^s_{\Delta^+} \geq b^s_{\Delta^+} - \delta^s_{\Delta^-} \geq \delta^s_{\Delta^+} - \delta^s_{\Delta^-} > (|\Delta^+| - |\Delta^-|) \times \rho_s$.*

Proof: If s is frequent in D' but not in D, we have, by definition:

$$\begin{aligned} \delta^s_{D'} &\geq |D'| \times \rho_s = (|D^-| + |\Delta^+|) \times \rho_s, \\ \delta^s_D &< |D| \times \rho_s = (|D^-| + |\Delta^-|) \times \rho_s. \end{aligned}$$

Hence,

$$\begin{aligned} \delta^s_{D'} - \delta^s_D &> (|D^-| + |\Delta^+|) \times \rho_s - (|D^-| + |\Delta^-|) \times \rho_s \\ \delta^s_{\Delta^+} - \delta^s_{\Delta^-} &> (|\Delta^+| - |\Delta^-|) \times \rho_s \quad \text{(by Equation 3).} \end{aligned}$$

Also,

$$b^s_{\Delta^+} \geq b^s_{\Delta^+} - \delta^s_{\Delta^-} \geq \delta^s_{\Delta^+} - \delta^s_{\Delta^-}.$$

Lemma 2 thus follows.

Given a candidate sequence s that is not frequent in the old database D, we first compare $b^s_{\Delta^+}$ against $(|\Delta^+| - |\Delta^-|) \times \rho_s$. If $b^s_{\Delta^+}$ is not large enough, s cannot be frequent in D', and hence s can be pruned. Otherwise, we scan Δ^- to find $b^s_{\Delta^+} - \delta^s_{\Delta^-}$ and see if s can be pruned. If not, we scan Δ^+ and consider $\delta^s_{\Delta^+} - \delta^s_{\Delta^-}$.

Similar to Lemma 1, Lemma 2 allows us to prune some candidate sequences without completely counting their supports in the new database.

4.1 GSP+

Based on the Lemmas, we modify GSP to incorporate the pruning techniques mentioned. The new algorithm, GSP+, is shown in Figure 4.

GSP+ shares the same structure with GSP. GSP+ is an iterative algorithm. During each iteration, a set of candidate sequences C_i is generated based on L_{i-1} (the set of frequent sequences of length $i-1$). Before the database is scanned to count the support of the candidate sequences, the pruning tests derived from Lemmas 1 and 2 are applied. Depending on the test results, the datasets Δ^- and/or Δ^+ may have to be processed to count the support of a candidate sequence. GSP+ carefully controls when such countings are necessary. If all pruning tests fail on a candidate sequence s, GSP+ checks whether s is frequent in D. If so, δ^s_D is available. Hence, $\delta^s_{D'}$ can be computed by $\delta^s_D + \delta^s_{\Delta^+} - \delta^s_{\Delta^-}$. Finally, if s is not frequent in D, the unchanged part of the database, D^-, is scanned to find out the actual support of s. Since D^- is typically much larger than Δ^+ and Δ^-, saving is achieved by avoiding processing D^- for certain candidate sequences. As we will see later in Section 5, the pruning tests can prune up to 60% of the candidate sequences in our experiment setting. The tests are thus quite effective.

In terms of I/O performance, GSP+ generally has a slightly higher I/O cost than GSP. This is because GSP+ scans and processes Δ^-, which is not needed by GSP. However, in some cases, the pruning tests (in GSP+) remove all candidate sequences during an iteration of the algorithm. Under such cases, GSP+ saves some database passes. This makes GSP+ slightly more efficient than GSP in I/O.

4.2 MFS+

The pruning tests can also be applied to MFS. We call the resulting algorithm MFS+. The interesting thing about MFS+ is that it uses the set of frequent sequences (L_{old}) of the old database D as an initial estimate of the set of frequent sequences of the new database D'. These sequences together with all possible 1-sequences are put into a candidate set *CandidateSet*. It then scans Δ^-, D^-, and Δ^+ to obtain $\delta^s_{\Delta^-}$, $\delta^s_{D^-}$, and $\delta^s_{\Delta^+}$ for each sequence s in *CandidateSet*. From these counts, we can deduce which sequences in *CandidateSet* are frequent in D'. The *maximals* of such frequent sequences are put into the set *MFSS*. We can consider the set *MFSS* as the set of maximal frequent sequences that MFS+ knows given the information that MFS+ has obtained so far. MFS+ then executes a loop, trying to refine *MFSS*. During each iteration, MFS+

```
Algorithm GSP+(Δ⁻, D⁻, Δ⁺, ρ_s, I)
  C_1 := {⟨{i}⟩ | i ∈ I}
  Scan Δ⁻, D⁻, Δ⁺ to get δ^s_{Δ⁻}, δ^s_{D⁻}, δ^s_{Δ⁺} for each s ∈ C_1
  L_1 := {s | s ∈ C_1, δ^s_{D⁻} + δ^s_{Δ⁺} ≥ |D'| × ρ_s}
  i := 2
  C_i := GGen(L_{i-1})
  while (C_i ≠ ∅)
    calculate b^s_{Δ⁺} for each s ∈ C_i
    divide C_i into two sets candp and candq
    (candidates that are frequent in D are put into candp; other candidates are put into candq)
    for each sequence s ∈ candp, if δ^s_D + b^s_{Δ⁺} < |D'| × ρ_s, delete s from candp
    for each sequence s ∈ candq, if b^s_{Δ⁺} ≤ (|Δ⁺| − |Δ⁻|) × ρ_s, delete s from candq
    scan Δ⁻ to count δ^s_{Δ⁻} for each s in candp and candq
    for each sequence s ∈ candp, if δ^s_D + b^s_{Δ⁺} − δ^s_{Δ⁻} < |D'| × ρ_s, delete s from candp
    for each sequence s ∈ candq, if b^s_{Δ⁺} − δ^s_{Δ⁻} ≤ (|Δ⁺| − |Δ⁻|) × ρ_s, delete s from candq
    scan Δ⁺ to count δ^s_{Δ⁺} for each s in candp and candq
    for each sequence s ∈ candq, if δ^s_{Δ⁺} − δ^s_{Δ⁻} ≤ (|Δ⁺| − |Δ⁻|) × ρ_s, delete s from candq
    scan D⁻ to count δ^s_{D⁻} for each s in candq
    L_i = ∅
    for each sequence s ∈ candp, if δ^s_D + δ^s_{Δ⁺} − δ^s_{Δ⁻} ≥ |D'| × ρ_s, insert s into L_i
    for each sequence s ∈ candq, if δ^s_{D⁻} + δ^s_{Δ⁺} ≥ |D'| × ρ_s, insert s into L_i
    if (|L_i| > i), C_{i+1} := GGen(L_i)
    i := i + 1
  Return L_1 ∪ L_2 ∪ ... ∪ L_{i-1}
```

Figure 4: Algorithm GSP+

```
Algorithm MFS+(Δ⁻, D⁻, Δ⁺, ρ_s, I, L_old)
  MFSS := ∅
  CandidateSet := {⟨{i}⟩ | i ∈ I} ∪ {s | s ⊢ L_old, |s| > 1}
  Scan Δ⁻, D⁻, Δ⁺ to get δ^s_{Δ⁻}, δ^s_{D⁻}, δ^s_{Δ⁺} for each s ∈ CandidateSet
  NewFrequentSequences := {s | s ∈ CandidateSet, δ^s_{D⁻} + δ^s_{Δ⁺} ≥ |D'| × ρ_s}
  AlreadyCounted := {s | s ⊢ L_old, |s| > 1}
  Iteration := 2
  while (NewFrequentSequences ≠ ∅)
    //Max(S) returns the set of all maximal sequences is S
    MFSS := Max(MFSS ∪ NewFrequentSequences)
    CandidateSet := MGen(MFSS, Iteration, AlreadyCounted)
    calculate b^s_{Δ⁺} for each s ∈ CandidateSet
    for each sequence s ∈ CandidateSet
      if b^s_{Δ⁺} ≤ (|Δ⁺| − |Δ⁻|) × ρ_s, delete s from CandidateSet
    scan Δ⁻ to count δ^s_{Δ⁻} for each s ∈ CandidateSet
    for each sequence s ∈ CandidateSet
      if b^s_{Δ⁺} − δ^s_{Δ⁻} ≤ (|Δ⁺| − |Δ⁻|) × ρ_s, delete s from CandidateSet
    scan Δ⁺ to count δ^s_{Δ⁺} for each s ∈ CandidateSet
    for each sequence s ∈ CandidateSet
      if δ^s_{Δ⁺} − δ^s_{Δ⁻} ≤ (|Δ⁺| − |Δ⁻|) × ρ_s, delete s from CandidateSet
    scan D⁻ to count δ^s_{D⁻} for each s ∈ CandidateSet
    NewFrequentSequences := {s | s ∈ CandidateSet, δ^s_{D⁻} + δ^s_{Δ⁺} ≥ |D'| × ρ_s}
    Iteration := Iteration+1
  Return MFSS
```

Figure 5: Algorithm MFS+

Parameter	Description	value
$\|D\|$	Number of customer sequences (size of database)	1,500,000
$\|C\|$	Average number of transactions per customer	10
$\|T\|$	Average number of items per transaction	2.5
$\|S\|$	Average length of maximal potentially large sequences	4
$\|I\|$	Average size of Itemsets in maximal potentially large sequences	1.25
N_S	Number of maximal potentially large sequences	5,000
N_I	Number of maximal potentially large itemsets	25,000
N	Number of items	10,000

Table 2: Parameters and values for data generation

generates a set of candidate sequences *CandidateSet* from *MFSS*. `MFS+` then deduces which candidate sequences must not be frequent by applying the pruning tests. In the process, the datasets Δ^- and Δ^+ may have to be scanned. For those sequences that are not pruned by the tests, D^- is scanned to obtain their exact support counts. Since sequences that are originally frequent in the old database D have already had their supports (w.r.t D') counted in the initial part of `MFS+`, all candidate sequences considered by `MFS+` in the loop section are infrequent in D. Hence, only those pruning tests resulting from Lemma 2 are used. `MFS+` terminates when no refinement is made to *MFSS* during an iteration. Figure 5 shows the algorithm `MFS+`.

5 Results

We performed a number of experiments comparing the performance of `GSP` and `MFS` with that of the incremental algorithms. In this section we present some representative results from our experiments.

5.1 Synthetic Dataset

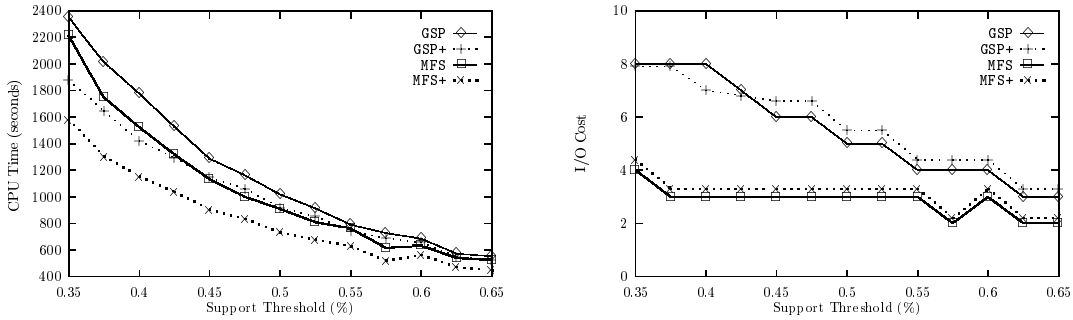
We used synthetic data as the test databases. The generator is obtained from the IBM Quest data mining project. Readers are referred to [6] for the details of the data generator. The values of the parameters we used in the data generation are listed in Table 2.

In our baseline experiment, we use a test database (D) of 1,500,000 customer sequences. 10% (150,000) of the sequences are removed. These sequences form the dataset Δ^-. Another 10% (150,000) sequences are generated to form the set Δ^+. These sequences are added to the database to form D'. We execute a sequence mining program on D to obtain the supports of all the frequent sequences in D. These counts are used by the incremental algorithms.

Four algorithms `GSP`, `MFS`, `GSP+`, and `MFS+` are executed to mine the updated database D' using different values of support threshold ($0.35\% \leq \rho_s \leq 0.65\%$). The experiments were performed on a 700MHz PIII Xeon machine with 4GB main memory running Solaris 8.

We compare the algorithms in terms of CPU cost and I/O cost. CPU costs are measured by recording the CPU time each algorithm takes. I/O cost is measured in terms of database scans normalized by the size of D'. For example, if `GSP` scans D' 8 times, then the I/O cost of `GSP` is 8. For an incremental algorithm, if it reads Δ^- n_1 times, D^- n_2 times, and Δ^+ n_3 times, then the I/O cost is $(n_1|\Delta^-| + n_2|D^-| + n_3|\Delta^+|)/|D'|$. We notice that while the I/O

C10T2.5S4I1.25 D1500-150+150(thousand)



(a) CPU costs of four algorithms under different support thresholds

(b) I/O costs of four algorithms under different support thresholds

Figure 6: Comparison of four algorithms under different support thresholds

costs of GSP and MFS are integral numbers (because D' is the only dataset they read), those of the incremental algorithms could be fractional. We think showing I/O cost in number of database scans instead of real time (such as seconds) is more reasonable, since different systems have different I/O speeds and different data caching policies.

5.2 Performance

Figure 6(a) shows the CPU costs of the algorithms as the support threshold ρ_s varies from 0.35% to 0.65%. In this graph (as well as other performance graphs), we use solid lines for GSP and MFS, and dotted lines for the incremental algorithms.

From the figure, we see that as the support threshold increases, the CPU costs of all four algorithms decrease. This is because a larger ρ_s means fewer and shorter frequent sequences, and thus fewer candidate sequences whose supports need to be counted.

Among the four algorithms, GSP has the highest CPU cost. Using the set of frequent sequences in the old database as an initial estimate allows MFS to disover those long frequent sequences early. As we will see shortly, this results in fewer iterations for MFS. During each iteration, each sequence s in the database is matched against a set of candidate sequences to see which of the candidate sequences are contained in s. The support counts of those candidate sequences contained in s will be incremented. Since MFS performs fewer database scans compared with GSP, the database sequences are matched against candidate sequences fewer times. The overall effect is a lower CPU cost for MFS compared with GSP.

From the figure, we also see that GSP and MFS consume more CPU time than their incremental counterparts — GSP+ is more efficient than GSP, while MFS+ is more efficient than MFS. The saving in CPU cost is obtained by the pruning effect of the incremental algorithms. As we have mentioned in the previous section, some candidate sequences are pruned by processing only Δ^- and/or Δ^+; the set D^- is avoided. In our baseline experiment setting, the size of D^- is 9 times that of Δ^+ and Δ^-. ($|\Delta^-| = |\Delta^+| = 10\%$ of $|D|$; $|D^-| = 90\%$ of $|D|$.) Avoid counting the supports of the pruned candidate sequences in D^- results in a significant CPU cost reduction.

ρ_s	0.35%	0.4%	0.45%	0.5%	0.55%	0.6%	0.65%
Total # of candidates	34,065	18,356	10,024	5,812	3,365	2,053	1,160
Those requiring $\delta^s_{D^-}$	13,042	7,161	3,966	2,313	1,353	867	509
Percentage (row3/row2)	38%	39%	40%	40%	40%	42%	44%

Table 3: Effectiveness of prunning tests

As an illustration of the effectiveness of the pruning tests, if we set $\rho_s = 0.35\%$ in our experiment, there are about 34,000 candidate sequences processed by `GSP` that are of length 2 or above. (Note that length-1 candidate sequences are not subjected to pruning.) Our experiment result shows that `GSP+` needs to process D^- to obtain the support counts of only about 38% of those candidate sequences. This accounts for the saving in CPU cost achieved by `GSP+` over `GSP`. Similarly, `MFS+` outperforms `MFS`, mainly due to candidate pruning. Table 3 shows the number of candidate sequences processed by `GSP+` (second row) and the number of them that require the scanning of D^- (third row).

Finally, when the support threshold becomes large, the CPU times of the algorithms are roughly the same. This is because a large support threshold implies short and few frequent sequences. (For example, when $\rho_s = 0.65\%$, the longest frequent sequence in D' is only 3.) In such a case, `GSP` and `MFS` take similar number of iterations, and hence the CPU saving of `MFS` over `GSP` is diminished. Moreover, there are much fewer candidate sequences when ρ_s is large, and there are much fewer opportunities for the incremental algorithms to prune candidate sequences. This observation is illustrated in Table 3.

Figure 6(b) shows the I/O costs of the four algorithms. Note that the I/O costs take on discrete values. The lines connecting points of the same kind are there for legibility reason only and should not be interpreted as interpolation.

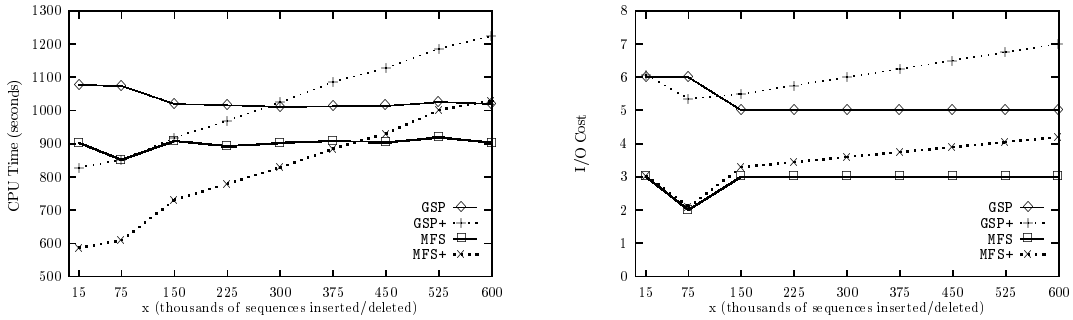
From the figure, we see that as ρ_s increases, the I/O costs of the algorithms decrease. This is again due to the fact that a larger ρ_s leads to shorter frequent sequences. Hence, the number of iterations (and database scans) needed is small.

Comparing `GSP` and `MFS`, we see that `MFS` is a very I/O-efficient algorithm. For example, when $\rho_s = 0.4\%$, `GSP` requires 8 passes over D' while `MFS` needs only 3. Using the frequent sequences in D as an estimate of those in D' gives `MFS` a head start. This allows `MFS` to discover all the maximal frequent sequences in much fewer database scans.

The incremental algorithms generally require a slightly higher I/O cost than their non-incremental counterparts. The reason is that the incremental algorithms scan the dataset Δ^- in addition to D' to make pruning deductions. This additional cost is shown in Figure 6(b). Exceptions occur for `GSP+` when ρ_s is less than 0.45%. Our experiment data shows that in those particular cases, the pruning tests eliminated all candidate sequences in some iterations. This results in a lower I/O cost.

From Figure 6 we can conclude that `GSP` has a high CPU cost and a high I/O cost. `GSP+` reduces the CPU requirement but does not help in terms of I/O. `MFS` is very I/O-efficient and it also performs better than `GSP` in terms of CPU cost. `MFS+` is the overall winner. It requires the least amount of CPU time and its I/O cost is comparable to that of `MFS`.

C10T2.5S4I1.25 D1500-x+x(thousand) ρ_s=0.5%



(a) CPU costs of four algorithms under different insertion and deletion sizes

(b) I/O costs of four algorithms under different insertion and deletion sizes

Figure 7: Comparison of four algorithms under different insertion and deletion sizes

5.3 Varying $|\Delta^+|$ and $|\Delta^-|$

In our previous experiment, we fixed the size of the datasets Δ^+ and Δ^- to 10% of D. As we have discussed, the incremental algorithms achieve efficiency by processing Δ^+ and Δ^- and use the result to prune candidate sequences. The performance of the incremental algorithms is thus dependent upon the sizes of Δ^+ and Δ^-. we ran an experiment to study how $|\Delta^+|$ and $|\Delta^-|$ affect the performance of the algorithms. In the experiment, we set $|D| = |D'| = 1,500,000$ sequences and $\rho_s = 0.5\%$. We varied $|\Delta^+|$ and $|\Delta^-|$ from 15,000 to 600,000 (i.e., from 1% to 40% of $|D|$). Figures 7(a) and 7(b) show the CPU costs and the I/O costs of the four algorithms, respectively.

From Figure 7(a), we see that the CPU costs of `GSP` and `MFS` stay relatively steady. This is expected since the size of the database D' does not change. The CPU costs of the incremental algorithms, on the other hand, increase linearly with the size of Δ^+ and Δ^-. The increase in CPU costs is due to a longer processing time taken by `GSP+` and `MFS+` to find the quantities $\delta^s_{\Delta^-}$ and $\delta^s_{\Delta^+}$, which are needed to make pruning deductions. From the figure, we see that `MFS+` outperforms others even when the database is changed substantially. In particular, if $|\Delta^+|$ and $|\Delta^-|$ are less than 375,000 sequences (or 25% of $|D|$), `MFS+` is the most CPU-efficient algorithm in our experiment. For cases in which only a small fraction of the database is changed, the incremental algorithms can achieve significant performance gains.

As we have discussed previously, the incremental algorithms usually have slightly higher I/O costs (compared with their non-incremental counterparts) since they have to read Δ^-. The additional I/O costs thus increase with $|\Delta^-|$. The effect is shown in Figure 7(b) in which the gap between `GSP` and `GSP+` (and also that between `MFS` and `MFS+`) widens as $|\Delta^-|$ increases.

We have conducted quite a number of sensitivity studies, varying various parameters of the experiment. The results obtained from such studies confirm the general observations we presented in this section. Due to space limitation, we omit those results in this paper.

6 Conclusions

In this paper we studied the maintenance problem of frequent sequences. We proposed two incremental algorithms, namely, `GSP+` and `MFS+`. We analyzed their performance in terms of CPU efficiency and I/O efficiency, and compared them with `GSP` and `MFS`. Our experiment results show that the incremental algorithms outperform their non-incremental counterparts in CPU cost at the expense of a small penalty in I/O cost, or no extra expense at all in some rare cases. The performance gain is obtained by pruning candidate sequences to avoid processing the bulk of the database.

The `MFS`-based algorithms perform better than the `GSP`-based ones, particularly in I/O-efficiency. The `MFS`-based algorithms use the set of frequent sequences of the old database as an estimate of those in the new database. This allows the algorithms to obtain a set of frequent sequences that is close to the final complete set in one database scan. After that, a few more database scans are sufficient to refine the set of frequent sequences to the complete one.

Finally, we observed that the performance gains of the incremental algorithms are the most prominent when the database contains long and numerous frequent sequences. This is because, in those cases, there are many candidate sequences for the algorithms to consider, and the impact of the pruning technique is the most significant.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[2] Necip Fazil Ayan, Abdullah Uz Tansel, and Erol Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proc. 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA USA, August 1999.

[3] D. W. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating techniques. In *Proc. 12th IEEE International Conference on Data Engineering (ICDE)*, New Orleans, Louisiana, U.S.A, March 1996.

[4] D. W. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. International Conference On Database Systems For Advanced Applications (DASFAA)*, Melbourne, Australia, April 1997.

[5] D. W. Cheung, V. Ng, and B. W. Tam. Maintenance of discovered knowledge: A case in multi-level association rules. In *Proc. Second International Conference on Knowledge Discovery and Data Mining (KDD)*, Portland, Oregon, August 1996.

[6] http://www.almaden.ibm.com/cs/quest/.

[7] S. D. Lee and D. W. Cheung. Maintenance of discovered association rules: When to update? In *Proc. 1997 ACM-SIGMOD Workshop on Data Mining and Knowledge Discovery (DMKD)*, Tucson, Arizona, May 1997.

[8] E. Omiecinski and A. Savasere. Efficient mining of association rules in large dynamic databases. In *Proc. BNCOD'98*, 1998.

[9] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proceedings of the 1999 ACM 8th International Conference on Information and Knowledge Management (CIKM'99)*, Kansas City, MO USA, November 1999.

[10] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proc. 17th IEEE International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001.

[11] T. Imielinski R. Agrawal and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, page 207, Washington, D.C., May 1993.

[12] N. L. Sarda and N. V. Srinivas. An adaptive algorithm for incremental mining of association rules. In *Proc. DEXA Workshop'98*, 1998.

[13] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Conference on Extending Database Technology (EDBT)*, Avignion, France, March 1996.

[14] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. KDD'97*, 1997.

[15] Mohammed J. Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management(CIKM'98)*, Washington, United States, November 1998.

[16] Minghua Zhang, Ben Kao, C.L. Yip, and David Cheung. A GSP-based efficient algorithm for mining frequent sequences. In *Proc. of IC-AI'2001*, Las Vegas, Nevada, USA, June 2001.