Incremental Update on Sequential Patterns in Large Databases

Ming-Yen Lin and Suh-Yin Lee
Institute of Computer Science and Information Engineering
National Chiao-Tung University
Email: ericl@info4.csie.nctu.edu.tw, sylee@csie.nctu.edu.tw

Abstract

Mining of sequential patterns in a transactional database is time-consuming due to its complexity. Maintaining present patterns is a non-trivial task after database update, since appended data sequences may invalidate old patterns and create new ones. In contrast to re-mining, the key to improve mining performance in the proposed incremental update algorithm is to effectively utilize the discovered knowledge. By counting over appended data sequences instead of the entire updated database in most cases, fast filtering of patterns found in last mining and successive reductions in candidate sequences together make efficient update on sequential patterns possible.

1. Introduction

As computerized applications facilitate automatic collection of large amount of data into databases, new challenges to extract practical knowledge from large databases arise. Data Mining and Knowledge Discovery in Databases (KDD) thus has been recognized as a promising field of AI, statistics and database researches [1].

Mining of sequential patterns was first introduced in [2]. The purpose is to discover sequential patterns in a database of customer transactions, consisting of records having customer id, transaction time, and transaction items. A sequential pattern indicates a sequence of transactions that usually occurred serially in time. For example, in a transactional database, one might purchase a PC and then purchase a printer at some later time. After a period of time, he/she could possibly buy some printing software and a scanner. If there exists a sufficient number of customers in the database who have a purchasing sequence of PC, printer, printing software and scanner, then such a sequence is a sequential pattern. Note that items in a sequential pattern need not be simple items, and that transactions need not be consecutive. As pointed out in [11], this problem was motivated by retailing applications, but the results can be applied to many business and scientific domains.

In general, the process for sequential pattern mining is comprised of several passes. Before mining process starts, the user specifies the minimum support, where the support is the percentage of data sequences that contain the pattern. Note that the support in association rules is transaction-based, while in sequential patterns, the support is based on data sequences. The counting of support is different. Suppose that a customer has two transactions that bought the same item. In the discovery of association rules, the customer contributes to the support count of that item by two, whereas it is counted only once for the support count in sequential pattern mining [2].

The transaction database is sorted by customer id into customer sequences, or data sequences. The mining process makes several passes over these data sequences. Various candidate sequences, which are potential sequential patterns, are generated in individual passes. In each pass, all data sequences, or trimmed data sequences, are scanned to increment the supports of candidate sequences. Candidate sequences having minimum support become frequent sequences. They are then used for the creation of longer candidate sequences for next pass. The algorithm stops when there is no candidate sequence any more. The performance of mining depends heavily on the number of candidate sequences and the number of data sequences in a pass.

The issue of maintaining sequential patterns becomes essential because database transactions may be updated over time. Due to new transactions, some existing sequential patterns would become invalid after database update since they might no longer have sufficient supports, while some new sequential patterns might appear. However, there is not much work on incremental updating of sequential patterns. In order to ascertain sequential patterns up to date for the updated database, re-execution of mining algorithm on the updated database is required. Nevertheless, because of appended database transactions, re-execution of mining algorithm demands more time than previous mining. Moreover, the effort of mining last time is wasted if all discovered sequential patterns in the original database were ignored.

There are many excellent algorithms that deal with the mining of association rules [1, 4, 10, 12] and several algorithms were developed for the mining of sequential patterns [2, 11, 13]. Algorithms to discover frequent episodes in a single long sequence and its generalization can be found in [8, 9]. There is no efficient algorithm designed for the maintenance of sequential patterns in large databases. As mentioned earlier, they call for the need of re-mining the whole database. Incremental updating techniques for the maintenance of association rules were proposed in [5, 7, 12]. However, previous work dealing with the incremental updating of sequential patterns cannot be found. Besides, appended transactions induce more complicated problems in sequential pattern mining than in association rule mining. The problem of finding association rules concerns with intra-transaction patterns whereas that of sequential pattern mining concerns with inter-transaction patterns [2]. Appended transactions bear no relation to original database for the former problem, while for the latter problem, different transactions with same customer id in both databases must be sorted into one data sequence.

In this paper, we propose a new algorithm that can utilize information about discovered sequential patterns, and efficiently reduce candidate sequences for mining in updated database. The objective of this work is to solve the update problem of sequential patterns after a nontrivial number of new transactions have been appended to original database. Assuming that minimum support keeps the same, existing frequent sequences and their supports in original database could be utilized for the mining of updated database. Through effective reuse of previous derived knowledge in each pass, the number of candidate sequences is substantially reduced. Instead of counting all candidate sequences for full updated database, counting reduced candidate sequences for original database and for increment database, could achieve better performance. For transactions that have same customer id in both original database and increment database, we extract old transactions from original database and merge into increment database. Since the size of increment database is smaller than original database in general, better performance thus could be retained.

The remaining paper is organized as follows. Section 2 describes the problems of sequential pattern mining and incremental update. In Section 3, we propose our new algorithm for the maintenance of sequential patterns. Section 4 concludes our study.

2. Problem Formulation

2.1 Sequential pattern mining

Let database D be a set of transactions, where each transaction T consists of transaction-time (*tid*), customer-id (*cid*), and a set of items. The quantities of items are not considered here. For simplicity, we assume that no customer has more than one transaction at same transaction time.

An itemset I, denoted by $(x_1, x_2, \dots, x_m)$, is a nonempty set of items, where each item x_k is represented by an integer. A sequence s, denoted by $<a_1a_2\dots a_n>$, is an ordered set of itemsets. Each a_j is an itemset, which is called an element of the sequence. An item can occur only once in an element of a sequence, but can occur multiple times in different elements of a sequence. A k-sequence consists of k items. For example, $<(8)(2)(1)>$ and $<(3)(5,9)>$ are both 3-sequences.

A sequence $<a_1a_2\dots a_n>$ is a subsequence of another sequence $<b_1b_2\dots b_m>$ if there exist integers $i_1<i_2< \dots <i_n$ such that $a_1 \subseteq b_{i1}$, $a_2 \subseteq b_{i2}, \dots, a_n \subseteq b_{in}$. For example, $<(2)(5,9)(8)>$ is a subsequence of $<(6)(2)(1)(5,9)(3,8)>$ since $(2) \subseteq (2)$, $(5, 9) \subseteq (5, 9)$ and $(8) \subseteq (3, 8)$. However, $<(3)(8)>$ is not a subsequence of $<(3,8)>$, and vice versa.

All transactions from the same customer in the database can be grouped together, and then sorted by transaction time in increasing order into a customer sequence, or called data sequence. The support for a sequence s is defined as the fraction of total data sequences that contain s. A data sequence contains a sequence s if s is a subsequence of the data sequence. A sequence is a frequent sequence if its support is greater than the user-specified minimum support, denoted *min_sup*.

Given a database of customer transactions, the problem of sequential pattern mining is to find all frequent sequences. Concerning association rules, the discovery of sequential patterns can be thought of as association discovery over a temporal database.

2.2 Incremental update on discovered sequential patterns

Let $|DB|$ be the number of data sequences in the **original database** *DB* and *min_sup* be the minimum support. After some update of the database, a few transactions are appended to *DB*. These transactions can be sorted by *cid* into $|db|$ data sequences in the **increment database** *db*. *UD* is the **updated database** combining all data sequences from *DB* and *db*, $UD = DB \cup db$. Let there be $|b|$ customers appearing both in *DB* and *db*. If all customers in *db* are new customers with respect to *DB*, that is, all *cids* in *db* are different from those in *DB*, $|b|$ is zero. Let S_k^{DB} be **the set of all frequent *k*-sequences** in *DB*, S_k^{UD} be the set of all frequent *k*-sequences in *UD*, and the set of all sequential patterns in *DB* and *UD* be S^{DB} and S^{UD} respectively.

Assume that for each sequential pattern *s* in *DB*, its support count, denoted by S_{count}^{DB}, is available. With respect to the same minimum support *min_sup*, a sequence *s* is a frequent sequence in the updated database *UD*, if its support count S_{count}^{UD} is greater than $min\_sup \times$ (total data sequences in *UD*). That is, S_{count}^{UD} is no less than $min\_sup \times (|DB|+|db|-|b|)$. X_{count}^{DB}, X_{count}^{db} and X_{count}^{UD} are support counts of a sequence *X* in *DB*, *db* and *UD*. If there is no same *cid* in *DB* and *db*, $X_{count}^{UD} = X_{count}^{DB} + X_{count}^{db}$. A sequential pattern *s* in S^{DB} might not be in S^{UD} because of database update. On the other hand, a sequence *s'* that is not in S^{DB} might turn out to be in S^{UD}.

Consequently, the problem of incremental update on sequential patterns is to find the new set S^{UD} of frequent sequences in *UD*. As shown in Fig. 1, In order to find S^{UD}, previous approaches take all data sequences to compute sequential patterns and their supports. With discovered S^{DB} and support counts, the incremental algorithm updates supports of sequential patterns in S^{UD} by scanning data sequences in *db* only, if they were frequent in *DB*. New candidate sequences, which might not exist in *DB*, are generated in the same scan. Some new candidate sequences are pruned, if they do not have sufficient supports relative to *db*, before the scan of *DB* starts. The complexity of mining process is reduced due to the employment of previous knowledge and the increment database.

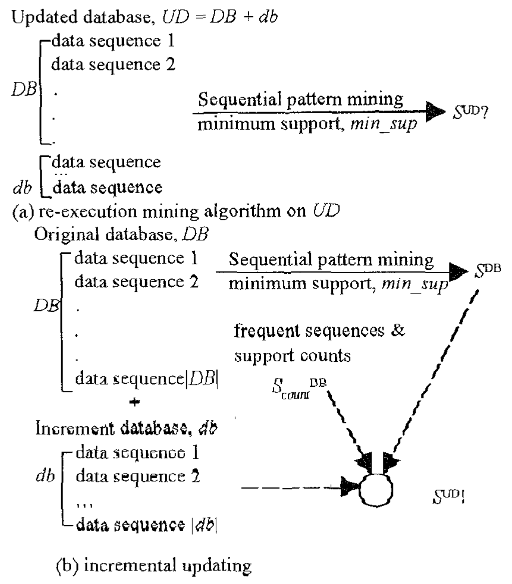

Figure 1: Incremental update versus re-mining

2.3 Example of sequential patterns after database updated

Consider a database with transactions sorted by customer id into 6 data sequences as shown in Fig. 2. Assume minimum support is set to 33%, i.e., minimum support count of data sequences being 2. The sequential patterns are <(1)>, <(2)>, <(3)>, <(4)>, <(5)>, <(8)>, <(1,2)>, <(8)(2)>, <(8)(1)>, <(2)(1)>, <(2)(8)>, <(5)(1)>, <(5)(3)>, and <(3)(1)>.

Customer Id	Data Sequence
1	<(8)(2)(1)>
2	<(1,2)(8)(7)>
3	<(1,4)>
4	<(1)(4)>
5	<(5)(3)(1)>
6	<(2,5)(8)(3)(1,2)>

Figure 2: The original database *DB* with 6 data sequences

After some update activities, assume the transactions appended to the database all come from new customers. The increment database, sorted by *cid* into data sequences, is shown in Fig. 3. With same minimum support, it requires 3 data sequences to be a frequent sequence now. Previous sequential patterns <(3)>, <(8)(2)>, <(2)(1)>, <(2)(8)>, <(5)(1)>, <(5)(3)>, and <(3)(1)> are no longer frequent due to this update. While <(7)> and <(8)(7)> become new sequential patterns because they have minimum supports now.

Customer Id	Data Sequence
7	<(5,7,8)(7)>
8	<(8)(1)(7)>
9	<(1,2)(4)>

Figure 3: Data sequences in the increment database *db*

In cases of update when the customer id of new transactions appears in the original database, these transactions must be appended to the same customer to form his/her data sequence. For example, assume the customers, whose *cids* are 1 and 4, bought item 7 at later time and the data sequences for *cid*=1 and *cid*=4 now are <(8)(2)(1)(7)> and <(1)(4)(7)> respectively. Fig. 4 shows an example of data sequences, which come from both old and new customers, in the increment database. The data sequences for mining in updated database *UD* is shown in Fig. 5.

After invalidating sequences <(5)> and <(8)(1)>, the resulting sequential patterns in this database are <(1)>, <(2)>, <(4)>, <(7)>, <(8)>, <(1,2)> and <(8)(7)>, for the given minimum support 33%.

Customer Id	Data Sequence
1	<(7)>
4	<(7)>
10	<(4)>
11	<(1,2)>
12	<(8)(7)>

Figure 4: Data sequences in *db* with *cids* occurring in *DB*

Customer Id	Data Sequence
1	<(8)(2)(1)(7)>
2	<(1,2)(8)(7)>
3	<(1,4)>
4	<(1)(4)(7)>
5	<(5)(3)(1)>
6	<(2,5)(8)(3)(1,2)>
7	<(5,7,8)(7)>
8	<(8)(1)(7)>
9	<(1,2)(4)>
10	<(4)>
11	<(1,2)>
12	<(8)(7)>

Figure 5: Merged data sequences in *UD*

3. Fast Sequential Pattern Update Algorithm (FASTUP)

3.1 Previous algorithms versus FASTUP algorithm

Apriori algorithm was designed to discover association rules [1], while *AprioriAll* algorithm introduced in [2] was the first algorithm that deals with the problem of sequential patterns mining. In subsequent work [11], the same authors proposed *GSP* (Generalized Sequential Pattern) algorithm that outperforms *AprioriAll*. An algorithm called *SPADE* (Sequential PAttern Discovery using Equivalence classes) was designed that uses simple join operations to find sequential patterns [13]. However, vertical database layout was used, instead of horizontal database layout used in earlier approaches [2, 11]. Therefore, the *GSP* algorithm is briefly reviewed here.

GSP algorithm makes multiple passes over the database. In the first pass, frequent 1-sequences with their supports are determined by counting the support of 1-itemsets. In the subsequent passes, candidate *k*-sequences are generated from frequent (*k-1*)-sequences obtained in pass-(*k-1*). Then the supports for these candidates are computed and those with minimum support become frequent sequences. This process is iterated until no more candidate sequences are formed. In each pass, every data sequence is checked to increment the support count of candidates contained in this data sequence. Hence, there are two essential sub-processes in this algorithm:

1. **Candidate generation**: Let L_k denote the set of all frequent *k*-sequences, and C_k denote the set of candidate *k*-sequences. Given L_{k-1}, C_k is generated by selectively join L_{k-1} with L_{k-1} itself. If there exist any (*k-1*)-subsequence of a candidate which is not in L_{k-1}, the candidate is pruned from C_k. Candidates are inserted into a hash tree to enable fast counting. Please refer to [11] for the detailed join operation and the hash tree mechanism.
2. **Support counting**: Every item and its following items in the checking data sequence is hashed to reach leaf buckets in the candidate hash tree. The support of each candidate in the buckets, if the candidate is contained in the data sequence, is incremented.

Note that as the name *GSP* suggests, it not only solves problems formulated in previous section, but it also solves problems generalized with constraints on pattern hierarchy and transaction-time [11]. The algorithm described here is to solve the fundamental problem of sequential patterns without constraints.

The basic construct of our algorithm *FASTUP* is similar to that of *GSP*, with improvements on candidate generation and support counting. *FASTUP* algorithm has several arguments similar to the incremental association update algorithm *FUP* [5]. Nevertheless, the effect of candidate reduction is more dramatic in sequence mining. Moreover, we have to consider whether appended data sequences are required to merge with old data sequences. Features that distinguish *FASTUP* from *GSP* are listed as follows.

1. During pass-*k*, for each $X \in S_k^{DB}$, The support count of X is updated against *db*, without re-scan *DB*. Old frequent sequences that do not satisfy new support count are filtered out.
2. During pass-*k*, if $X \in S_{k-1}^{DB}$ and $X \notin S_{k-1}^{UD}$, prune every candidate $C \in C_k$ such that X is a subsequence of C. With this feature *FASTUP* has greater capability in candidate reduction by a simple check in *db* before counting starts in *DB*.
3. During pass-*k*, a *k*-sequence X, where $X \notin S_k^{DB}$, with X_{count}^{db}, is added to C_k if $X_{count}^{db} \geq min\_sup \times |db|$. For each $X \in C_k$, The support count of X is updated against *DB*. The generation of candidate sequence X requires less data sequences checking in *db* than in *DB* or in *UD*. Only the supports of "new" candidates, instead of all candidates, need to be checked in *DB*.

These features altogether enable *FASTUP* to have better performance than re-execution *GSP* on updated database. Previous frequent sequences and candidate sequences in *DB* that do not belong to *UD* are pruned away by simple checks on *db*. Without missing any potential new frequent sequences in *UD*, fewer new candidates are generated from *db* for checking against *DB*.

3.2 Merge data sequences of the same customer for sequential pattern mining

For the given data sequences in *DB* and *db*, the sequential patterns in *UD* can be found by re-execution of *GSP*. Transactions that come from the same customer, either in *DB* or in *db*, are parts of the unique data sequence for that customer in *UD*. Data sequences with same customer id in both databases must be merged into one data sequence before *GSP* starts. Likewise, *FASTUP* algorithm merges data sequences when necessary but relays practical information about discovered sequential patterns.

In order to retain as much information as possible, and to keep smaller number of data sequences for subsequent mining processes, we extract data sequences from *DB* and merge them to *db*. The merging is accomplished as follows.

1. With given data sequences, sorted by *cid*, in *db* and *DB*, we check *cid*s in *db* against *DB* to find any existence of that *cid*.
2. For the found data sequence *ds* in *DB*, the support counts of all frequent sequences contained in *ds* are decremented by one.
3. The *ds* is merged into that data sequence *ds'* with same *cid* in *db*, with *ds* followed by *ds'*.

Apparently, with sorted data sequence, searching *cid* could be done in very short time. With known sequences for checking, decrement operations add slightly overhead in merging process. After extraction and merging, there is no data sequence in *DB* and *db* with same customer id. *FASTUP* algorithm then could proceed to next stage.

3.3 FASTUP algorithm

Pass-1: find frequent 1-sequences in updated database.

1. Scan *db* for all 1-sequence X to get X_{count}^{db}.
2. If $X \in S_1^{DB} \wedge X_{count}^{DB} + X_{count}^{db} < min\_sup \times (|DB|+|db|)$, put X to the failed set, otherwise add X to S_1^{UD}.
3. In the same scan, if $X \notin S_1^{DB} \wedge X_{count}^{db} < min\_sup \times |db|$, X could not be frequent; otherwise, add X to C_1.
4. Scan *DB* for each $X \in C_1$ to get X_{count}^{DB}. If $X_{count}^{DB} + X_{count}^{db} \geq min\_sup \times (|DB|+|db|)$ at the end of pass-1, add X to S_1^{UD}.

In comparison with *GSP*, *FASTUP* discovers previous frequent 1-sequences that is still frequent, filters out those that are invalid now, and generates potential candidate 1-sequences within a scan on increment database. Every new candidate in the set of new candidate 1-sequences is then checked against the original database to see if it is frequent. Contrast to *GSP*, *GSP* takes every item as a candidate and counts over the whole database. *FASTUP* is obviously faster than *GSP* in this pass.

Pass-k: find frequent k-sequences in updated database.

1. Generate C_k from S_{k-1}^{UD} as *GSP* described [11]. C_k is characterized into two subsets. Let $C_{k1} = C_k \cap S_k^{DB}$, $C_{k2} = C_k - C_{k1}$. C_{k1} consists of candidates that are also frequent *k*-sequence in *DB*. While C_{k2} consists of remaining candidates that are "newly" generated.
2. A previous frequent *k*-sequence X in *DB*, which $X \in (S_k^{DB} - C_{k1})$, is not contained in C_k. Such X, though in S_k^{DB}, need not be checked against *db*.
3. Scan *db* for each X in C_k to get X_{count}^{db}. For each X in C_{k1}, since X_{count}^{DB} is available without checking *DB*, X is added to S_k^{UD} if it has minimum support.
4. In the same scan of *db*, for each X in C_{k2}, if $X_{count}^{db} < min\_sup \times |db|$, X could not be frequent. Such X is removed from C_{k2}. This step reduces the number of candidates to be checked against *DB*.
5. Scan *DB* for each $X \in C_{k2}$, compute X_{count}^{DB}. X is added to S_k^{UD} if $X_{count}^{DB} + X_{count}^{db} \geq min\_sup \times (|DB|+|db|)$ at the end of pass-*k*.

The above process is iterated until no more candidates are generated. At each pass, *FASTUP* updates supports of frequent sequences in *DB* which are still frequent in *UD*, and generates most likely frequent candidates within one scan over small increment *db*. Candidate sequences, which do not have sufficient supports relative to *db*, are pruned before they are verified by *DB*. The set of candidates that *FASTUP* generated for further checking with *DB* is smaller than *GSP*. Consequently, *FASTUP* could be much faster than previous algorithms of sequential pattern mining.

Complete FASTUP algorithm.

Algorithm Merge_Data_Sequences.

```
/* Let current data sequence be $ds_{db}$ */
/* Let cid of current data sequence be cid */
For_all data sequences in db do
  If cid is found in data sequence $ds_{DB}$ in DB then
    For_all frequent sequences contained in $ds_{DB}$ do
      Decrement support count of this sequence by 1
    End_for
    Replace $ds_{db}$ by $ds_{DB}$ merged with $ds_{db}$, where
      transactions in $ds_{DB}$ followed by transactions in $ds_{db}$
  End_if
End_for
```

Algorithm_Fast_Update_Sequential_Patterns.

```
/* PASS-1 */
/* Initialize all $X_{count}^{db}$ to zero, X is 1-sequence, i.e., 1-
itemsets. */
For_all data sequences in db do
  Increment $X_{count}^{db}$ by 1 if current data sequence
    contains X
End_for
For_all 1-sequence X do
  If X is frequent in DB, then
    If $X_{count}^{DB} + X_{count}^{db} \geq min\_sup \times (|DB|+|db|)$ then
      add X to $S_1^{UD}$
    End_if
  If X is not frequent in DB, then
    If $X_{count}^{db} \geq min\_sup \times |db|$ then
      add X to $C_1$
    End_if
End_for
For_all X in $C_1$ do
  Initialize $X_{count}^{DB}$ to zero
End_for
For_all data sequences in DB do
  For_all X in $C_1$ do
    Increment $X_{count}^{DB}$ by 1 if current data sequence
      contains X
  End_for
End_for
For_all X in $C_1$ do
  If $X_{count}^{DB} + X_{count}^{db} \geq min\_sup \times (|DB|+|db|)$ then
    add X to $S_1^{UD}$
End_for
/* All frequent 1-sequences & their counts accumulated*/
/* Pass-2 and beyond */
k = 2
Generate $C_k$ from $S_{k-1}^{UD}$
While $C_k$ is not empty do
  /* Initialize all $X_{count}^{db}$ to zero, X is k-sequence. in $C_k$ */
    For_all data sequences in db do
      Increment $X_{count}^{db}$ by 1 if current data sequence
        contains X
    End_for
    For_all k-sequence X in $C_k$ do
      If X is frequent in DB, then
        If $X_{count}^{DB} + X_{count}^{db} \geq min\_sup \times (|DB|+|db|)$ then
          add X to $S_k^{UD}$
        End_if
      If X is not frequent in DB, then
        If $X_{count}^{db} \geq min\_sup \times |db|$ then
          add X to $C_{k2}$
        End_if
    End_for
    For_all X in $C_{k2}$ do
      Initialize $X_{count}^{DB}$ to zero
    End_for
    For_all data sequences in DB do
      For_all X in $C_{k2}$ do
        Increment $X_{count}^{DB}$ by 1 if current data sequence
          contains X
      End_for
    End_for
    For_all X in $C_{k2}$ do
      If $X_{count}^{DB} + X_{count}^{db} \geq min\_sup \times (|DB|+|db|)$ then
        add X to $S_k^{UD}$
      End_if
    End_for
    /* All frequent k-sequences and their counts
    accumulated */
    k = k + 1
    Generate $C_k$ from $S_{k-1}^{UD}$
End_while
```

3.4 Examples on merging and FASTUP algorithm

Example 1. Given a original database *DB* which combines sequences in Fig. 2 and Fig. 3, with total 9 data sequences. If an increment database *db* in Fig. 4 with total 5 sequences is appended, the sequences of *cid*=1 and *cid*=4 would be extracted from *DB* and merged into *db*. Now *DB* has 7 and *db* has 5 data sequences. The count for <(8)>, <(2)>, <(1)>, and <(8)(1)> is decremented by 1 due to the extraction of data sequence with *cid*=1. Likewise, extraction of data sequence with *cid*=4 would cause support of <(1)> and <(4)> decreasing. After extraction and merging, *FASTUP* can be applied correctly since no cid appears in both *DB* and *db*.

Example 2. Given a database *DB*, and an increment database *db*, such that after data sequences are merged, $|DB|$=1000 and $|db|$=125. For *min_sup*=8%, in last discovery, after extraction and merging, found sequential patterns are listed in Table 1.

Let us consider sequential patterns related to items *1*, *2*, *3*, *4*, *5*. In order to find frequent 1-sequences in *UD*, a scan on *db* is made at first. Table 2 shows the result of the scan. Now <(1)> and <(3)> remain frequent with counts updated to 120 and 95, both counts > *8%*×(1000+125), i.e. 90. <(2)> is no longer frequent since its count is only 85. <(5)> cannot be frequent because its count in *db*, being 9, is less than *8%*×125. A scan on *DB* for new candidate <(4)> is then conducted. Suppose $<(4)>_{count}^{DB}$ =70, <(4)> is a new frequent 1-sequence with count 94. Before proceeding to 2-sequence counting, <(1,2)>, <(1)(2)> and <(3)(2)> are eliminated from counting in *db* for having subsequence <(2)>, which is not frequent in *UD*. In fact, candidate 2-sequences now consists of old frequent 2-sequences <(1,3)> and <(1)(3)>, new candidate 2-sequences <(1,4)>, <(1)(4)>, <(4)(1)>, <(3,4)>, <(3)(4)>, <(4)(3)>, and <(3)(1)>. Note that the support count for <(3)(1)> is not available since it failed to be frequent in *DB* and its count was not kept. After a scan on *db* for these candidates, the result is shown in Table 3. Now <(1)(3)> remains frequent with count=93. Another counting on *DB* is only required for <(1)(4)> and <(3)(4)> to check if they have minimum support. It is not necessary to check <(3)(1)> since its count in *db* is less than minimum required. The discovery continues until no more candidate sequences are generated.

Frequent sequences	Sequence	Support Count
S_1^{DB}	<(1)>	$<(1)>_{count}^{DB}$=100
	<(2)>	$<(2)>_{count}^{DB}$=83
	<(3)>	$<(3)>_{count}^{DB}$=87
S_2^{DB}	<(1,2)>	$<(1,2)>_{count}^{DB}$=82
	<(1,3)>	$<(1,3)>_{count}^{DB}$=81
	<(1)(2)>	$<(1)(2)>_{count}^{DB}$=83
	<(1)(3)>	$<(1)(3)>_{count}^{DB}$=86
	<(3)(2)>	$<(3)(2)>_{count}^{DB}$=81

Table 1. Sequential patterns and their counts in *DB* after merging and extraction

Sequence	Support Count
<(1)>	$<(1)>_{count}^{db}$=20
<(2)>	$<(2)>_{count}^{db}$=2
<(3)>	$<(3)>_{count}^{db}$=8
<(4)>	$<(4)>_{count}^{db}$=24
<(5)>	$<(5)>_{count}^{db}$=9

Table 2. 1-sequences and their counts in *db*

Sequence	Support Count
<(1,3)>	$<(1,3)>_{count}^{db}$=2
<(1)(3)>	$<(1)(3)>_{count}^{db}$=7
<(3)(1)>	$<(3)(1)>_{count}^{db}$=8
<(1,4)>	$<(1,4)>_{count}^{db}$=5
<(1)(4)>	$<(1)(4)>_{count}^{db}$=11
<(4)(1)>	$<(4)(1)>_{count}^{db}$=9
<(3,4)>	$<(3,4)>_{count}^{db}$=6
<(3)(4)>	$<(3)(4)>_{count}^{db}$=22
<(4)(3)>	$<(4)(3)>_{count}^{db}$=3

Table 3. Candidate 2-sequences and their counts in *db*

4. Conclusion and Future Works

Due to the nature of sequence permutation, the problem of sequential pattern mining is more complicated than the discovery of association rules. Without maintenance, validity of discovered patterns may change after update on database. We propose *FASTUP* algorithm that efficiently solves the problem by incremental updating without re-mining the whole updated database from scratch. Using frequent sequences and support counts discovered from original database, *FASTUP* rapidly updates frequent sequences and their counts by scanning over increment database instead of whole updated database. Fewer but more promising candidates are generated by just checking counts in increment database.

The simulation of our algorithm on synthetic data is underway. Given the analysis on sequential pattern mining, the performance of the algorithm could be much faster than previous algorithms for the maintenance of sequential patterns. Further researches could be extended to problems of various minimum supports and problems of generalized sequential patterns such as patterns with is-a hierarchy.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, Fast Discovery of Association Rules, Advances in Knowledge Discovery and Data Mining, edited by U. M. Fayyad et al, AAAI/MIT Press, pp. 307-328, 1996.

[2] R. Agrawal and R. Srikant, Mining Sequential Patterns, Proceedings of the 11th International Conference on Data Engineering (ICDE'95), pp. 3-14, Taipei, Taiwan, 1995.

[3] R. Agrawal and J. Shafer, Parallel Mining of Association Rules, IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 6, pp. 962-969, Dec. 1996.

[4] S. Brin, R. Motwani, J. Ullman and S. Tsur, Dynamic Itemset Counting and Implication Rule for Market Basket Data, Proceedings of the 1997 SIGMOD Conference on Management of Data, pp. 255-264, 1997.

[5] D. W. Cheung, J. Han, V. T. Ng and C. Y. Wong, Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique, Proceedings of International Conference on Data Engineering, 1996.

[6] J. Han and Y. Fu, Discovery of Multi-Level Association Rules from Large Databases. Proceedings of the 21st International Conference on Very Large Data Bases, pp. 420-431, Zurich, Switzerland, Sep. 1995.

[7] S. D. Lee, D. Cheung, B. Kao, A General Incremental Technique For Maintaining Discovered Association Rules, Proceedings of the 5th International Conference On Database Systems For Advanced Applications, pp. 185-194, Melbourne, Australia, Apr. 1997.

[8] H. Mannila and H. Toivonen, Discovering Generalized Episodes using Minimal Occurrences, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96), pp. 146-151, Portland, 1996.

[9] H. Mannila, H. Toivonen and A. I. Verkamo, Discovering Frequent Episodes in Sequences, Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95), pp. 210-215, Montreal, Canada, 1995.

[10] J. S. Park, M. S. Chen, and P. S. Yu., Using a Hash-Based Method with Transaction Trimming for Mining Association Rules, IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 5, pp. 813-825, 1997.

[11] R. Srikant and R. Agrawal, Mining Sequential Patterns: Generalizations and Performance Improvements, Advances in Database Technology–5th International Conference on Knowledge Discovery and Data Mining (KDD'95), pp. 269-274, Montreal, Canada, 1995.

[12] H. Toivonen, Discovery of Frequent Patterns in Large Data Collections, Ph.D. thesis, University of Helsinki, Finland, 1996.

[13] M. J. Zaki, Fast Mining of Sequential Patterns in Very Large Databases, Technical Report 668, The University of Rochester, New York, Nov. 1997.