JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 40, 201-213 (2024)
DOI: 10.6688/JISE.202401_40(1).0013

An Efficient Approach for Discovering and Maintaining Sequential Patterns

SHOW-JANE YEN AND YUE-SHI LEE+
Department of Computer Science and Information Engineering
Ming Chuan University
Taoyuan, 222 Taiwan
E-mail: {sjyen; leeys}@mail.mcu.edu.tw

Sequential pattern mining analyzes the ordered user behaviors, such as ordered list of the products purchased by most of the users. Because the user transactions will be increased every day, the current sequential patterns may be different from the previous ones. Therefore, how to efficiently update the original sequential patterns in real time is a very important research topic. If the original sequential patterns cannot be updated in time, then the information may no longer represent the user behaviors. For the previous studies in this area, some approaches may loss information, and some methods need to re-find the previous discovered patterns. In this article, we propose a novel approach for mining and maintaining the discovered sequential patterns without losing any information and re-discovering the existed patterns. The experiments also represent that our approach is more efficient than the current most efficient algorithm.

Keywords: sequential pattern maintenance, transaction database, data mining, user sequence, data stream

1. INTRODUCTION

Mining sequential patterns [2, 14] can find a set of ordered lists of purchasing behaviors for most users from a user transaction database. A transaction database contains a set of user transactions, which contain user identifier (UID), date and time of the transaction, and the purchased products (items) in the transaction. A transaction contains the products purchased together by a certain user. For example, a sequential pattern "<{computer}{printer}{toner cartridge}> 80%" means that 80% of the users purchase printer after purchasing computer, and they purchase toner cartridge after purchasing printer. This information can be used to predict which products the users will purchase after they purchased a set of products. Therefore, after a user bought a set of products, we can promote or recommend the items to the user according to the sequential patterns.

There are more terminologies for mining sequential patterns: A *sequence* can be denoted as an ordered list $<s_1, s_2, ..., s_n>$, which s_i is a set of items. A sequence $<y_1, y_2, \ldots, y_m>$ contains another sequence $<x_1, x_2, \ldots, x_n>$ if $x_1 \subseteq y_{i1}, x_2 \subseteq y_{i2}, \ldots, x_n \subseteq y_{in}$, in which $i_1 < i_2 < \ldots < i_n$, and $1 \leqq i_k \leqq m$. A *user sequence* is an ordered transaction purchased by a user according to the transaction date and transaction time, which is stored in a *user sequence database*. For example, Table 1 can be transformed to the user sequence database shown in Table 2.

The *support* (*sup*) of a sequence *s* is defined as the number of user sequences containing *s* divided by the total number of the user sequences in a user sequence database. The

Received October 21, 2022; revised December 9, 2022 & January 19, 2023; accepted January 29, 2023.
Communicated by Tzung-Pei Hong.
+ Corresponding author.

Table 1. A transaction database.

UID	Date, Time	Items
1	2016/1/1, 10:30	EF
1	2016/1/1, 11:36	EFH
1	2016/1/1, 11:40	FGH
2	2016/2/1, 12:20	EF
2	2016/3/2, 15:36	FH
3	2016/2/2, 09:12	FG
4	2016/1/23, 17:11	FH

Table 2. A user sequence database transformed from Table 1.

UID	User Sequence
1	<(EF)(EFH)(FGH)>
2	<(EF)(FH)>
3	<(GH)>
4	<(FH)>

number of the user sequences containing sequence *s* is the *support count* (SupCount) of sequence *s*. A sequence *s* is called a sequential pattern or a *frequent sequence* if the support of sequence *s* is no less than a user-defined *minimum support* (*min-sup*). A sequence containing *k* items is called a *k*-sequence, and a frequent sequence containing *k* items is called a frequent *k*-sequence. For example, the minimum support is set to 40% for Table 2, that is the minimum support count is 2 user sequences. Since 4-sequence <(EF)(FH)> is contained in two user sequences, and the support of <(EF)(FH)> is 2/4 = 50%, sequence <(EF)(FH)> is a frequent sequence or a sequential pattern.

Lin and *et al.* [10] proposed an algorithm FUSP-tree, which finds the frequent sequences according to the created FUSP-tree when some transactions are increased. The algorithm FUSP-tree is based on FUFP-tree [6] and the algorithm IncSpan [3]. That is, only frequent items stored in the tree structure, and the added transactions are scanned to count support for each item. There are four cases for each sequence when some transactions are added: Case 1. The sequence was originally frequent and is also frequent after adding the transactions. Case 2. The sequence was originally frequent, but becomes infrequent after adding the transactions. Case 3. The sequence was not frequent originally, but turns out to be frequent after adding transactions. Case 4. The sequence was not frequent originally and is not frequent either, after adding the transactions. For Cases 1-3, FUSP-tree algorithm takes a large amount of time to adjust and update the structure of FUSP-tree. For Case 3, because there is no information about infrequent item in the tree structure, FUSP-tree needs to rescan the original user sequence database to count supports for these sequences. After adjusting the FUSP-tree structure, the algorithm applies the algorithm FP-Growth [7] to re-discover all the sequential patterns.

Hijawi and Saheb [8] proposed an algorithm DSSPM for sequence pattern mining in data stream. This algorithm first scans the added user transactions to count support for each item, and removes the items which do not satisfy the minimum support threshold. For the items which meet the minimum support, this algorithm permutates these items in each added sequence to generate all the candidate set and increment the count for each permutation (*i.e.*, candidate sequence). The algorithm DSSPM finally inserts the permutations

(*i.e.*, frequent sequences) which achieve the minimum support into a sequence tree. Although this algorithm does not need to re-scan the original database, the counts of some items and sequences will be lost, since infrequent items were removed and there is no information to obtain the support of the originally infrequent sequence, such that the final set of frequent sequences may be incorrect. Moreover, this algorithm generates all the permutations of frequent items in all the user sequences, which is very time consuming to search and count all the candidate sequences.

The previous approaches [3, 5, 9-13] for maintaining sequential patterns also re-scan the original transaction database and re-find the previous generated sequential patterns, which are not efficient. Therefore, we propose an efficient approach for mining and maintaining sequential patterns to address the disadvantages of the previous approaches.

2. THE MAINTENANCE OF SEQUENTIAL PATTERNS

Because user transactions will continue to increase over time, we converted the transaction database into a *date-stamped user transaction database*, which records the user ID (UID), transaction date (time), and purchased items. The transaction date (time) is denoted by DT*i* ($i \geqq 1$) in order. As the date increases, the value of *i* also increases. Table 3 is an example of date-stamped user transaction database, which records the items purchased by each user on date DT*i*.

Table 3. A date-stamped user transaction database.

UID	DT1	DT2	DT3
1	F	G	FGH
2	E	FI	F
3	G	FGH	G

In this section, we represent our approach for mining and maintaining sequential patterns when a set of transactions on date DT*i* is added. There are two algorithms SPStream and SPStream_Ins proposed for mining and maintaining the sequential patterns. After describing SPStream algorithm, we point out the problems of SPStream and explain how SPStream_Ins can improve these problems.

2.1 SPStream Algorithm

SPStream considers the transactions day by day, and sequentially adds each item in the transactions to the *status table* of this item, which we designed to record the information for an item or a sequence according to the order of the transaction date. The status table for a sequence *s* includes the sequence *s*, its support count (SupCount), and the UID of the user who purchased the items in the sequence and the list of the dates the user purchased these items. For example, in Table 3, the status table for sequence <G> is shown in Table 4, where the support count of sequence <G> is 2, since there are 2 users which purchased item G. In Table 3, item G was purchased by user UID1 on dates DT2 and DT3, so the list of dates is (2, 3), and item G was purchased by user UID3 on three dates DT1, DT2 and DT3, so the list of dates is (1, 2, 3).

Table 4. The status table for sequence <G>.

<G>
SupCount: 2
UID 1: (2, 3)
UID 3: (1, 2, 3)

Table 5. The status tables for 1-sequences after adding the transactions on DT1.

<E>	<F>	<G>
SupCount: 1	SupCount: 1	SupCount: 1
UID 2: (1)	UID 1: (1)	UID 3: (1)

Our algorithm SPStream updates or creates the status tables for 1-sequences after adding the transactions on DT*i*. For example, assume *min-sup* = 50% in Table 3, that is the minimum support count is 3∗50% = 1.5, because there are three users. Table 5 shows the status tables for the sequences created after adding all the transactions on DT1.

If the supports of 1-sequences are no less than *min-sup*, then these 1-sequences can be joined with each other to generate candidate 2-sequences, and their status tables would be created. For instance, after adding the transactions on DT1, there is not any frequent item generated. SPStream adds the dates DT2 to UID2 and UID3 in the status table for sequence <F> after adding the transactions on DT2. At the same time, date DT2 is also added into UID1 in the status table for sequence <G> and the list of the dates for UID3 is updated as (1, 2), and the status tables for sequences <H> and <I> are created. Table 6 shows status tables for 1-sequences.

Table 6. The status tables for 1-sequences after adding the transactions on DT1 and DT2.

<E>	<F>	<G>	<H>	<I>
SupCount: 1	SupCount: 3	SupCount: 2	SupCount: 1	SupCount: 1
UID 2: (1)	UID 1: (1)	UID 1: (2)	UID3: (2)	UID2: (2)
	UID2: (2)	UID3: (1, 2)		
	UID3: (2)			

From Table 6, the 1-sequences <F> and <G> are frequent, SPStream generates the candidate 2-sequences <FG>, <GF>, <(FG)>, <FF> and <GG> and their status tables are created. For two frequent *k*-sequences $A = \langle a_1, a_2, ..., a_k \rangle$ and $B = \langle b_1, b_2, ..., b_k \rangle$, if $a_i = b_i$ ($\forall i$, $1 \leqq i \leqq k-1$) and $a_k \neq b_k$, then the candidate $(k+1)$-sequences $C1 = \langle a_1, a_2, ..., a_k, b_k \rangle$, $C2 = \langle a_1, a_2, ..., a_{k-1}, b_k, a_k \rangle$ and $C3 = \langle a_1, a_2, ..., a_{k-1}, (a_k, b_k) \rangle$ can be generated. For example, suppose there are two frequent 3-sequences $A = \langle p, q, r \rangle$ and $B = \langle p, q, s \rangle$. The two frequent 3-sequences can be joined to generate candidate 4-sequences $C1 = \langle p, q, r, s \rangle$, $C2 = \langle p, q, s, r \rangle$ and $C3 = \langle p, q, (r, s) \rangle$.

SPStream creates the status tables for candidate sequences $C1$, $C2$ and $C3$. If the status tables for frequent *k*-sequences *A* and *B* have the same UID, and the lists of dates for the UID are $(d_1, ..., d_n)$ and $(e_1, ..., e_m)$, respectively, the two lists of dates are scanned from left to right. If $d_1 < e_j$, and there is no e_h ($h < j$) such that $d_1 < e_h$, then the list of dates of the status table for sequence $C1$ in the UID is $(e_j, e_{j+1}, ..., e_m)$; For example, if the lists of dates

for A and B in the same UID are (2, 4, 5) and (1, 2, 3, 5), respectively, then the list of dates for $C1$ is (3, 5). If $e_1 < d_i$, and there is no d_h ($h < i$) such that $e_1 < d_h$, then the list of dates of the status table for sequence $C2$ in the UID is (d_i, d_{i+1}, ..., d_n); If $d_i = e_j$, ($1 \leqq j \leqq n$, $1 \leqq i \leqq m$), then d_i is inserted into the list of dates in this UID for the status table of sequence $C3$. For the above example, the list of dates for $C3$ is (2, 5).

Besides, SPStream generates the candidate sequences $<a_1, a_2, ..., a_k, a_k>$ and $<b_1, b_2, ..., b_k, b_k>$ for sequences A and B themselves, and the lists of dates for the two candidate sequences on the UID are (d_2, ..., d_n) and (e_2, ..., e_m), respectively. For the two frequent k-sequences $A = <a_1, a_2, ...a_{p-1}, (a_p, ..., a_k)>$ and $B = <a_1, a_2, ..., a_{p-1}, (a_p, ..., a_{k-1}, b_k)>$, and $a_k \neq b_k$, only one candidate sequence $<a_1, a_2, ..., a_{p-1}, (a_p, ..., a_k, b_k)>$ is generated, and d_i is inserted into the list of dates in this UID for the candidate sequence if $e_j = d_i$, ($1 \leqq j \leqq n$, $1 \leqq i \leqq m$). For the two frequent k-sequences $A = <(a_1, a_2, ..., a_k)>$ and $B = <(a_1, a_2, ..., a_{k-1}), b_k)>$, only one candidate ($k$+1)-sequences $<(a_1, a_2, ..., a_k)\ , b_k>$ can be generated. If $d_1 < e_j$, and there is no e_h ($h < j$) such that $d_1 < e_h$, then the list of dates for the status table of the candidate sequence in the UID is recorded as (e_j, e_{j+1}, ..., e_m).

For the example in Table 6, Table 7 shows the status tables for the candidate 2-sequences <FG>, <GF>, <(FG)>, <GG> and <FF>.

After adding the transactions on date DT3 in Table 3, for 1-sequence <F>, DT3 is inserted into the lists of dates in UID1 and UID2, and inserted into the lists of dates in UID1 and UID3 for 1-sequence <G>, and in UID1 for 1-sequence <H>. Table 8 shows the status tables for 1-sequences in Table 3.

Table 7. The status tables for candidate 2-sequences on DT1 and DT2.

<FG>	<GF>	<(FG)>	<GG>	<FF>
SupCount: 1	SupCount: 1	SupCount: 1	SupCount: 1	SupCount: 0
UID 1: (2)	UID 3: (2)	UID 3: (2)	UID 3: (2)	

Table 8. The status tables for 1-sequences after adding all the transactions in Table 3.

<E>	<F>	<G>	<H>	<I>
SupCount: 1	SupCount: 3	SupCount: 2	SupCount: 2	SupCount: 1
UID 2: (1)	UID 1: (1,3)	UID 1: (2,3)	UID 2: (3)	UID 2: (2)
	UID 2: (2,3)	UID3: (1,2,3)	UID 3: (2)	
	UID 3: (2)			

From Table 8, we can see that the generated frequent 1-sequences are <F>, <G>, and <H>. Therefore, the candidate 2-sequences <(FG)>, <FG>, <GF>, <(FH)>, <FH>, <HF>, <(GH)>, <GH>, <HG>, <FF>, <GG> and <HH> can be generated. If the status tables for the candidate sequences have been created, they only need to be updated. For example, items F, G and H are purchased by user UID1 on DT3, the status tables for the candidate sequences with last items F, G and H need to be updated. That is, UID1: (3) is added into the status table for sequence <GG>, and DT3 is also added into the list of dates in UID3 for <GG>. Table 9 shows the updated status tables for candidate 2-sequences. For the candidate sequences which their status tables have not been created, SPStream constructs the status tables for these candidate sequences as shown in Table 10.

From Tables 9 and 10, we can see that the generated candidate 2-sequences are frequent sequences. Table 11 shows the generated candidate 3-sequences and their status tables. From Table 11, the frequent 2-sequences <GF> and <GH> are joined to generate candidate sequence <G(FH)>; the frequent 2-sequences <GF> and <GG> are joined to generate candidate sequence <G(FG)>; the frequent 2-sequences <(FG)> and <(FH)> are joined to generate candidate sequence <(FGH)>; the frequent 2-sequences <(GG)> and <GH> are joined to generate <G(GH)>. Finally, the frequent 3-sequences <G(FH)> and <G(FG)> are joined to generate 4-sequence <G(FGH)>. Table 12 shows the status table for sequence <G(FGH)>.

Table 9. Updated status tables for candidate 2-sequences after adding the transactions.

<FG>	<GF>	<(FG)>	<GG>	<FF>
SupCount: 2	SupCount: 2	SupCount: 2	SupCount: 2	SupCount: 2
UID 1: (2,3)	UID 1: (3)	UID 1: (3)	UID 1: (3)	UID 1: (3)
UID 3: (3)	UID 3: (2)	UID 3: (2)	UID 3: (2,3)	UID 2: (3)

Table 10. Created status tables for candidate 2-sequences after adding the transactions.

<(GH)>	<GH>	<(FH)>
SupCount: 2	SupCount: 2	SupCount: 2
UID 1: (3)	UID 1: (3)	UID 1: (3)
UID 3: (2)	UID 3: (2)	UID 3: (2)

Table 11. The status tables for the candidate 3-sequences.

<G(FH)>	<G(GH)>	<G(FG)>	<FGH>
SupCount: 2	SupCount: 2	SupCount: 2	SupCount: 2
UID 1: (3)	UID 1: (3)	UID 1: (3)	UID 1: (3)
UID 3: (2)	UID 3: (2)	UID 3: (2)	UID 2: (2)

Table 12. The status table for the candidate 4-sequence.

<G(FGH)>
SupCount: 2
UID 1: (3)
UID 3: (2)

2.2 SPStream_Ins Algorithm

In this section, we present another algorithm SPStream_Ins to improve the efficiency of SPStream algorithm to reduce the number of generated candidate sequences. For any two frequent k-sequences ($k \geqq 2$), only one candidate $(k+1)$-sequence will be generated by SPStream_Ins. Let $V = \langle v_1, v_2, \ldots, v_k \rangle$ and $W = \langle w_1, w_2, \ldots, w_k \rangle$ be the two frequent k-sequences. If the two sequence $\langle v_1', v_2, \ldots, v_k \rangle$ and $\langle w_1, w_2, \ldots, w_{k-1}, w_k' \rangle$ are the same, in which itemset v_1' is formed by eliminating the first item from v_1 and w_k' is formed by eliminating the last item from w_k, then a candidate $(k+1)$-sequence can be generated by joining the two frequent k-sequences V and W. There are two cases for candidate generation:

The first case is that if w_k contains only one item x, then the candidate $(k+1)$-sequence is generated by concatenating x after v_k in sequence V. For instance, $V = <(A, B)\ (C)\ (D)>$ and $W = <(B)\ (C)\ (D)\ (E)>$ would be joined to generate candidate sequence $<(A, B)\ (C)\ (D)\ (E)>$. The second case is that if w_k contains more than one items and the last item is x, then the candidate $(k+1)$-sequence is generated by adding x into itemset v_k in sequence V. For instance, $V = <(A, B)\ (C)\ (D, E)>$ and $W = <(B)\ (C)\ (D, E, F)>$ would be joined to generate the candidate sequence $<(A, B)\ (C)\ (D, E, F)>$. For any two frequent items x and y, all the candidate 2-sequences $<xy>$, $<yx>$, $<(xy)>$ $<xx>$and $<yy>$ need to be generated.

If V and W are joined to generate a candidate sequence, and they have the same UID c in their status tables, then the list of dates for the UID c in status table for the candidate sequence can be generated as follows. Suppose the lists of dates for the UID c in the status tables of V and W are $(d_1, \ldots, d_n)$ and $(e_1, \ldots, e_m)$, respectively. If the last itemset in V contains only one item, $d_1 < e_j$ and $d_1 > e_k$ $(\forall ek, k < j)$, then the list of dates for UID c in the status table for the generated candidate sequence is $(e_j, e_{j+1}, \ldots, e_m)$. If the last itemset in W contains more than one item, and $e_j = d_i$ $(\forall j, i, 1 \leqq j \leqq m, 1 \leqq i \leqq n)$, then di will be added to the list of dates for UID c in the status table for the generated candidate sequence.

SPStream_Ins is more efficient than SPStream, since SPStream_Ins can generate less candidate sequences than SPStream, and can shorten the time for calculating the lists of dates. For example, for the two sequential patterns $<AB>$ and $<BC>$, only one candidate sequence $<ABC>$ will be generated for SPStream_Ins algorithm. However, for the two sequential patterns $<AB>$ and $<AC>$, SPStream generates three candidate sequences $<ABC>$, $<ACB>$ and $<A(BC)>$.

In the following, we describe how SPStream_Ins handles the newly added user sequences. When there are new users added into the user sequence database, the minimum support count will increase, such that some frequent sequences may turn out to be infrequent. If they turn out to be not candidate sequences either, SPStream_Ins removes them from the constructed status tables.

For the newly added transactions, SPStream_Ins updates the status tables and supports for the items in these transactions firstly. For a k-sequence, if it is not a frequent sequence originally, but becomes frequent after the update, then the frequent k-sequence can be joined with the other frequent k-sequences to generate candidate $(k+1)$-sequences, and the status tables also need to be constructed for these candidate sequences. If a k-sequence X is frequent before the update, but turns out to be infrequent after the update, then sequence X will be put in a Deletion Set *DelSequence*, and the status tables for the $(k+1)$-sequences containing X will be removed, since these $(k+1)$-sequences are no longer candidate sequences after the update.

For example, Table 13 is a date-stamped user transaction database after adding the transactions on date DT4 to Table 3, in which there are two users UID 4 and UID 5 added. Table 14 shows the 1-sequences and the status tables for these 1-sequences.

Table 13. The date-stamped user transaction database on date DT4.

UID	DT1	DT2	DT3	DT4
1	F	G	FGH	
2	E	FI	F	G
3	G	FGH	G	F
4				EFI
5				EG

Table 14. The status tables for 1-sequences in Table 13.

<E>	<F>	<G>	<H>	<I>
SupCount: 3	SupCount: 4	SupCount: 4	SupCount: 2	SupCount: 2
UID 2: (1)	UID 1: (1,3)	UID 1: (2,3)	UID 1: (3)	UID 2: (2)
UID 4: (4)	UID 2: (2,3)	UID 2: (4)	UID 3: (2)	UID 4: (4)
UID 5: (4)	UID 3: (2,4)	UID 3: (1,2,3)		
	UID 4: (4)	UID 5: (4)		

When the transactions for the new users UID4 and UID5 are added, the minimum support count is increased to 2.5 (5∗50%). The frequent 1-sequence <*H*> becomes infrequent due to the increase of minimum support count, indicating that the original candidate sequences which generated by joining 1-sequence <H> with the other sequences are no more candidate sequences. Therefore, sequence <H> is put into the deletion set *DelSequence*. And then, all the status tables for the 2-sequences containing the sequences in *DelSequence* are removed. For this example, the status tables for the 2-sequences <(FH)>, <FH>, <(GH)>, <GH>, <HF>, <HG> and <HH> are removed.

The status tables for the sequences containing <F> or <G> need to be updated, since frequent 1-sequences <F> and <G> remain frequent. Table 15 shows the updated status tables for the 2-sequences. The 1-sequence <E> was not frequent before adding the transactions on date DT4, and becomes frequent after adding the transactions on date DT4. Therefore, the candidate sequences will be generated by joining 1-sequence <E> with the other frequent 1-sequences, and the status tables for these candidate 2-sequences are created as above, which are shown in Table 16.

Table 15. Updated status tables for the 2-sequences after adding transactions on DT4.

<FG>	<GF>	<(FG)>	<GG>	<FF>
SupCount: 3	SupCount: 2	SupCount: 2	SupCount: 2	SupCount: 3
UID 1: (2,3)	UID 1: (3)	UID 1: (3)	UID 1: (3)	UID 1: (3)
UID 2: (4)	UID 3: (2,4)	UID 3: (2)	UID 3: (2,3)	UID 2: (3)
UID 3: (3)				UID 3: (4)

Table 16. Created status tables for the 2-sequences after adding the transactions on DT4.

<EF>	<FE>	<(EF)>	<EG>	<GE>	<(EG)>
SupCount:1	SupCount:0	SupCount:1	SupCount:1	SupCount:0	SupCount:1
UID 1: (2,3)		UID 4: (4)	UID 2: (4)		UID 5: (4)

After the updating, the frequent 2-sequences <(GF)>, <(FG)> and <GG> turn out to be infrequent, which are added into the deletion set *DelSequence* = {<H>, <(FG)>, <GF>, <GG>}. SPStream_Ins removes the status tables of the 3-sequences containing the sequences in the set *DelSequence*. Therefore, only the status table for 3-sequence <FFG> is remained after the deletion, which is generated by joining sequences <FF> and <FG>. So, the status table for the 3-sequence <FFG> need to be updated according to the status tables for sequences <FF> and <FG>. The 4-sequence <G(FGH)> will be removed since there is

no frequent 3-sequence generated. Finally, all the sequential patterns <F>, <G>, <FG> and <FF> will be generated.

When the set T of all the transactions on date d ($d \geqq 1$) is added, our proposed algorithm SPStream_Ins is shown in Algorithm 1, in which *STs* is the set of status tables for all the sequences generated from the original user sequence database; F is the set of all the frequent sequences, F_k is the set of all the frequent k-sequences, C_k is the set of all the candidate k-sequences, and I is the set of all items in the database. The proposed algorithm SPStream_Ins for maintaining sequential patterns is presented as Algorithm 1.

Algorithm 1: SPStream_Ins algorithm

```
    SPStream_Ins (STs, T, I, F, min-sup)
1     DelItemset = φ
2   for each transaction t in T {
3     for each item i in t {
4       update sequence structure for <i>
5       if i ∉ I
6         create status table for sequence <i>
7                 I = I ∪ {<i>}}}
8   for each item i∈ I {
9     if <i> ∈ F1 and sup(<i>) < min-sup {
10      F1 = F1 – {<i>}
11      DelItemset = DelItemset ∪ {<i>}}
12    if <i> ∉ F1 and sup(<i>) ≧ min-sup
13      F1 = F1 ∪ {<i>}}
14    k = 1;
15  while STs is updated {
16    Ck+1 = the set of candidates generated from Fk
17    for each candidate c∈Ck+1 {
18      if there is a subset of c in DelItemset {
19                remove the status table for sequence c
20                Ck+1 = Ck+1 – {c}
21                DelItemset = DelItemset ∪ {c}}}
22    for each candidate c∈Ck+1 {
23      if c exists in STs
24        update sequence structure for c
25      else
26                create status table for sequence c
27      if c∈Fk+1 and sup (c) < min-sup {
28        Fk+1 = Fk+1 – {c}
29        DelItemset = DelItemset ∪ {c}}
30      if c∉Fk+1 and sup(c) ≧ min-sup
31      Fk+1 = Fk+1 ∪ {c}}
32    k = k+1}
```

3. EXPERIMENTAL RESULTS

We compare the efficiency of the most efficient algorithm FUSP-tree [10] with our proposed algorithms SPStream and SPStream_Ins. The algorithm FUSP-tree is the extension of the two algorithms FUFP-tree and IncSpan [3] for discovering sequential patterns from the whole FUSP tree after adding the transactions. For synthetic dataset generation, we used IBM Generator in SPMF [4], where the number N of items is 10K, the number C of the total users is 1000, the average number T of the transactions per user is 4, and the average number I of the items contained in each transaction is 2. After generating the synthetic dataset, we randomly distribute the transactions for 168 days to facilitate the processing for daily transactions.

For the first experiment, the transactions are increased from the first date to the 168th date. The execution time for FUSP-tree, SPStream and SPStream_Ins are shown in Fig. 1-6, in which the minimum support (*min-sup*) is increased from 1.4% to 1.9%, respectively. From which x axis is the date, 28 days is an interval, and the execution time for each 28 days is shown in y axis.

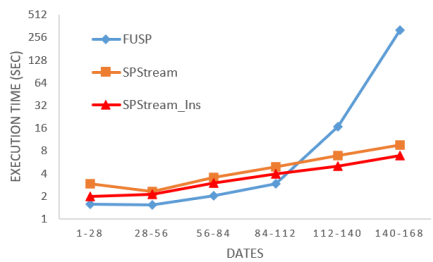


Fig. 1. The execution times with min-sup = 1.4%.

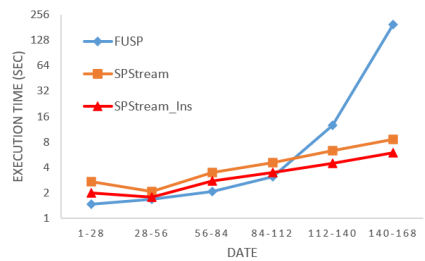


Fig. 2. The execution times with *min-sup* = 1.5%.

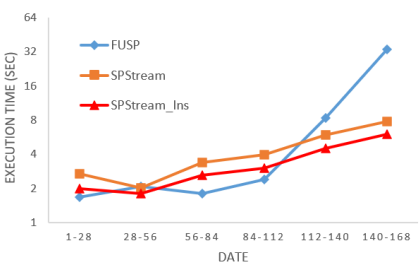


Fig. 3. The execution times with *min-sup* = 1.6%.

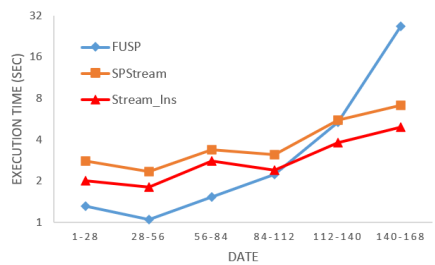


Fig. 4. The execution times with *min-sup* = 1.7%.

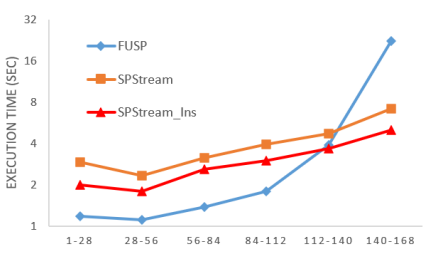


Fig. 5. The execution times with *min-sup* = 1.8%.

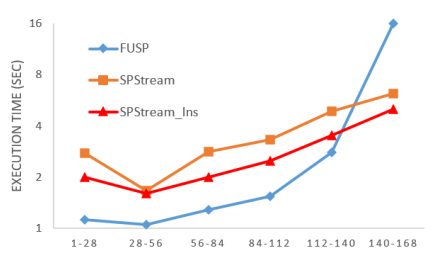


Fig. 6. The execution times with min-sup = 1.9%.

Because tree structure is relatively complex, FUSP-tree algorithm take much time to adjust the existing tree structure. From the updated tree structure, all the frequent sequences need to be re-discovered. SPStream and SPStream_Ins update and create the status tables from the shorter sequence to the longer sequence according to the newly added transactions, which is not necessary to re-discover the existing sequential patterns. From these experimental results, we can see that our algorithms initially need to take more time to construct the status tables for all the candidate sequences, but the tree structure for FUSP-tree algorithm only contains frequent items, which is small and easier to handle. Therefore, FUSP-tree takes less time than our algorithms at the beginning.

However, the transactions continue to increase as the number of dates increases. There are more and more frequent items generated, such that the tree structure for FUSP_tree is getting larger and more complicated. Since FUSP_tree needs to rescan the original large transaction database and more nodes in the tree structure need to be adjusted, it takes a large amount of time to re-discover all the frequent sequences. When the size of user transaction database is getting larger, the sequential patterns and candidate sequences gradually become stable. Our approach just updates a small part of status tables, and it is not necessary to re-processing the originally transaction database and re-find the previous frequent sequences. Therefore, the execution time for our algorithm would be much less than that of FUSP_tree. SPStream_Ins algorithm improves the candidate generation method for SPStream, which generates less candidate sequences and reduces the execution time for updating the status tables and calculating the list of dates, such that the storage space also can be reduced. Therefore, SPStream_Ins is more efficient than SPStream algorithm.

We generate another synthetic dataset I3T6 by setting the average number T of the transactions per user to 6, the average number I of the items contained in each transaction to 3, and the other parameters are the same as the previously generated dataset I2T4. The execution time from the first date to the 168th date for the three algorithms on the two datasets I2T4 and I3T6 under different minimum support threshold are shown in Fig. 7 and Fig. 8, respectively, from which we can see that SPStream and SPStream_Ins significantly outperform FUSP-tree as the support threshold decreases when the minimum support below 1.6%, because the nodes in FUSP-tree, and the time to update FUSP-tree and to re-discover the frequent sequences increase, as the minimum support decreases. Our approach just updates and creates the status tables according to added transactions, which is unnecessary to re-discover the frequent sequences. Therefore, FUSP-tree needs to take more time than our approach.

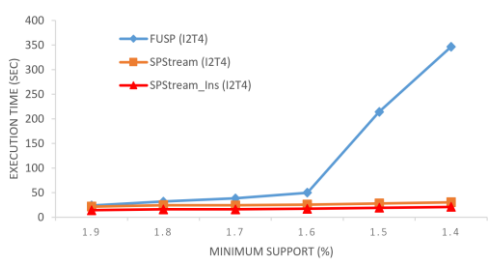


Fig. 7. The execution time for the three algorithms on dataset I2T4.

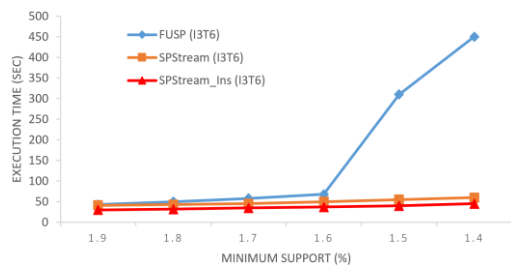


Fig. 8. The execution time for the three algorithms on dataset I3T6.

The memory space used by the algorithms FUSP-tree, SPStream and SPStream_Ins on the two datasets I2T4 and I3T6 under different minimum supports are shown in Figs. 9 and 10, respectively, in which our approach uses less memory space than FUSP-tree, since FUSP-tree needs large space to store the entire tree. Although our algorithms also need to store the status tables, they are all in digital form. Because SPStreams generates more candidate sequences than SPStream_Ins, it uses more memory space than SPStream_Ins algorithm. From these experiments, the execution time and memory usages for I3T6 are more than I2T4, since the number of transactions and the number of items contained in a transaction for I3T6 are more than those of I2T4.

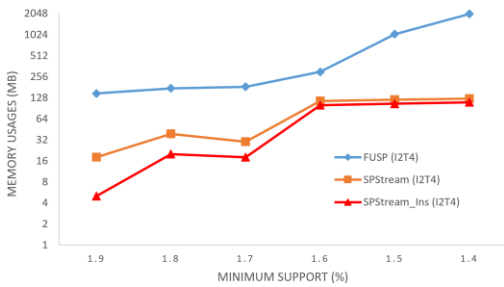


Fig. 9. The memory usages on dataset I2T4.

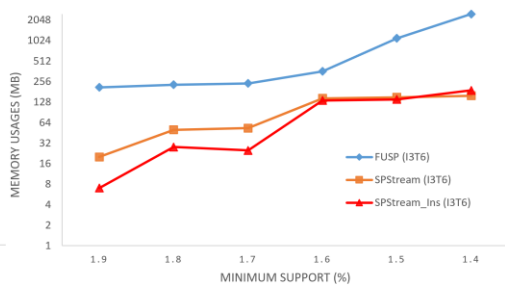


Fig. 10 The memory usages on dataset I3T6.

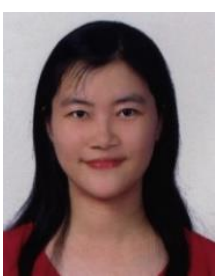
4. CONCLUSIONS

This article presents an efficient approach for maintaining and updating sequential patterns as the user transactions are added continuously, which can provide users understanding the users' purchasing behaviours in real time. Our algorithms mainly use and maintain the status tables for candidate sequences to update the sequential patterns during the process of transaction addition. The advantage of our algorithms is that the frequent sequences can be updated immediately, without re-discovering the existing sequential patterns and re-scanning the whole transaction database. Our algorithms outperform FUSP-tree algorithm and SPStream_Ins is better than SPStream algorithm according to the experiments. In the future, we will study how to further improve SPStream_Ins algorithm in terms of execution time and memory usages.

REFERENCES

1. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th Very Large Data Bases Conference*, 1994, pp. 487-499.
2. R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the 8th International Conference on Data Engineering*, 1995, pp. 3-14.
3. H. Cheng, X. Yan, and J. Han, "IncSpan: incremental mining of sequential patterns in large database," in *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 527-532.
4. P. Fournier-Viger, *et al.*, "SPMF: Open-source data mining platform," http://www.philippe-fournier-viger.com/spmf/index.php? link=datasets.php, 2015.

5. C. C. Ho, H. F. Li, F. F. Kuo, and S. Y. Lee, "Incremental mining of sequential patterns over a stream sliding window," in *Proceedings of the 16th IEEE International Conference on Data Mining Workshops*, 2006, pp. 677-681.
6. T. P. Hong, C. W. Lin, and Y. L. Wu, "Incrementally fast updated frequent pattern trees," *Expert Systems with Applications*, Vol. 24, 2008, pp. 2424-2435.
7. J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, Vol. 29, 2000, pp. 1-12.
8. H. M. Hijawi and M. H. Saheb, "Sequence pattern mining in data streams," *Computer and Information Science*, Vol. 8, 2015, pp. 64-70.
9. H. F. Li, C. C. Ho, H. S. Chen, and S. Y. Lee, "A single-scan algorithm for mining sequential patterns from data streams," *International Journal of Innovative Computing, Information and Control*, Vol. 8, 2012, pp. 1799-1820.
10. C. W. Lin, T. P. Hong, W. Y. Lin, and G. C. Lan, "Efficient updating of sequential patterns with transaction insertion," *Intelligent Data Analysis*, Vol. 18, 2014, pp. 1013-1026.
11. M. Y. Lin and S. Y. Lee, "Incremental update on sequential patterns in large databases," in *Proceedings of the 10th IEEE International Conference on Tools with Artificial Intelligence*, 1998, pp. 24-31.
12. F. Masseglia, P. Poncelet, and M. Teisseire, "Incremental mining of sequential patterns in large databases," *Data and Knowledge Engineering*, Vol. 46, 2003, pp. 97-121.
13. W. Ouyang, "Mining rare sequential patterns in data streams with a sliding window," in *Proceedings of International Conference on Systems and Informatics*, 2016, pp. 1023-1027.
14. J. Pei, "Mining sequential patterns by pattern-growth: the PrefixSpan approach," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, 2004, pp. 1-17.



Show-Jane Yen (顏秀珍) received the MS degree and the Ph.D. degree in Computer Science from National Tsing Hua University, Hsinchu, Taiwan, in 1993 and 1997, respectively. She is currently a Professor in the Department of Computer Science and Information Engineering, Ming Chuan University, Taoyuan, Taiwan. Her research interests include database management systems, data mining and data warehousing.

Yue-Shi Lee (李御璽) received the MS degree and the Ph.D. degree in Computer Science and Information Engineering from National Taiwan University, Taipei, Taiwan, in 1993 and 1997, respectively. He is currently a Professor in the Department of Computer Science and Information Engineering, Ming Chuan University, Taoyuan, Taiwan. His research interests include data mining, information retrieval and extraction.