Efficient Mining of High Utility Sequential Patterns Over Data Streams

Morteza Zihayat, Cheng-Wei Wu, Aijun An and Vincent S. Tseng

Technical Report EECS-2014-04

October 10 2014

Department of Electrical Engineering and Computer Science
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

Efficient Mining of High Utility Sequential Patterns Over Data Streams

Morteza Zihayat
Department of Computer Science and Engineering
York University
Toronto, Canada
Email: zihayatm@cse.yorku.ca

Cheng-Wei Wu
Department of Computer Science and Information Engineering
National Cheng Kung University
Taiwan, ROC
Email: silvemoonfox@gmail.com

Aijun An
Department of Computer Science and Engineering
York University
Toronto, Canada
Email: aan@cse.yorku.ca

Vincent S. Tseng
Department of Computer Science and Information Engineering
National Cheng Kung University
Taiwan, ROC
Email: tsengsm@mail.ncku.edu.tw

Abstract*—High utility sequential pattern mining has emerged as an important topic in data mining. Although several preliminary works have been conducted on this topic, the existing studies mainly focus on *mining high utility sequential patterns (HUSPs)* in static databases and do not consider the streaming data. Mining HUSPs over data streams is very desirable for many applications. However, addressing this topic is not an easy task. First, streaming data come continuously in high speed and the mining result should be instantly available when users request it. Second, we need to overcome the problem of combinatorial explosion of a large search space. Third, pruning search space for HUSP mining is difficult because the *downward closure property* does not hold for the utility of sequences. In this paper, we propose a new framework for *mining high utility sequential patterns over data streams*, which has not been explored previously. A novel data structure named *HUSP-Tree* is proposed to maintain the essential information for mining HUSPs. HUSP-Tree can be easily updated when new data arrive and old data expire in a data stream. An efficient and single-pass algorithm named *HUSP-Stream* is proposed to generate HUSPs from *HUSP-Tree*. When data arrive at or leave from a sliding window, *HUSP-Stream* incrementally updates *HUSP-Tree* online to find HUSPs based on previous mining results. *HUSP-Stream* uses a new utility estimation model to more effectively prune the search space. Experimental results on real and synthetic datasets show that our algorithm outperforms the state-of-the-art algorithms and serves as an efficient solution to the new problem of mining high utility sequential patterns over data streams.

I. INTRODUCTION

Even though *frequent sequential pattern mining* plays an important role in many data mining applications [13], in the traditional sequential pattern mining the number of occurrences of an item inside a transaction (e.g., quantity) is ignored in the problem setting, so is the importance (e.g., unit price/profit) of an item in the databases. Thus, not only some infrequent patterns that bring high profits to the business may be missed, but also a large number of frequent patterns having low selling profits are discovered. Motivated by these limitations, *high utility sequential pattern (HUSP) mining* has emerged as a novel research topic in data mining recently [2], [3], [16], [19]. In HUSP mining, each item has a weight (e.g., price/profit) and can appear more than once in any transaction (e.g., purchase quantity), and the goal is to find sequences whose total utility in the database is no less than a user-specified minimum utility threshold.

On the other hand, in recent years, many applications such as customer transactions in retail business, sensor networks and users web click streams in web applications generate huge volumes of *data as streams* [11]. Streaming data is considered as one of the main sources of big data. A significant part of such data is volatile, which means it needs to be analyzed in real time as it arrives. Mining big data streams faces three main challenges [11]: *volume*, *velocity* and *volatility*. Although traditional batch oriented systems such as MapReduce (i.e.,Hadoop) are able to scale-out and process very large volumes of data in parallel, they may suffer from the significant latency problem. *Data stream mining* is a research field to study methods for extracting knowledge from high-velocity and volatile data. Although a few studies have been proposed for HUSP mining [2], [3], [16], [19], existing studies consider mainly static databases. In this paper, we focus on finding HUSPs from high-velocity and evolving data streams.

As an example, in a retail dataset, each customer has a sequence of shopping transactions. In this dataset, a pattern like $\{(Cereal, Milk)\}$ is a frequent pattern, but its profit is very low. However, the pattern $\{(Necklace, Ring)\}$ is much less frequent than $\{(Cereal, Milk)\}$ but it often brings much more profit. These profitable patterns address several important questions in business area decisions such as how to maximize revenue or minimize marketing or inventory costs. Moreover, in reality, when a customer makes a new transaction, the transaction should be appended to her purchasing history sequence. Also at a time interval (such as one hour), there may be many active customers who update their purchasing sequences simultaneously. The effect of these transactions should be reflected to the mining results. Therefore, real-time additions, deletions and modifications of the transactions and mining results are needed in the real world applications.

Although mining HUSPs over high-velocity data streams is very desirable in many real-life applications, addressing this topic is not an easy task due to the following challenges. First, keeping all the data records in memory (even on disk) is infeasible and real-time processing of each new incoming record is required. On the other hand, once a data record is removed, it is prohibitively costly to backtrack over previously data records. Hence, how to efficiently discover HUSPs over data streams by reading data records only once using limited computing and storage capabilities is a challenging problem. Second, the *downward closure property*[19] does not hold for the utility of sequences. That is, the utility of a sequence may be higher than, equal to, or lower than that of its super/sub-sequences[3], [16], [19]. Thus, search space pruning techniques that rely on the downward closure property cannot be directly used for mining high utility sequential patterns. Third, mining HUSPs over a data stream of sequences need to overcome the large search space problem due to combinatorial explosion of sequences. Since items with different quantities and unit profits can occur simultaneously in any data record of data streams, the search space is much larger and the problem is much more challenging than mining HUSPs over static databases. Fourth, comparing to mining HUSPs from a static database, mining HUSPs over dynamic data streams has far more information to track and far greater complexity to manage. However, if an incorrect approach for tracking information is used, it may result in some HUSPs being pruned. Thus, how to effectively track the information of HUSPs without missing any HUSP is a challenging problem.

In this paper, we address all of the above deficiencies and challenges by proposing a new framework for *high utility sequential pattern mining over evolving data streams*. This problem has not been explored so far. The major contributions of this work are summarized as follows. (1)We incorporate the concept of stream mining into HUSP mining and formally define the new problem of mining high utility sequential patterns over data streams. (2) We propose two efficient data structures named *ItemUtilLists (Item Utility Lists)* and *HUSP-Tree* (*High Utility Sequential Pattern Tree*) for maintaining the essential information of high utility sequential patterns over a data stream. To the best of our knowledge, the *ItemUtilLists* structure is the first vertical data representation for HUSP mining over data streams that can be used to efficiently calculate the utility of sequences. (3) We also propose a novel over-estimate utility model, called *Sequence-Suffix Utility (SFU)* model. We prove that SFU of a sequence is an upper bound of the utilities of some of its super-sequences, which can be used to effectively prune the search space in finding HUSPs. (4) We propose a new one-pass algorithm called *HUSP-Stream (High Utility Sequential Pattern Mining over evolving Data Streams)* for efficiently constructing and updating *ItemUtilLists* and *HUSP-Tree* by reading a transaction in the data stream *only once*. (5)The effectiveness and efficiency of the proposed algorithm are evaluated extensively on real and synthetic datasets.

The remaining of the paper is organized as follows. In Section II, we discuss related work. Section III provides definitions and a problem statement. Section IV presents the proposed algorithms and data structures. Experimental results are shown in Section V. We conclude the paper in Section VI.

II. Related Work

Mining sequential patterns in sequence databases was first introduced by Agrawal et al [1]. A sequence is called *sequential pattern* or *frequent sequence* if its frequency in the sequence database is no less than a user-specified *support threshold* [1]. Sequential pattern mining has played an important role in data mining and many algorithms have been proposed, e.g., GSP [17], SPAM [5] and PrefixSpan [14]. These algorithms can be generally categorized as using a horizontal database (e.g., GSP) or a vertical representation of the database (e.g., SPAM). A vertical representation has the advantage of calculating frequencies of patterns without performing costly database scans. Algorithms using vertical representations perform better on datasets with dense or long sequences than the ones using the horizontal format. Although sequential pattern mining algorithms have been applied to solve various real-world problems [9], they treat all items as having the same importance/utility and assume that an item appears at most once at a time point, which is not the case for many applications.

High utility pattern (HUP) mining was proposed to address this limitation, which finds patterns (itemsets or sequences) whose utility is no less than a minimum utility threshold. The utility of a pattern is defined in a way to consider both the degree of importance of an item and its internal quantity inside a transaction. Most of HUP mining algorithms (e.g., a 2-phase algorithm in [12], IHUP [4], UP-Growth [18]) find high utility itemsets (HUIs) from a transaction database, where the sequential ordering of itemsets is not considered. To consider the sequential information, high utility sequential pattern (HUSP) mining has been studied very recently. To the best of our knowledge, only four studies (i.e., [2], [3], [16], [19]) have been conducted. The concept of HUSP mining was first proposed by Ahmed et al [2], who defined an over-estimated sequence utility measure, SWU, which has the downward closure property, and proposed the UL and US algorithms for mining HUSPs which use SWU to prune the search space. UL is a level-wise candidate generation-and-testing algorithm and hence involves mutiple scans of the database and generates a large number of high-SWU candidate sequences. US uses a pattern growth method inspired by PrefixSpan [14] to generate all sequences whose SWU satisfies the utility threshold, and then scan the database again to compute the exact utilities of high-SWU sequences to find HUSPs. Shie et al. [16] proposed a framework for mining HUSPs in a mobile environment. Their algorithm can only handle sequences with a single item in each sequence element. Ahmed et al. proposed efficient algorithms for mining *high utility access sequences* from web log data [3], which also only considered single-item sequences. Most recently, Yin et al. [19] proposed the USpan algorithm for mining HUSPs. They used a lexicographic tree to extract the complete set of high utility itemset sequences and designed mechanisms for expanding the tree with two pruning strategies. However, one of the pruning strategies needs to be used after candidate generation, which is not efficient. In addition, all of the HUSP mining methods were designed for static datasets, not for data streams.

Due to the widespread existence of data streams, stream mining has become one of the most important and challenging topics in data mining. Since a data stream is an unbounded,

(a) A Data Stream of Itemset-Sequences

SID \ TID	T_1	T_2	T_3	T_4	T_5	T_6	T_7	...
S_1	{(a,2)(b,3)(c,2)}	{(b,1)(c,1)(d,1)}		{(c,3)(d,1)}			{(b,1)(c,3)(d,2)}	...
S_2			{(b,4)}		{(a,4)(b,5)(c,1)}			...
S_3						{(a,2)(b,5)(e,2)}		...

SW_1

SW_2

(b) Profit Table

Item	a	b	c	d	e
Profit	2	3	1	4	3

Fig. 1. An example of a data stream of itemset-squences

fast, and dynamically-changing flow of data, a stream mining algorithm is often required to process each data record only once, and only the most recent or relevant data can be stored in memory. Incremental learning from new data is also required to provide fast response to the changes in data. Studies [7], [10], [6], [16] have been conducted to mine frequent sequential patterns over data streams. For example, Ho et al. proposed IncSPAM [10] to find sequential patterns over a data stream of itemset-sequences. Rassi et al. proposed the SPEED algorithm [6] for mining maximal sequential patterns over streaming data. Chang et al. proposed SeqStream [7] for mining closed sequential patterns over data streams. However, all these methods are for finding frequent sequential patterns and some useful infrequent patterns with high utility may be missed. So far, no study has been conducted to learn high utility sequential patterns from data streams, which is more challenging than finding frequent sequences due to the fact that the sequence utility does not satisfy the downward closure property.

In this paper, we will propose a framework for *mining high utility sequential patterns over data streams*. As surveyed above, no study was conducted to learn high utility sequential patterns from data streams, which is more challenging than finding frequent sequences over data streams and mining HUSPs in static databases.

III. Problem Statement

Let $I^* = \{I_1, I_2, \cdots, I_N\}$ be a set of *items*. An *itemset* is a set of distinct items. An *itemset-sequence* S (or *sequence* in short) is an ordered list of itemsets $\langle X_1, X_2, \cdots, X_Z\rangle$, where Z is the *size of* S. The *length* of S is defined as $\sum_{i=1}^{Z} |X_i|$. An *L-sequence* is a sequence of length L. A *sequence database* consists of a set of sequences $\{S_1, S_2, ..., S_K\}$, in which each sequence S_r has a unique sequence identifier r called SID and consists of an ordered list of transactions $\langle T_{d_1}, T_{d_2}, ..., T_{d_n}\rangle$. A transaction T_d in the sequence S_r is also denoted as S_r^d.

Definition 1: (***Data stream***) *A* data stream of itemset-sequences *(or* data stream *in short)* $DS = \langle T_1, T_2, ..., T_M\rangle$ *is an ordered list of transactions that arrive continuously in a time order. Each transaction* $T_i \in DS$ (1≤i≤M) *belongs to a sequence of transactions. A data stream can thus also be considered as a set of dynamically-changing sequences.*

Figure 1 shows a data stream $DS = \langle S_1^1, S_1^2, S_2^3, S_1^4, S_2^5, S_3^6, S_1^7\rangle$ with 7 transactions, each belonging to one of three sequences: S_1, S_2 and S_3.

Following, we define *transaction-sensitive sliding window* which not only considers new sequences, also a new element (e.g., item/itemset) can belong to an existing sequence.

Definition 2: (***Transaction-sensitive sliding window***) *Given a user-specified* window size w *and a data stream* $DS = \langle T_1, T_2, ..., T_M\rangle$, *a* transaction-sensitive sliding window SW *captures the* w *most recent transactions in* DS. *When a new transaction arrives, the oldest one is removed from* SW. *The* i-th *window over* DS *is defined as* $SW_i = \langle T_i, T_{i+1}, ..., T_{i+w-1}\rangle$.

According to the definition, transactions in a sliding window can belong to different sequences. Thus, a sliding window is actually a sequence database that changes over time. For example, in Figure 1, if the window size w is set to 5, the first and the second windows over DS are $SW_1 = \langle S_1^1, S_1^2, S_2^3, S_1^4, S_2^5 >$ (which has 2 sequences) and $SW_2 = \langle S_1^2, S_2^3, S_1^4, S_2^5, S_3^6\rangle$ (which has 3 sequences), respectively.

Definition 3: (***External utility and internal utility***) *Each item* $I \in I^*$ *is associated with a positive number* $p(I)$, *called its external utility, representing, e.g., the unit profit of* I. *Also, an item* I *in transaction* T_d *has a positive number* $q(I, T_d)$, *called its internal utility, representing, e.g., the quantity of* I *in* T_d.

Figure 1 gives the external utility (e.g., profit) of each item in DS in the profit table. The internal utility (e.g., quantity) of an item in a transaction is shown in the transaction. For example, $q(e, T_6) = 2$.

Definition 4: (***Utility of an item in a transaction***) *The utility of an item* I *in the transaction* T_d *of the sequence* S_r *is defined as* $u(I, S_r^d) = p(I) \times q(I, S_r^d)$.

Definition 5: (***Utility of an itemset in a transaction***) *Given itemset* $X \subseteq T_d$, *the utility of* X *in the transaction* T_d *of the sequence* S_r *is defined as* $u(X, S_r^d) = \sum\limits_{I \in X} u(I, S_r^d)$.

Definition 6: (***Transaction utility***) *The transaction utility of transaction* $S_r^d \in DS$ *is denoted as* $TU(S_r^d)$ *and computed as* $su(S_r^d, S_r^d)$.

For example, $u(b, S_1^1) = p(b) \times q(b, S_1^1) = 3 \times 3 = 9$, and $u(\{bc\}, S_1^1) = u(b, S_1^1) + u(c, S_1^1) = 9 + 2 = 11$. Therefore, *Transaction utility* of S_1^1 is $TU(S_1^1) = 2\times2+3\times3+1\times2 = 15$.

Definition 7: (***Occurrence of a sequence*** α ***in a sequence*** S_r) *Given a sequence* $S_r = \langle Y_1, Y_2, ..., Y_n\rangle$ *and a sequence* $\alpha = \langle X_1, X_2, ..., X_Z\rangle$ *where* Y_i *and* X_i *are itemsets,* α *occurs in* S_r *(or* α *is a subsequence of* S_r, *denoted as* $\alpha \preceq \beta$*) iff there exist integers* $1 \le e_1 < e_2 < ... < e_Z \le n$ *such that* $X_1 \subseteq Y_{e_1}, X_2 \subseteq Y_{e_2}, ..., X_Z \subseteq Y_{e_Z}$. *The ordered list of transactions* $\langle Y_{e_1}, Y_{e_2}, ..., Y_{e_Z}\rangle$ *is called an occurrence of* α *in* S_r. α *may have multiple occurrences in* S_r. *The set of all occurrences of* α *in* S_r *is denoted as* $OccSet(\alpha, S_r)$.

For example, in Figure 1, the set of all occurrences of the sequence $\langle\{ab\}\{c\}\rangle$ in S_1 in SW_1 is $OccSet(\langle\{ab\}\{c\}\rangle, S_1)$, is $\{\langle S_1^1, S_1^2\rangle, \langle S_1^1, S_1^4\rangle\}$.

Definition 8: (***Utility of a sequence*** α ***in a sequence*** S_r) *Let* $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z}\rangle$ *be an occurrence of* $\alpha = \langle X_1, X_2, ..., X_Z\rangle$ *in the sequence* S_r. *The utility of* α *w.r.t.* $\tilde{o}$ *is*

TABLE I. SUMMARY OF NOTATIONS

Notation	Description
$u(X, S_r^d)$	Utility of item/itemset X in transaction T_d of S_r
$TU(S_r^d)$	Utility of transaction T_d of sequence S_r
$\alpha \preceq \beta$	α is a subsequence of β, or α occurs in β
$OccSet(\alpha, S_r)$	Set of all the occurrences of α in sequence S_r
$su(\alpha, S_r)$	Utility of a sequence α in sequence S_r
$\alpha \oplus I$	Itemset-extended of sequence α and item I
$\alpha \otimes I$	Sequence-extended of sequence α and itemset $\{I\}$
$TSWU(\alpha, SW_i)$	Sequence weighted utility of sequence α in SW_i
$suffix(S_r, \alpha)$	Suffix of sequence S_r w.r.t sequence α
$SFU(\alpha, SW_i)$	Sequence-suffix utility of sequence α in SW_i

defined as $su(\alpha, \tilde{o}) = \sum_{i=1}^{Z} u(X_i, T_{e_i})$. *The utility of* α *in* S_r *is defined as* $su(\alpha, S_r) = \max\{su(\alpha, \tilde{o}) | \forall \tilde{o} \in OccSet(\alpha, S_r)\}$. *That is, the maximum utility of sequence* α *among all its occurrences in* S_r *is used as its utility in* S_r.

Definition 9: ***(Utility of a sequence in a sliding window)*** *The utility of a sequence* α *in the* i*-th sliding window* SW_i *over* DS *is defined as* $su(\alpha, SW_i) = \sum_{S_r \in SW_i} su(\alpha, S_r)$.

For example, let $\alpha = \langle\{ab\}\{c\}\rangle$. In SW_1 of Figure 1, $OccSet(\alpha, S_1) = \{\langle S_1^1, S_1^2\rangle, \langle S_1^1, S_1^4\rangle\}$. The utility of α in S_1 is $su(\alpha, S_1) = \max\{su(\alpha, \langle S_1^1, S_1^2\rangle), su(\alpha, \langle S_1^1, S_1^4\rangle)\} = \max\{14, 16\} = 16$. The utility of α in SW_1 is $su(\langle\{ab\}\{c\}\rangle, SW_1) = su(\alpha, S_1) + su(\alpha, S_2) = 16 + 0 = 16$.

Definition 10: ***(High utility sequential pattern (HUSP))*** *A sequence* α *is called a high utility sequential pattern (*HUSP*) in a sliding window* SW_i *iff* $su(\alpha, SW_i)$ *is no less than a user-specified minimum utility threshold* δ.

Problem statement. Given a minimum utility threshold δ, the problem of mining high utility sequential patterns over a data stream DS of transactions is to mine the complete set of itemset-sequences whose utility is no less than δ from the current transaction-sensitive sliding window over DS.

For convenience, Table I summarizes the concepts and notations we define in this paper.

IV. HUSP-STREAM ALGORITHM

In this section we propose a single-pass algorithm named *HUSP-Stream (High Utility Sequential Pattern mining over evolving data Stream)* for incrementally mining the complete set of HUSPs in the current window SW_i of a data stream based on the previous mining results for SW_{i-1}. We propose a vertical representation of the dataset called *ItemUtilLists (Item Utility Lists)* and a tree-based data structure, called *HUSP-Tree (High Utility Sequential Pattern Tree)*, to model the essential information of HUSPs in the current window.

The overview of *HUSP-Stream* is presented in Algorithm 1. The algorithm includes three main phases: (1) *Initialization phase*, (2) *update phase* and (3) *HUSP mining phase*. The initialization phase applies when the input transaction belongs to the first sliding window. In the initialization phase (lines 1-5), the *ItemUtilLists* structure is constructed for storing the utility information for every item in the input transaction S_r^i. When there are w transactions in the first window, *HUSP-Tree* is constructed for the first window. If there are already w transactions in the window when the new transaction S_r^i arrives, S_r^i is added to the window and the oldest transaction in

Algorithm 1 *HUSP-Stream*

Input: a new transaction S_r^i, window size w, minimum utility threshold δ, ItemUtilLists, HUSP-Tree
Output: ItemUtilLists, HUSP-Tree, HUSPs
1: **if** $i \le w$ *(when* S_r^i *is a transaction in the first window)* **then**
2: $\quad\forall$ item $\in S_r^i$, $put(r, i, u(item, S_r^i))$ to $ItemUtilLists(item)$
3: $\quad$**if** i = w **then**
4: $\quad\quad$Construct HUSP-Tree using ItemUtilLists and δ
5: **else**
6: $\quad$Update ItemUtilLists and HUSP-Tree using S_r^i, w and δ
7: **if** the user requests to get HUSPs for the current window **then**
8: $\quad$Return all the HUSPs by traversing HUSP-Tree once
9: **return** $ItemUtilLists$, $HUSP-Tree$, $HUSPs$ if requested

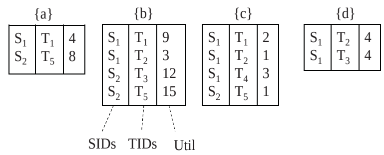

Fig. 2. *ItemUtilLists* for items in SW_1 in Figure 1

the window is removed. This is done by incrementally updating the *ItemUtilLists* and *HUSP-Tree* structures on line 6, which is the *update phase* of the algorithm. After the updating phase, if the user requests to find HUSPs from the new window, HUSP-Stream returns all the HUSPs to the user by traversing HUSP-Tree once.

A. Initialization phase

In this phase, HUSP-Stream reads the transactions in the first sliding window one by one to construct *ItemUtilLists* and *HUSP-Tree*. Below we first introduce these two structures and then explain how to construct them in the initialization phase.

1) ItemUtilLists: The first component of the proposed algorithm is an effective representation of items to restrict the number of candidates and to reduce the processing time and memory usage. *ItemUtilLists* is a vertical representation of the transactions in the sliding window. The *ItemUtilLists* of an item I consists of several tuples. Each tuple stores the utility of item I in the transaction S_v^u (i.e., transaction T_u in sequence S_v) that contains I. Each tuple has three fields: SID, TID and $Util$. Fields SID and TID store the identifiers of S_v and T_u, respectively. Field $Util$ stores the utility of I in S_v^u (Definition 4). Figure 2 shows *ItemUtilLists* for the first sliding window SW_1 in Figure 1.

2) HUSP-Tree Structure: A HUSP-Tree is a lexicographic tree where each non-root node represents a sequence of itemsets. Figure 3 shows part of the HUSP-Tree for the first window SW_1 in Figure 1, where the root is empty. Each node at the first level under the root represents a sequence of length 1, a node on the second level represents a 2-sequence, and all the child nodes of a parent are listed in alphabetic order of their represented sequences. There are two types of child nodes for a parent: *I-node* and *S-node*, which are defined as follows.

Definition 11: ***(Itemset-extended node (I-node))*** *Given a parent node* p *representing a sequence* α*, an* I-node *is a child node of* p *which represents a sequence generated by adding an item* I *into the last itemset of* α *(denoted as* $\alpha \oplus I$*).*

Definition 12: ***(Sequence-extended node (S-node))*** *Given a parent node* p *representing a sequence* α*, an* S-node *is a*

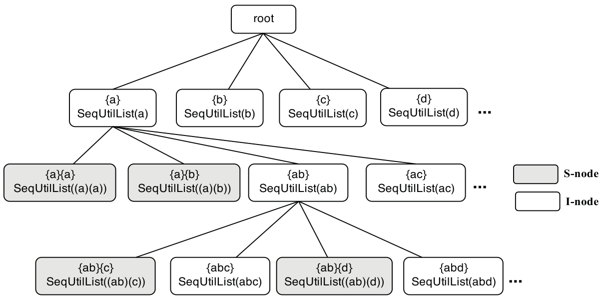

Fig. 3. An Example of HUSP-Tree for SW_1 in Figure 1

child node of p *which represents a sequence generated by adding a 1-Itemset* $\{I\}$ *after the last itemset of* α *(denoted as* $\alpha \otimes I$*).*

For example, in Figure 3, the node for sequence $\langle\{abc\}\rangle$ is an *I-node*, while the node for $\langle\{ab\}\{c\}\rangle$ is an *S-node*. Their parents are $\{ab\}$.

In data stream mining, the size of the tree can be very large since the number of possible patterns is exponential in the number of items. To avoid generating such a tree, we need to design strategies to prune the tree so that only the nodes representing *potential HUSPs* (to be defined later) are generated. These strategies will be presented later in this section. Moreover, we need to store summarized information regarding potential HUSPs to prune the tree during tree construction and updating, and identify HUSPs from these patterns. Hence, we design each non-root node of a HUSP-Tree to have a field, called *SeqUtilList*, for storing the needed information about the sequence represented by the node.

*Definition 13: (**Sequence Utility List**) The sequence utility list (*SeqUtilList*) of a sequence* α *in sliding window* SW_i *is a list of 3-value tuples, where each tuple* $\langle SID, TID, Util\rangle$ *represents an occurrence of* α *in the sequences of* SW_i *and the utility of* α *with respect to the occurrence. The* SID *in a tuple is the ID of a sequence in which* α *occurs,* TID *is the ID of the last transaction in the occurrence of* α*, and* $Util$ *is the utility of* α *with respect to the occurrence. The tuples in a* $SeqUtilList$ *are ranked first by* SID *and then by* TID*. The* $SeqUtilList$ *of* α *is denoted as* $SeqUtilList(\alpha)$*.*

For example, in Figure 1, if $\alpha = \langle\{a\}\{c\}\rangle$, α has two occurrences in SW_1, which are $\langle T_1, T_2\rangle$ and $\langle T_1, T_4\rangle$, the $SeqUtilList$ of α in SW_1 is $\{\langle S_1, T_2, (4+1)\rangle, \langle S_1, T_4, (4+3)\rangle\} = \{\langle S_1, T_2, 5\rangle, \langle S_1, T_4, 7\rangle\}$.

3) HUSP-Tree Nodes Construction : The first level of the tree under the root is constructed by using the items in *ItemUtilLists* as nodes. The *SeqUtilList* of these nodes is the *ItemUtilLists* of the items. Given a non-root node, its child nodes are generated using *I-Step* and *S-Step*, which generate *I-nodes* and *S-nodes* respectively. The processes of *I-Step* and *S-Step* are described below.

Given a node N representing sequence α, **I-Step** generates all the *I-nodes* of N (Definition 11). We define *I-Set* of α as the set of items occurring in the sliding window (i.e., in *ItemUtilLists*) that are ranked alphabetically after the last item in α. In **I-Step**, given an item I in the *I-Set* of α, for each tuple $Tp = \langle s, t, u\rangle$ in $SeqUtilList(\alpha)$, if there is a tuple $Tp' = \langle s', t', u'\rangle$ in $ItemUtilLists(I)$ such that $s = s'$ and $t = t'$, then add a new tuple $\langle s, t, (u+u')\rangle$ to $SeqUtilList(\beta)$, where $\beta = \alpha \oplus I$, and $SeqUtilList(\beta)$ was initialized to empty before the *I-Step*. An *I-node* representing β is added as a child node of N if $SeqUtilList(\beta)$ is not empty.

For example, if $\alpha = \langle\{a\}\rangle$ and I = b. To construct *SeqUtilList* of $\beta = \alpha \oplus I = \langle\{ab\}\rangle$, we find the tuples for common transactions from $SeqUtilList(\langle\{a\}\rangle) = \{\langle S_1, T_1, 4\rangle, \langle S_2, T_5, 8\rangle\}$ and $ItemUtilLists(b) = \{\langle S_1, T_1, 9\rangle, \langle S_1, T_2, 3\rangle, \langle S_2, T_3, 12\rangle, \langle S_2, T_5, 15\rangle\}$, which are the ones containing $\langle S_1, T_1\rangle$ and $\langle S_2, T_5\rangle$. Hence, $SeqUtilList(\langle\{ab\}\rangle)$ is $\{\langle S_1, T_1, (4+9)\rangle, \langle S_2, T_5, (8+15)\rangle\} = \{\langle S_1, T_1, 13\rangle, \langle S_2, T_5, 23\rangle\}$.

S-Step generates all the *S-nodes* for a non-root node. Given a node N for sequence α, the *S-Set* of α contains all the items that occur in the sliding window. The *S-Step* checks each item I in the *S-Set* to generate the *S-nodes* of N as follows. Let β be $\alpha \otimes I$ (i.e., a sequence by adding itemset $\{I\}$ to the end of α). First, $SeqUtilList(\beta)$ is initialized to empty. For each tuple $Tp = \langle s, t, u\rangle$ in $SeqUtilList(\alpha)$, if there is a tuple $Tp' = \langle s', t', u'\rangle$ in $ItemUtilLists(I)$ such that $s = s'$ and $t < t'$ (i.e., t' occurs after t), then a new tuple $\langle s, t', (u+u')\rangle$ is added to $SeqUtilList(\beta)$. If $SeqUtilList(\beta)$ is not empty, an *S-node* is created under the node N to represent β.

For example, if $\alpha = \langle\{ab\}\rangle$ and I = *d*. To construct *SeqUtilList* of $\beta = \alpha \otimes I = \langle\{ab\}\{d\}\rangle$, we need to find the tuples that satisfy the above conditions from $SeqUtilList(\langle\{ab\}\rangle) = \{\langle S_1, T_1, 13\rangle, \langle S_2, T_5, 23\rangle\}$ and $ItemUtilLists(d) = \{\langle S_1, T_2, 4\rangle, \langle S_1, T_3, 4\rangle\}$. The tuple $\langle S_1, T_1, 13\rangle$ in $SeqUtilList(\langle\{ab\}\rangle)$ and two tuples $\langle S_1, T_2, 4\rangle$ and $\langle S_1, T_3, 4\rangle$ in ***ItemUtilLists(d)*** satisfy the conditions. Hence, $SeqUtilList(\langle\{ab\}\{d\}\rangle)$ is $\{\langle S_1, T_2, (13+4)\rangle, \langle S_1, T_3, (13+4)\rangle\} = \{\langle S_1, T_2, 17\rangle, \langle S_1, T_3, 17\rangle\}$.

4) Pruning Strategies: In HUSP mining, the *downward closure property* does not hold for the sequence utility. Hence, the search space cannot be pruned as it is done in traditional sequential pattern mining. To effectively prune the search space, the concept of *Sequence-Weighted Utility (SWU)* was proposed in [2] to serve as an over-estimate of the true utility of a sequence, which has the downward closure property. However, this property has never been integrated into streaming environment. Below we incorporate SWU model into our proposed framework and propose a new model called *Transaction based Sequence-Weighted Utility (TSWU)* to effectively prune the search space.

Definition 14: The Transaction based Sequence-Weighted Utility (**TSWU**) of a sequence α in the i-th transaction-sensitive window SW_i, defined and denoted as follow: $TSWU(\alpha, SW_i) = \sum_{S \in SW_i \wedge \alpha \preceq S} \sum_{T \in S} TU(T)$, where *TU(T)* is the utility of transaction T.

For example, in SW_1 in Figure 1, there are two sequences S_1 and S_2 contain the sequence $\langle\{b\}\{c\}\rangle$. The $TSWU$ of $\langle\{b\}\{c\}\rangle$ in SW_1 is $TSWU(\langle\{b\}\{c\}\rangle, SW_i)$= (15+8+7) + (12+24) = 66.

Since it uses the utilities of all the transactions of all the sequences containing α in SW_i, $TSWU$ of a sequence is higher than the utility of α (i.e., Definition 9). That is, $TSWU(\alpha, SW_i) \geq su(\alpha, SW_i)$. The theorem below states that TSWU has the downward closure property over sliding window.

Theorem 1: Given a sliding window SW_i and two sequences α and β such that $\alpha \preceq \beta$, $TSWU(\alpha, SW_i) \geq TSWU(\beta, SW_i)$.

Proof: Let DS_α be the set of sequences containing α in SW_i and DS_β be the set of sequences containing β in SW_i. Since $\alpha \preceq \beta$, β cannot be present in any sequence where α does not exist. Therefore, $DS_\beta \subseteq DS_\alpha$. Thus, according to Definition 14 $TSWU(\alpha, SW_i) \geq TSWU(\beta, SW_i)$. ■

Since TSWU has the *downward closure property*, we can use it to prune the HUSP-Tree.

Pruning Strategy 1 (Pruning by TSWU): Let α be the sequence represented by a node N in the HUSP-Tree and δ be the minimum utility threshold. If $TSWU(\alpha, SW_i) < \delta$, there is no need to expand node N. This is because the sequence β represented by a child node is always a super-sequence of the sequence represented by the parent node. Hence $su(\beta, SW_i) \leq TSWU(\beta, SW_i) \leq TSWU(\alpha, SW_i) < \delta$, meaning β cannot be a HUSP.

Below we propose a novel concept called *Sequence-Suffix Utility (SFU)*, and then develop a new pruning strategy based on *SFU*.

Definition 15: **(First occurrence of a sequence α in the sequence S_r)** Let $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z} \rangle$ be an occurrence of a sequence α in the sequence S_r. $\tilde{o}$ is called the first occurrence of α in S_r if the last transaction in $\tilde{o}$ (i.e., T_{e_Z}) occurs before the last transaction of all the occurrences in $OccSet(\alpha, S_r)$.

For example, in Figure 1, the sequence $\langle\{a\}\{c\}\rangle$ has two occurrences $\langle T_1, T_2\rangle$ and $\langle T_1, T_4\rangle$ in S_1 for SW_1. $\langle T_1, T_2\rangle$ is the first occurrence because T_2 occurs earlier than T_4.

Definition 16: **(Suffix of a sequence S_r w.r.t. a sequence α)** Given sequence $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z} \rangle$ as the first occurrence of α in S_r. The suffix of S_r w.r.t. α (denoted as $suffix(S_r, \alpha)$) is the list of all the transactions in S_r after the last transaction in $\tilde{o}$ (i.e., after T_{e_Z}).

Definition 17: **(Sequence-Suffix utility of sequence α in sequence S_r)** Given a sequence $\alpha \preceq S_r$, the sequence-suffix utility of α in S_r is defined as follows: $SFU(\alpha, S_r) = su(\alpha, S_r) + \sum\limits_{T \in suffix(S_r, \alpha)} TU(T)$.

In other words, the sequence-suffix utility of a sequence in S_r is the utility of α in S_r plus the sum of the utilities of the transactions in the suffix of S_r with respect to α.

Note that for any non-root node N in the HUSP-Tree, $SFU(\alpha, S_r)$ can be computed easily using the information in the *SeqUtilList* of N. According to Definition 8, $su(\alpha, S_r) = \max\limits_{\tilde{o} \in OccSet(\alpha, S_r)} \{su(\alpha, \tilde{o})\}$ which can be obtained using the highest *Util* value among all the tuples with S_r as its SID. The *TID* field of the first tuple stores the TID of the last transaction in α's first occurrences in S_r. With this TID value, we can easily get the TIDs of all the transactions in $suffix(S_r, \alpha)$, and obtain their TU values (which were pre-computed and stored when a transaction was scanned to build *ItemUtilLists*). For example, the sequence-suffix utility of $\alpha = \langle\{a\}\{c\}\rangle$ in S_1 in Figure 1 is calculated as follow. According to $SeqUtilList(\alpha) = \{\langle S_1, T_2, 5\rangle, \langle S_1, T_4, 7\rangle\}$, $su(\alpha, S_1) = \max\{5, 7\} = 7$ and $suffix(S_1, \alpha) = \{T_4\}$. Hence, $SFU(\alpha, S_1) = 7 + TU(T_4) = 7 + 7 = 14$.

Definition 18: **(SFU of a sequence in a sliding window)** The SFU of a sequence α in the i-th window SW_i, denoted as $SFU(\alpha, SW_i)$, is defined as follows: $SFU(\alpha, SW_i) = \sum\limits_{S \in SW_i} SFU(\alpha, S)$.

The sequence-suffix utility value of α in a sliding window SW_i is an upper bound of the true utility of α in SW_i. That is, $su(\alpha, SW_i) \leq SFU(\alpha, SW_i)$.

Theorem 2: Given pattern α and sliding window SW_i and item I, $SFU(\alpha, SW_i)$ is an upper bound on:

1) the utility of pattern $\beta = \alpha \otimes I$. That is, $su(\beta, SW_i) \leq SFU(\alpha, SW_i)$.
2) the utility of any $\beta's$ offspring θ (i.e., any sequence prefixed with β). That is, $su(\theta, SW_i) \leq SFU(\alpha, SW_i)$.

Proof: Let $\beta = \alpha \otimes I$ and $S \in SW_i$. According to Definition 8, the utility of β can be rewritten as:

$$su(\beta, S) = \max_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o}) + u(I, \tilde{o})\}$$

Assume that I occurs in transaction $T_i \in \ddot{o}$ where $\ddot{o}$ is the occurrence with the maximum utility of β. We have $su(\beta, S) \leq \max\limits_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o}) + TU(T_i)\}$.

Since all occurrences of I are in $suffix(S, \alpha)$, $TU(T_i) \leq \sum\limits_{T \in suffix(S, \alpha)} TU(T)$. Therefore:

$$su(\beta, S) \leq \max_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o}) + \sum_{T \in suffix(S, \alpha)} TU(T)\}$$

The second part is independent of $\tilde{o}$. Thus, $su(\beta, S) \leq \max\limits_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o})\} + \sum\limits_{T \in suffix(S, \alpha)} TU(T) = SFU(\alpha, S)$.

Below we prove that utility of any offspring of β is less than $SFU(\alpha, S)$. Assume that $\theta = \alpha \otimes I \odot ... \odot ... \odot IS$ where IS is the last itemset in θ and $\odot \in \{\otimes, \oplus\}$. Let $\tilde{o}_1$ be the occurrence with maximum utility of θ in S. The utility of θ can be rewritten as follows:

$$su(\theta, \tilde{o}_1) = su(\alpha, \tilde{o}_1) + \sum_{i \in \theta \wedge i \in suffix(S, \alpha)} u(i, \tilde{o}_1)$$

Note that all items in θ which are not in α occur in $suffix(S, \alpha)$. We know that $su(\alpha, \tilde{o}_1) \leq su(\alpha, S)$. Hence:

$$su(\theta, \tilde{o}_1) \leq su(\alpha, S) + \sum_{i \in \theta \wedge T \in \tilde{o}_1 \wedge T \in suffix(S, \alpha)} u(i, T)$$

Since the utility of each item in a transaction is no more than the utility of the transaction, $su(\theta, \tilde{o}_1) \leq su(\alpha, S) + \sum\limits_{i \in T \wedge T \in suffix(S, \alpha)} TU(T) = SFU(\alpha, S)$.

The conclusion can be easily extended from S to SW_i. ■

Pruning Strategy 2 (Pruning by SFU): Let α be the sequence represented by a node N in the HUSP-Tree and δ

Algorithm 2 *TreeGrowth*

Input: $ND(\alpha)$*: node representing sequence* α
Output: *HUSP-Tree*

1: **if** $TSWU(\alpha, SW_i) < \delta$ **then**
2: remove node $ND(\alpha)$
3: **else**
4: $I\_Set \leftarrow$ items in $ItemUtilLists$ whose $TSWU >= \delta$ and whose id ranks lexicographically after the last item in the last itemset of α
5: **for** each item $\gamma \in I\_Set$ **do**
6: Compute $SeqUtilList(\alpha \oplus \gamma)$ using the I-Step
7: **if** $SeqUtilList(\alpha \oplus \gamma)$ is not empty **then**
8: Create I-node $ND(\alpha \oplus \gamma)$ as child of $ND(\alpha)$
9: Call Algorithm 2 ($ND(\alpha \oplus \gamma)$)
10: **if** $SFU(\alpha, SW_i) \geq \delta$ **then**
11: $S\_Set \leftarrow$ items in $ItemUtilLists$ whose $TSWU >= \delta$
12: **for** each item $\gamma \in S\_Set$ **do**
13: Compute $SeqUtilList(\alpha \otimes \gamma)$ using the S-Step
14: **if** $SeqUtilList(\alpha \otimes \gamma)$ is not empty **then**
15: Create S-node $ND(\alpha \otimes \gamma)$ as child of $ND(\alpha)$
16: Call Algorithm 2 ($ND(\alpha \otimes \gamma)$)

be the minimum utility threshold. If $SFU(\alpha, SW_i) < \delta$, there is no need to generate S-nodes from N. This is because the utility of $\alpha \otimes I$ and that of any $\alpha \otimes I$'s offspring is no more than $SFU(\alpha, SW_i)$, which is less than δ.

The pruning using SFU becomes more effective than $TSWU$ when the length of the pattern increases. That is, it may prune more low utility patterns at each deeper level of the HUSP-Tree. This is due to the fact that overestimation using SFU decreases as the length of the pattern increases. In other words, given a sequence α, to extend it using I-step or S-step and items in sequence S, the items are added from the end of first occurrence of α in S. And those items in S within the first occurrence are unable to form a new extension of α. However, for a sequence β formed by an itemset or sequence extension, the utilities of those items are added to $TSWU(\beta)$. For example in Table 1 $SFU(\langle\{a\}\{b\}\{c\}\rangle\}, S_1) = 10 + 14 = 24$ and $TSWU(\langle\{a\}\{b\}\{c\}\rangle\}, S_1) = 15 + 8 + 7 + 14 = 44$.

Using the proposed pruning strategies, our tree construction process will generate only the nodes that represent potential HUSPs, defined as follows.

Definition 19: **(Potential High Utility Sequential Pattern (i.e., PHUSP))** *A sequence α is called* PHUSP *in sliding window SW_i iff: (i) If the node representing α is an* I-node *and $TSWU(\alpha, SW_i) \geq \delta$ (ii) If the node representing α is an* S-node *and $SFU(\alpha, SW_i) \geq \delta$.*

5) HUSP-Tree Construction Algorithm: The complete tree construction process is as follows. The algorithm first generates the child nodes of the root as described in Section IV-A3. Then for each child node, the *TreeGrowth* algorithm (see Algorithm 2) is called to generate its *I-nodes* and *S-nodes* using the two pruning strategies and the I-Step and S-Step described in Section IV-A3. *TreeGrowth* is a recursive function and it generates all potential HUSPs in a depth-first manner. Given the input node $ND(\alpha)$, it first checks whether $TSWU(\alpha) < \delta$. If yes, the node is pruned. Otherwise, it generates the I-nodes from $ND(\alpha)$ using the I-Step (Lines 4-8) and recursively calls Algorithm 2 with each I-node. Then, the algorithm checks whether $SFU(\alpha)$ satisfies the threshold δ. If yes, it generates the S-nodes of $ND(\alpha)$ using the S-Step (Lines 11-15) and recursively calls the Algorithm 2 with each S-node.

B. Update Phase

When a new transaction S_v^u arrives, if the current window SW_i is full, the oldest transaction S_c^d expires. In this scenario, the algorithm needs to incrementally update *ItemUtilLists* and *HUSP-Tree* to find the HUSPs in SW_{i+1}. This process involves four types of updates: (i) inserting new sequences, (ii) deleting existing sequences, (iii) appending new items/itemsets to the existing sequences and (iv) dropping items/itemsets from the existing sequences.

Let H^+ be the complete set of HUSPs in the current sliding window SW_i, H^- be the complete set of HUSPs after a transaction removed from or added to SW_i, D^+ represents the window after transaction S_v^u is added to SW_i, D^- represents the window after S_c^d is removed from SW_i and S be a pattern found in SW_i. The following lemmas state how utility of S changes when a transaction is added to or removed from the window.

Lemma 1: Given sequence S, after S_v^u is added to the window, one of the following cases is held:

(1) If $S \preceq S_v$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^+) \geq su(S, SW_i)$.

(2) If $S \preceq S_v$ and $S \notin H^+$, then $su(S, D^+) \geq su(S, SW_i)$.

(3) If $S \npreceq S_v$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^+) = su(S, SW_i)$.

(4) If $S \npreceq S_v$ and $S \notin H^+$, then $S \notin H^-$ and $su(S, D^+) = su(S, SW_i)$.

Proof: Let S'_v be sequence S_v before transaction S_v^u is appended to and $OSet_{SW_i}$ be the set of occurrences of S in SW_i and $OSet_{D^+}$ be the set of occurrences of S in D^+. Below, we prove each case separately:

(1) Since $S \in H^+$, according to Definition 10, $su(S, SW_i) \geq \delta$. Also, $S \preceq S_v$ hence $OSet_{SW_i} \subseteq OSet_{D^+}$. In this case there is $o' \in OSet_{D^+}$ where $o' \notin OSet_{SW_i}$. If $su(S, o') > su(S, S'_v)$ then $su(S, D^+) > su(S, SW_i)$. Otherwise, $su(S, D^+) = su(S, SW_i)$. In both cases, since $su(S, SW_i) \geq \delta$ then $su(S, D^+) \geq \delta$ and $S \in H^-$.

(2) Since $S \preceq S_v$ hence $OSet_{SW_i} \subseteq OSet_{D^+}$. In this case there is $o' \in OSet_{D^+}$ where $o' \notin OSet_{SW_i}$. Also, $S \notin H^+$, according to Definition 10, $su(S, SW_i) < \delta$. If $su(S, o') > su(S, S'_v)$ then $su(S, D^+) > su(S, SW_i)$. Otherwise, $su(S, D^+) = su(S, SW_i)$.

(3) Since $S \npreceq S_v$ hence $OSet_{SW_i} = OSet_{SW_{i+1}}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{SW_{i+1}})$. Also, $S \in H$, according to Definition 10, $su(S, SW_i) \geq \delta$. Since the utility of S is the same, $S \in H^-$.

(4) Since $S \npreceq S_v$ hence $OSet_{SW_i} = OSet_{D^+}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{D^+})$. Also, $S \notin H^+$, according to Definition 10, $su(S, SW_i) < \delta$. Consequently, $su(S, D^+) < \delta$ so $S \notin H^-$. ■

Lemma 2: Given sequence S, sequence S'_c before S_c^d is removed from S_c, one of the following cases is held:

(1) If $S \preceq S'_c$ and $S \in H^+$, then $su(S, D^-) \leq su(S, SW_i)$.

(2) If $S \preceq S'_c$ and $S \notin H^+$, then $S \notin H^-$ $su(S, D^-) \leq su(S, SW_i)$.

(3) If $S \npreceq S'_c$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^-) = su(S, SW_i)$.

(4) If $S \npreceq S'_c$ and $S \notin H^+$, then $S \notin H^-$ and $su(S, D^-) = su(S, SW_i)$.

Proof:

Let $OSet_{SW_i}$ be the set of occurrences of S in SW_i and $OSet_{D^-}$ be the set of occurrences of S in D^-:

(1) Since $S \in H^+$, according to Definition 10, $su(S, SW_i) \geq \delta$. Also, since $S \preceq S'_c$ and $S_c \preceq S'_c$, hence $OSet_{D^-} \subseteq OSet_{SW_i}$. In this case there is $o' \in OSet_{SW_i}$ where $o' \notin OSet_{D^-}$. If $su(S, o') > su(S, S_c)$ then $su(S, D^-) < su(S, SW)$. Otherwise, $su(S, SD^-) = su(S, SW_i)$.

(2) Since $S \preceq S'_c$ and $S_c \preceq S'_c$, hence $OSet_{D^-} \subseteq OSet_{SW_i}$. In this case there is $o' \in OSet_{SW_i}$ where $o' \notin OSet_{D^-}$. Also, $S \notin H$, according to Definition 10, $su(S, SW_i) < \delta$. If $su(S, o') > su(S, S_c)$ then $su(S, D^-) < su(S, SW_i)$. Otherwise, $su(S, D^-) = su(S, SW_i)$. In both cases, $S \notin H^-$.

(3) Since $S \npreceq S'_c$ hence $OSet_{D^-} = OSet_{SW_i}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{D^-})$. Also, $S \in H^+$, according to Definition 10, $su(S, SW_i) \geq \delta$. Since the utility of S is the same, $S \in H^-$.

(4) Since $S \npreceq S'_c$ hence $OSet_{SW_i} = OSet_{D^-}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{D^-})$. Also, $S \notin H^+$, according to Definition 10, $su(S, SW_i) < \delta$. Consequently, $su(S, D^-) < \delta$ so $S \notin H^+$.

■

Below we propose an efficient approach to update *itemUtilLists* and *HUSP-Tree* based on Lemma 1 and Lemma 2.

The first step is to update ItemUtilLists. For each item γ in the oldest transaction S_c^d, the algorithm removes each tuple T_p whose SID and TID are c and d from $ItemUtilLists(\gamma)$. Then, the addition operation is invoked, which is performed as follows. For each item γ in the new transaction S_v^u, the algorithm inserts new tuple $\langle S_v, T_u, u(\gamma, S_v^u)\rangle$ to $ItemUtilLists(\gamma)$.

After updating *ItemUtilLists* of items, the algorithm uses the updated *ItemUtilLists* to update the TSWU value of items. The promising items (i.e., the items whose $TSWU$ is no less than the utility threshold) are collected into an ordered set *pSet*. For each item γ in $pSet$, if $ND(\gamma)$ is already under the root and its *SeqUtilList* has not been updated, the algorithm replaces the old *SeqUtilList* by the updated *ItemUtilLists* of item γ. If $ND(\gamma)$ has not been created under the root, the algorithm creates it under the root. Then, for each child node $ND(\alpha)$ under the root, the algorithm calls the procedure $UpdateTree(ND(\alpha))$ to update the sub-tree of $ND(\alpha)$, which is performed as follows. For each child node $ND(\beta)$ where β is $\alpha \oplus \gamma$ or $\alpha \otimes \gamma$ and $\gamma \in pSet$, the algorithm checks whether $ND(\beta)$ is already in the current HUSP-Tree. If $ND(\beta)$ is not in the HUSP-Tree, the algorithm constructs β's *SeqUtilList* using I-Step or S-Step and creates $ND(\beta)$ under

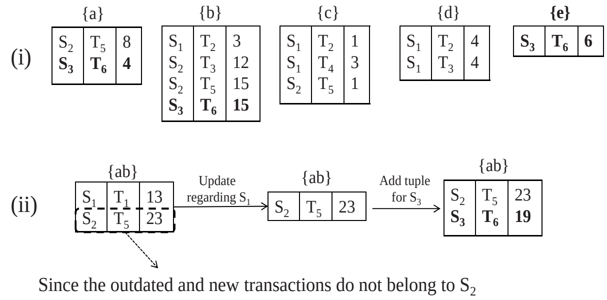

Fig. 4. The updated (i) $ItemUtilLists$ and (ii) $SeqUtilList(\{ab\})$ after removing T_1 from and adding T_6 to the window

$ND(\alpha)$. If $ND(\beta)$ is already in the HUSP-Tree, the algorithm incrementally updates the tuples in *SeqUtilList(β)* related to the new and oldest transactions as follows. Given the oldest transaction S_c^d and the newest transaction S_v^u, according to Lemma 1 and Lemma 2, the $SeqUtilList(\beta)$ should be updated if it has a tuple whose SID is either S_c or S_v. These tuples (not all the tuples in $SeqUtilList(\beta)$) are reconstructed by applying I-Step (if β is $\alpha \oplus \gamma$) or S-Step (if β is $\alpha \otimes \gamma$) on $SeqUtilList(\alpha)$ and $itemUtilLists(\gamma)$. Then the algorithm updates TSWU of β based on the updated $SeqUtilList(\beta)$. If TSWU of β is less than the utility threshold, the algorithm removes $ND(\beta)$ and the sub-tree under $ND(\beta)$. Otherwise, if β is $\alpha \oplus \gamma$, the algorithm calls the procedure *UpdateTree(*$ND(\beta)$*)* to update the sub-tree of $ND(\beta)$. If β is $\alpha \otimes \gamma$, the SFU of β is updated using the updated $SeqUtilList(\beta)$. If SFU of β is less than the threshold, node $ND(\beta)$ and its subtree are removed from the tree; otherwise, it recursively calls *UpdateTree(*$ND(\beta)$*)*.

Example 1 Figure 4 shows the updated $ItemUtilLists$ and $SeqUtilList(\{ab\})$ when T_1 is removed from and T_6 is added to the window. Note that we do not reconstruct the whole $SeqUtilList(\{ab\})$. Since T_1 belongs to S_1, we only need to update/remove the first tuple and also add a new tuple for the new sequence S_3. The other tuples are not updated. In this figure, since $\{ab\}$ is not in S_1 any more but exists in S_3, $SeqUtilList(\{ab\})$ is updated as $SeqUtilList(\{ab\}) = \{\langle S_2, T_5, 23\rangle, \langle S_3, T_6, 19\rangle\}$.

Since a tuple in $ItemUtilLists$ can be accessed directly and the number of tuples needed to be updated in $ItemUtilLists$ is $L_{oldest} + L_{new}$, where L_{oldest} is the length of the transaction to be removed and L_{new} is the length of the new transaction added to the sliding window, the average time complexity for updating $ItemUtilLists$ is $O(L_{avg})$, where L_{avg} is the average length of transactions in the data stream. The average time complexity for updating HUSP-Tree is $O(NumPot \times NumOccAff_{avg})$ where $NumPot$ is the number of potential high utility patterns in the new sliding window, and $NumOccAff_{avg}$ is the average number of occurrences of a potential high utility pattern in the sequences affected by the removal of the oldest transaction and the addition of the new transaction.

C. HUSP Mining Phase

HUSP mining phase is straight forward. After performing the update phase, *HUSP-Tree* maintains the information of the sequences in the current window. When users request

TABLE II. DETAILS OF PARAMETER SETTING

Dataset	#Seq	#Trans	#Items	w
BMS	77K	120K	3340	60K
DS1	100K	800K	1000	400K
ChainStore	400K	1000K	46,086	500K

the mining results, the algorithm performs the mining phase by traversing the HUSP-Tree once. For each traversed node $ND(\alpha)$, the algorithm uses the *SeqUtilList* of $ND(\alpha)$ to calculate the utility of α in the current window. If the utility of α is no less than the minimum utility threshold, the algorithm outputs α as a HUSP. After traversing the tree, all the HUSPs are outputted. Note that this HUSP mining phase can be combined with the update phase. During HUSP-Tree update, the utility of the sequence represented by each node can be computed. If the utility is no less than the threshold, the sequence can be outputted as a HUSP during the update phase.

V. EXPERIMENTS

In this section, we evaluate the performance of the proposed method. The experiments were conducted on an Intel(R) Core(TM) i7 2.80 GHz computer with 16 GB of RAM. Both synthetic and real datasets are used in the experiments. *Chainstore* is a real-life dataset acquired from [15], which already contains internal and external utilities. In order to use this dataset as a sequential dataset, we grouped transactions in different sizes so that each group represents a sequence of transactions. *BMS* is obtained from SPMF [8] which contains sequences of clickstream data from an e-retailer. A synthetic dataset *DS1:T3I2N1KD100K* was generated from the IBM data generator [1]. We follow previous studies [2] to generate internal and external utility of items for *BMS* and *DS1*. Table II shows characteristics of the datasets and parameter settings in the experiments. The *w* column of Table II shows the default window size for each dataset.

We use the following measures to evaluate the performance of the algorithms: (1) *Number of potential high utility sequential patterns* (*#$PHUSP$*): the total number of potential HUSPs produced by the algorithm in all sliding windows. (2) *Total execution time (sec.)*: the total execution time of the algorithms. (3) *Sliding Time (sec.)*: the average execution time of the algorithms to update data structures when a transaction arrives to or leaves from the window. (4) *Memory Usage (MB)*: the average memory consumption per window.

To the best of our knowledge, no study has been proposed for mining high utility sequential patterns over evolving data streams. Hence, we compare our method with USpan [19], which is the current best algorithm for mining high utility sequential patterns in static databases. Since the datasets used in the experiments are quite large and the window slides a large number of times, USpan runs very slow. To reduce the execution time of USpan, we modified USpan so that we run it per set of transactions (i.e., per batch). This approach is called $USpan\_Batch$. We set the size of each batch to 0.01% of whole transactions in data set. Moreover, in order to see the effect of using SFU to prune the tree in comparison to the other pruning strategy, $TSWU$, we implemented a basic version of HUSP-Stream in the experiments, called $HUSP\_TSWU$ which applies the $TSWU$ pruning strategy for pruning I-nodes and S-nodes.

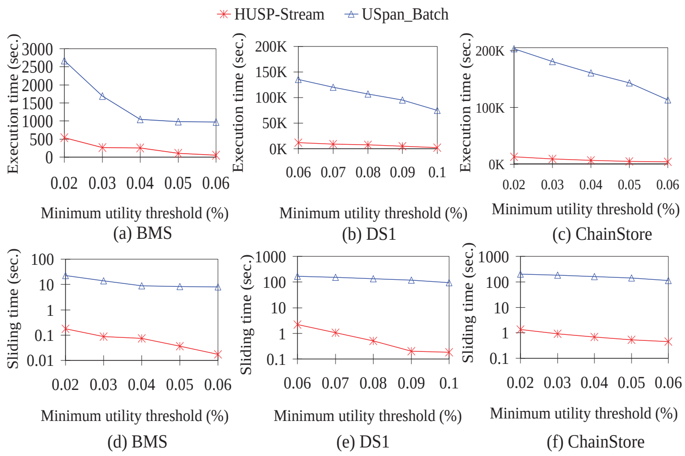

Fig. 5. Execution time and sliding time (shown in logarithmic scale) on different datasets

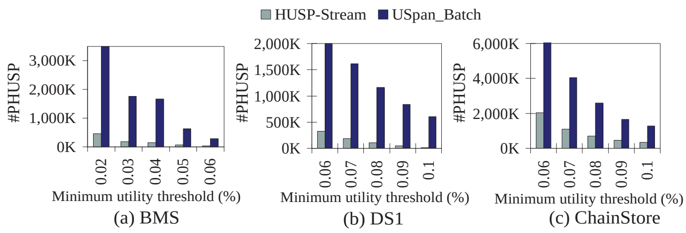

Fig. 6. Number of PHUSPs on different datasets

A. Time Efficiency of HUSP-Stream

Figure 5(a), Figure 5(b) and Figure 5(c) show the total execution time of the algorithms on each of the three datasets with different minimum utility threshold. As it is shown in the figure, HUSP-Stream is much faster than USpan_Batch. For example, HUSP-Stream runs 5 times faster on the BMS dataset and more than 10 times faster than USpan_Batch on DS1. Besides, it can be observed that HUSP-Stream is very scalable. Even under the low threshold, it can perform well. A reason is that USpan_Batch re-run the whole mining process, while HUSP-Stream performs incremental mining on each new window by efficiently updating its data structures.

Then we evaluate the average window sliding time of the algorithms under different minimum utility thresholds. Figure 5(d), Figure 5(e) and Figure 5(f) show the average window sliding time of the algorithms on BMS, DS1 and ChainStore respectively. For the dataset BMS, the average window sliding time of our algorithm is below 1 second, which is 100 times faster than that of USpan_Batch. For the largest dataset ChainStore, when the threshold is set to 0.04%, HUSP-Stream only spends 1.1 second, while USpan_Batch sends more that 260 seconds. In this case, HUSP-Stream is 200 times faster than the USpan_Batch.

B. Number of Potential HUSPs

In this section, we evaluate the algorithms in terms of the number of potential HUSPs (PHUSPs) produced by the algorithms. Figure 6 shows the results under different utility thresholds. For consistency across datasets, the minimum threshold is shown as a percentage of the total utility of all

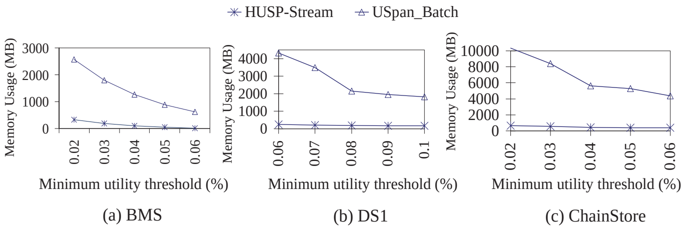

Fig. 7. Memory Usage of the algorithms

the sequences in a dataset. As shown in Figure 6, HUSP-Stream produces much fewer PHUSPs than USpan_Batch. For example, on BMS, when the threshold is 0.02%, the number of PHUSPs generated by USpan_Batch is 10 times more than that generated by HUSP-Stream. On the larger data sets, i.e., DS1 and ChainStore, the number of PHUSPs grows quickly when the threshold decreases. For example, on DS1, when the threshold is 0.06%, the number of PHUSPs produced by USpan_Batch is 14 times larger than that generated by HUSP-Stream. The main reason why our approach produces much fewer candidates is that HUSP-Stream incrementally updates HUSP-Tree by reusing the previous mining results. Hence it avoids regenerating a large number of intermediate PHUSPs during the mining process. Another reason is that our pruning strategies are more effective than the ones used in USpan_Batch.

C. Memory Usage

We also evaluate the memory usage of the algorithms under different utility thresholds. The results are shown in Figure 7, which indicate our approach consumes less memory than USpan_Batch. For example, for the dataset DS1, when the threshold is 0.06%, the memory consumption of HUSP-Stream is around 300 MB, while that of USpan_Batch is over 4,000 MB. A reason is that USpan_Batch produces too many PHUSPs during the mining process, which causes USpan_Batch to have more tree nodes than HUSP-Stream.

D. Effectiveness of SFU Pruning

In this section, we evaluate the use of SFU (in comparison to the use of only $TSWU$) for pruning the tree. To show effectiveness of the proposed pruning strategy, HUSP-Stream is compared to its basic version, *HUSP_TSWU*, which only applies the $TSWU$ pruning strategy for pruning I-nodes and S-nodes.

Figure 8(a), Figure 8(b) and Figure 8(c) illustrate the run time, the number of PHUSPs generated by the two methods, and their memory usage under different utility threshold values. These figures show that our new pruning strategy is more effective than using only *TSWU* in all three performance measures. Moreover, these figures show that the differences between the two pruning methods in the number of PHUSPs, run time and memory usage increase in general when the utility threshold decreases. These results indicate that our proposed SFU is much more effective than $TSWU$ in pruning.

E. Performance Evaluation with Window Size Variation

Below we evaluate the performance of the algorithms under different window sizes. In this experiment, the minimum

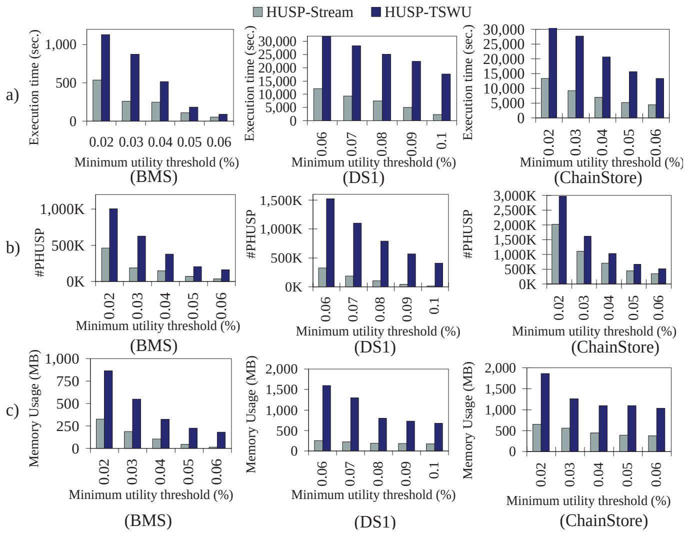

Fig. 8. Impact of SFU on (a) Run Time, (b) Number of PHUSPs and (c) Memory Usage.

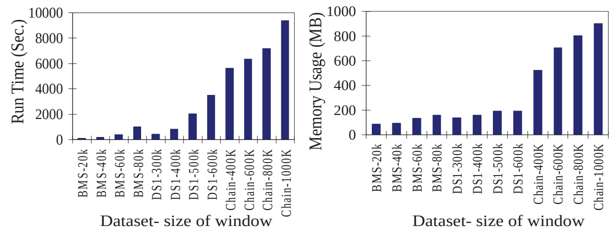

Fig. 9. Evaluation of HUSP-Stream under different window sizes

utility threshold is set to 0.03%,0.09%, 0.04% for the datasets BMS, DS1 and ChainStore, respectively. The results are shown in Figure 9. In Figure 9(a), each bar shows the memory consumption of HUSP-Stream on a data set under a window size. For example, the most left bar is the memory consumption of HUSP-Stream on BMS when the window size is set to 20,000 transactions. From Figure 9(a), we can observe that the memory consumption of HUSP-Stream increases very slowly with increasing window sizes. Figure 9(b) shows the execution time of HUSP-Stream under different window sizes. We can see that HUSP-Stream is also scalable in time with increasing window sizes.

F. Scalability

To further evaluate the scalability of HUSP-Stream, we generate a number of subsets of the BMS, DS1 and ChainStore datasets. The size of a subset ranges from 50% to 100% transactions of the dataset it is generated from. Figure 10 illustrates how the run time and memory usage of HUSP-Stream for producing HUSPs vary with different dataset sizes. We observe that the run time increases (almost) linearly when the number of transactions increases. This indicates that HUSP-Stream scales well with the size of dataset.

VI. CONCLUSIONS

In this paper, we proposed a novel framework for *mining high utility sequential patterns over data a stream*. We

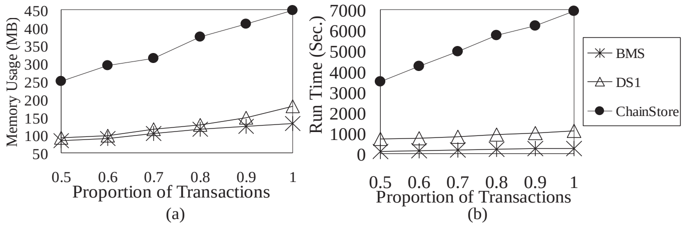

Fig. 10. Scalability of HUSP-Stream on different datasets: (a) Run Time, (b) Memory Usage

proposed a novel algorithm named *HUSP-Stream* to discover high utility sequential patterns in a transaction-sensitive sliding window over an itemset-sequence stream. Two data structures named *ItemUtilLists* and *HUSP-Tree (High Utility Sequential Pattern Tree)* are proposed to maintain the essential information of potential high utility sequences over data streams. When data arrive at or leave from the sliding window, *HUSP-Stream* incrementally updates *HUSP-Tree* and *ItemUtilLists* online to find high utility sequential patterns based on previous mining results. We also defined a new over-estimated sequence utility measure named *Suffix Utility (SFU)*, and used it to effectively prune the *HUSP-Tree*. Both real and synthetic datasets are used to show the performance of HUSP-Stream. In the experiments, we compared HUSP-Stream with USpan [19], a state-of-the-art algorithm for mining high utility sequential patterns in static databases. Extensive experimental results show that our approach substantially outperforms USpan and serves as an efficient solution to the new problem of mining high utility sequential patterns over data streams.

REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *In Proc. of ICDE Conf.*, pages 3–14, 1995.

[2] C. Ahmed, S. Tanbeer, and B. Jeong. A novel approach for mining high-utility patterns in sequence databases. *In ETRI Journal*, 32:676–686, 2010.

[3] C. Ahmed, S. Tanbeer, and B. Jeong. A framework for mining high utility web access sequences. *In IETE Journal*, 28:3–16, 2011.

[4] C. Ahmed, S. Tanbeer, and B.-S. Jeong. Interactive mining of high utility patterns over data streams. *Expert Systems with Applications*, 39:11979–11991, 2012.

[5] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *In Proc. of ACM SIGKDD Conf.*, pages 429–435, 2002.

[6] P. P. C. Rassi and M. Teisseire. Speed : Mining maximal sequential patterns over data streams. In *In Proc. of the IEEE Int'l Conf. on Intelligent Systems*, pages 546–552, 2006.

[7] L. Chang, T. Wang, D. Yang, and H. Luan. Seqstream: Mining closed sequential patterns over stream sliding windows. In *In Proc. ICDM Conf.*, pages 83–92, 2008.

[8] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C. Wu, and V. S. Tseng. Spmf: a java open-source pattern mining library. *JMLR*, 15:3389–3393, 2014.

[9] J. Gupta and J. Han. Applications of pattern discovery using sequential data mining. In *Pattern Discovery Using Sequence Data Mining: Applications and Studies*, pages 1–23, 2012.

[10] C. Ho, H. Li, F. Kuo, and S. Lee. Incremental mining of sequential patterns over a stream sliding window. In *In Proc. of the ICDMW*, pages 677–681, 2006.

[11] G. Krempl, I. Žliobaite, D. Brzeziński, E. Hüllermeier, M. Last, V. Lemaire, T. Noack, S. Sievi, M. Spiliopoulou, and J. Stefanowski. Open challenges for data stream mining research. *SIGKDD Explor. Newsl.*, 16(1):1–10, 2014.

[12] Y. Liu, W. k. Liao, and A. Choudhary. A fast high utility itemsets mining algorithm. In *Proceedings of the 1st international workshop on Utility-based data mining*, pages 90–99, 2005.

[13] C. H. Mooney and J. F. Roddick. Sequential pattern mining approaches and algorithms. *ACM Comput. Surv.*, 45(2):19:1–19:39, 2013.

[14] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *TKDE*, 16:1424–1440, 2004.

[15] J. Pisharath, Y. Liu, B. Ozisikyilmaz, R. Narayanan, W. K. Liao, A. Choudhary, and G. Mem-ik. Nu-minebench version 2.0 dataset. Technical Report CUCIS-2005-08-01, Northwestern University, 2005.

[16] B. Shie, H. Hsiao, and V. S. Tseng. Efficient algorithms for discovering high utility user behavior patterns in mobile commerce environments. *In KAIS*, 37:363–387, 2013.

[17] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *In Proc. of EDBT Conf.*, pages 3–17, 1996.

[18] V. Tseng, B. Shie, C.-W. Wu, and P. Yu. Efficient algorithms for mining high utility itemsets from transactional databases. *TKDE*, 25:1772–1786, 2013.

[19] J. Yin, Z. Zheng, and L. Cao. Uspan:an efficient algorithm for mining high utility sequential patterns. In *In Proc. of ACM SIGKDD Conf.*, pages 660–668, 2012.