Mining High Utility Sequential Patterns from Evolving Data Streams

Morteza Zihayat†, Cheng-Wei Wu†, Aijun An†, Vincent S. Tseng ‡
†Department of Electrical Engineering and Computer Science, †Department of Computer Science
‡York University, Toronto, Canada, ‡National Chao Tung University, Taiwan, ROC
zihayatl@cse.yorku.ca, silvemoonfox@gmail.com, aan@cse.yorku.ca, vtseng@cs.nctu.edu.tw

ABSTRACT
In this paper, we define the problem of mining high utility sequential patterns (HUSPs) over high-velocity streaming data and propose an efficient algorithm for mining HUSPs over a data stream. The main challenges we tackle include how to maintain a compact summary of the data stream to reflect the evolution of sequence utilities over time and how to overcome the problem of combinatorial explosion of a search space. We propose a compact data structure named HUSP-Tree to maintain the essential information for mining HUSPs in an online fashion. An efficient and single-pass algorithm named HUSP-Stream is proposed to generate HUSPs from HUSP-Tree. 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 serves as an efficient solution to the new problem of mining high utility sequential patterns over data streams.

CCS Concepts
•Information systems → Data streams; •General and reference → Metrics;

Keywords
High Utility Sequential Pattern Mining; Data Stream; Big Data;

1. INTRODUCTION
Even though frequent sequential pattern mining plays an important role in many data mining applications [6], 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. Motivated by these limitations, high utility sequential pattern (HUSP) mining has emerged as a novel research topic in data mining recently [2, 3, 8, 9]. 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.

In addition, many applications generate huge volumes of data in a streaming fashion, such as customer transactions in retail business, sensor networks and users web click streams in web applications [5]. 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 [5]: volume, velocity and volatility. 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, 8, 9], existing studies consider mainly static databases. In this paper, we focus on finding HUSPs from high-velocity and evolving data streams.

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. Second, the downward closure property of 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. Third, mining HUSPs over a data stream of sequences needs to overcome the large search space problem due to combinatorial explosion of sequences. 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.

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. First, 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. Then, a novel over-estimate utility model, called Sequence-Suffix Utility (SFU) is proposed. 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. We also 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.

2. 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 \( (X_1, X_2, \cdots, X_L) \), 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 \( \{S_1, S_2, \cdots, S_k\} \), in which each sequence \( S_i \) has a unique sequence identifier.
When a new transaction arrives, the oldest one is removed in time order. Each transaction $T_i$ in the sequence $S_r$ is defined as $\langle \alpha, S_r \rangle = \langle \{\alpha\}, S_r \rangle$, where $\alpha$ is a subsequence of $S_r$, denoted as $\alpha \preceq \beta$ if there exist integers $1 \leq e_1 < e_2 < \ldots < e_\beta \leq n$ such that $X_1 \subseteq Y_{e_1}, X_2 \subseteq Y_{e_2}, \ldots, X_\beta \subseteq Y_{e_\beta}$. The ordered list of transactions $\langle Y_{e_1}, Y_{e_2}, \ldots, Y_{e_\beta} \rangle$ is called an occurrence of $\alpha$ in $S_r$. $\alpha$ may have multiple occurrences in $S_r$. The set of all occurrences of $\alpha$ 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 $OccSet(\langle \{ab\}, \{c\} \rangle, S_1) = \{S_1^1, S_1^2, S_1^3\}$.

**Definition 8. (Utility of a sequence $\alpha$ in a sequence $S_r$)** Let $\delta = \langle T_{e_1}, T_{e_2}, \ldots, T_{e_\beta} \rangle$ be an occurrence of $\alpha = \langle X_1, X_2, \ldots, X_\beta \rangle$ in the sequence $S_r$. The utility of $\alpha$ w.r.t. $\delta$ is defined as $su(\alpha, \delta) = \sum_{r=1}^{z} u(X_r, T_{e_r})$. The utility of $\alpha$ in $S_r$ is defined as $su(\alpha, S_r) = \max\{su(\alpha, \delta) | \delta \in OccSet(\alpha, S_r)\}$. That is, the maximum utility of sequence $\alpha$ among all its occurrences in $S_r$ is used as its utility in $S_r$.

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

For example, let $\alpha = \langle \{ab\}, \{c\} \rangle$. In $SW_1$ of Figure 1, $OccSet(\alpha, S_1) = \{S_1^1, S_1^2, S_1^3\}$, $su(\alpha, S_1^3) = \max\{su(\alpha, S_1^1), su(\alpha, S_1^2), su(\alpha, S_1^3)\} = 16$. The utility of $\alpha$ in $SW_1$ is $su(\langle \{ab\}, \{c\} \rangle, SW_1) = su(\alpha, S_1^3) = 16$.

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

**Problem statement.** Given a minimum utility threshold $\delta$, 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 $\delta$ from the current transaction-sensitive sliding window over $DS$.

### 3. 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 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, the ItemUtilLists structure is constructed for storing the utility information for every item in the input transaction $S_r$. When there are $w$ transactions in the first window, HUSP-Stream is constructed for the first window. If there are already $w$ transactions in the window when the new transaction $S_r$ arrives, $S_r$ is added to the window and the oldest transaction in the window is removed. This is done by incrementally updating the ItemUtilLists and HUSP-Tree structures, 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.
3.1 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.

3.1.1 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 \( S^n \) (i.e., transaction \( T_n \) in sequence \( S_n \)) that contains \( I \). Each tuple has three fields: \( SID \), \( TID \), and \( Util \). Fields \( SID \) and \( TID \) store the identifiers of \( S_n \) and \( T_n \), respectively. Field \( Util \) stores the utility of \( I \) in \( S^n \) (Definition 4). Figure 2 shows ItemUtilLists for the first sliding window \( SW_1 \) in Figure 1.

3.1.2 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 \( \alpha \), an \( I \)-node is a child node of \( p \) which represents a sequence generated by adding an item \( I \) into the last itemset of \( \alpha \) (denoted as \( \alpha \oplus I \)).

**Definition 12.** (Sequence-extended node (S-node)) Given a parent node \( p \) representing a sequence \( \alpha \), an \( S \)-node is a child node of \( p \) which represents a sequence generated by adding a 1-itemset \( \{I\} \) after the last itemset of \( \alpha \) (denoted as \( \alpha \oplus \{I\} \)).

For example, in Figure 3, the node for sequence \( \{abc\} \) is an \( I \)-node, while the node for \( \{ab\} \} \) is an \( S \)-node. Their parents are \( \{ab\} \} \).

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 \( \alpha \) in sliding window \( SW_i \) is a list of 3-value tuples, where each tuple \((SID, TID, Util)\) represents an occurrence of \( \alpha \) in the sequences of \( SW_i \) and the utility of \( \alpha \) with respect to the occurrence. The \( SID \) in a tuple is the ID of a sequence in which \( \alpha \) occurs, \( TID \) is the ID of the last transaction in the occurrence of \( \alpha \), and \( Util \) is the utility of \( \alpha \) with respect to the occurrence. The tuples in a SeqUtilList are ranked first by \( SID \) and then by

---

**Figure 2:** ItemUtilLists for items in \( SW_1 \) in Figure 1

**Figure 3:** An Example of HUSP-Tree for \( SW_1 \) in Figure 1

TID. The SeqUtilList of \( \alpha \) is denoted as SeqUtilList(\( \alpha \)).

For example, in Figure 1, if \( \alpha = \{ab\} \} \) and \( I = b \). To construct SeqUtilList of \( \beta = \alpha \oplus I = \{ab\} \} \), we list the tuples for common transactions from SeqUtilList(\( \{ab\} \}) = \{(S_1, T_1, 4), (S_2, T_2, 8), (S_3, T_3, 12), (S_4, T_4, 15)\} and ItemUtilLists(\( \{ab\} \}) = \{(S_1, T_1, \{1\}), (S_1, T_2, \{2\}), (S_2, T_3, \{3\}), (S_3, T_4, \{4\})\}.

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

For example, if \( \alpha = \{ab\} \} \) and \( I = b \). To construct SeqUtilList of \( \beta = \alpha \oplus I = \{ab\} \} \), we list the tuples for common transactions from SeqUtilList(\( \{ab\} \}) = \{(S_1, T_1, 4), (S_2, T_2, 8), (S_3, T_3, 12), (S_4, T_4, 15)\} and ItemUtilLists(\( \{ab\} \}) = \{(S_1, T_1, \{1\}), (S_1, T_2, \{2\}), (S_2, T_3, \{3\}), (S_3, T_4, \{4\})\}.

**S-Step** generates all the \( S \)-nodes for a non-root node. Given a node \( N \) for sequence \( \alpha \), the S-Set of \( \alpha \) 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 \( \beta = \alpha \oplus I \) (i.e., a sequence by adding itemset \( \{I\} \) to the end of \( \alpha \)). First, SeqUtilList(\( \beta \}) is initialized to empty. For each tuple \( T_p = (s, t, u) \) in SeqUtilList(\( \alpha \}), if there is a tuple \( T_p′ = (s′, t′, u′) \) in ItemUtilLists(\( \alpha \}) such that \( s = s′ \) and \( t < t′ \) (i.e., \( t′ \) occurs after \( t \)), then a new tuple \( (s, t′, (u + u′)) \) is added to SeqUtilList(\( \beta \}). If SeqUtilList(\( \beta \}) is not empty, an \( S \)-node is created under the node \( N \) to represent \( \beta \).

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

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. 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 $\alpha$ in the $i$-th transactionsensitive window $SW_i$, defined and denoted as follow:

$$TSWU(\alpha, SW_i) = \sum_{s \in SW_i, \alpha \preceq s \subseteq T} UT(T),$$

where $UT(T)$ 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 $\{b\}$. The TSWU of $\{b\}$ in $SW_1$ is $TSWU((\{b\}), SW_1) = (15 + 8 + 7) + (12 + 24) = 66$.

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 $\alpha$ and $\beta$ such that $\alpha \preceq \beta$, $TSWU(\alpha, SW_i) \geq TSWU(\beta, SW_i)$.

Proof. Omitted due to space limit. It is provided in the full version in [11].

Pruning Strategy 1 (Pruning by TSWU): Let $\alpha$ be the sequence represented by a node $N$ in the HUSP-Tree and $\delta$ be the minimum utility threshold. If $TSWU(\alpha, SW_i) < \delta$, there is no need to expand node $N$. This is because the sequence $\beta$ 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 $\beta$ 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 $\alpha$ in the sequence $S_r$) Let $\hat{\alpha} = \langle T_{e_1}, T_{e_2}, ..., T_{e_2} \rangle$ be an occurrence of a sequence $\alpha$ in the sequence $S_r$. $\hat{\alpha}$ is the first occurrence of $\alpha$ in $S_r$ if the last transaction in $\hat{\alpha}$ (i.e., $T_{e_2}$) occurs before the last transaction of all the occurrences in $OccSet(\alpha, S_r)$.

For example, in Figure 1, the sequence $\langle \{a\} \rangle$ has two occurrences $\langle T_1, T_2 \rangle$ and $\langle T_1, T_3 \rangle$ in $S_1$ for $SW_1$. $\langle T_1, T_2 \rangle$ is the first occurrence because $T_2$ occurs earlier than $T_3$.

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

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

For example, the sequence-suffix utility of $\alpha = \langle \{a\} \rangle$ in $S_1$ in Figure 1 is calculated as follow. According to SeqUtilList$(\alpha) = \{(S_1, T_2, \{a\}), (S_1, T_1, \{a\})\}, su(\alpha, S_1) = \max\{5, 7\} = 7$ and $\text{suffix}(S_1, \alpha) = \{T_2\}$. Hence, $SFU(\alpha, S_1) = 7 + UT(T_2) = 7 + 7 = 14$.

Algorithm 1: TreeGrowth

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

1. if TSWU(\alpha, SW_i) < \delta then
2. remove node ND($\alpha$)
3. else
4. $I_set = \{\text{items in ItemUtilLists whose TSWU } \geq \delta \}$ and whose $id$ ranks lexicographically after the last item in the last itemset of $\alpha$
5. for each $i \in I_set$ do
6. Compute SeqUtilList($\alpha \oplus \gamma$) using the I-Step
7. if SeqUtilList($\alpha \oplus \gamma$) is empty then
8. Call Algorithm 1 ($ND(\alpha \oplus \gamma)$)
9. if $SFU(\alpha, SW_i) \geq \delta$ then
10. $S_{Set} = \{\text{items in ItemUtilLists whose TSWU } \geq \delta \}$
11. for each $i \in S_{Set}$ do
12. Compute SeqUtilList($\alpha \oplus \gamma$) using the S-Step
13. if SeqUtilList($\alpha \oplus \gamma$) is empty then
14. Create S-node $ND(\alpha \oplus \gamma)$ as child of $ND(\alpha)$
15. Call Algorithm 1 ($ND(\alpha \oplus \gamma)$)

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

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

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

Proof. Omitted due to space limit. It is provided in the full version in [11].

Pruning Strategy 2 (Pruning by SFU): Let $\alpha$ be the sequence represented by a node $N$ in the HUSP-Tree and $\delta$ 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 $\alpha \oplus I$ and that of any $\alpha \oplus I$’s offspring is no more than $SFU(\alpha, SW_i)$, which is less than $\delta$.

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 $\alpha$ is called PHUSP in sliding window $SW_i$ iff: (i) If the node representing $\alpha$ is an I-node and TSWU($\alpha, SW_i$) $\geq \delta$ (ii) If the node representing $\alpha$ is an S-node and SFU($\alpha, SW_i$) $\geq \delta$.

3.1.5 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 3.1.3. Then for each child node, the TreeGrowth algorithm (see Algorithm 1) 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 3.1.3. 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 $\delta$. 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.
3.2 Update Phase

When a new transaction $S^n_i$ arrives, if the current window $SW_i$ is full, the oldest transaction $S^n_{i-1}$ expires. In this scenario, the algorithm needs to incrementally update ItemUtilLists and HUSP-Tree to find the HUSPs in $SW_{i+1}$. 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 $S^n_i$ is added to $SW_i$, $D^-$ represents the window after $S^n_{i-1}$ 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 $S^n_i$ is added to the window, one of the following cases is held: (1) If $S \subseteq S_i$ and $S \in H^+$, then $S \in H^+$ and $su(S, D^+) \geq su(S, SW_i)$. (2) If $S \not\subseteq S_i$ and $S \not\in H^+$, then $su(S, D^+) \geq su(S, SW_i)$. (3) If $S \not\subseteq S_i$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^-) = su(S, SW_i)$. (4) If $S \not\subseteq S_i$ and $S \not\in H^+$, then $S \not\in H^-$ and $su(S, D^-) = su(S, SW_i)$.

**Proof.** Omitted due to space limit. It is provided in the full version in [11].

**Lemma 2.** Given sequence $S$, sequence $S'$ before $S^n_i$ is removed from $S_i$, one of the following cases is held: (1) If $S \subseteq S_i$ and $S \in H^-$, then $su(S, D^-) \leq su(S, SW_i)$. (2) If $S \not\subseteq S_i$ and $S \not\in H^-$, then $S \not\in H^-$. (3) If $S \not\subseteq S_i$ and $S \in H^-$, then $S \not\in H^-$ and $su(S, D^-) = su(S, SW_i)$. (4) If $S \not\subseteq S_i$ and $S \not\in H^-$, then $S \not\in H^-$ and $su(S, D^-) = su(S, SW_i)$.

**Proof.** Omitted due to space limit. It is provided in the full version in [11].

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 $\gamma$ in the oldest transaction $S^i_1$, the algorithm removes each tuple $T_\gamma$ 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 $\gamma$ in the new transaction $S^n_i$, the algorithm inserts new tuple $(S_i, T_\gamma, \mu(\gamma, S^n_i))$ 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 $\gamma$ 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 $\gamma$. 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 $\beta = \alpha \oplus \gamma$ or $\alpha \ominus \gamma$, the algorithm checks whether $\beta$ is already in the current HUSP-Tree. If $\beta$ is not in the HUSP-Tree, the algorithm constructs $\beta$'s SeqUtilList using I-Step or S-Step and creates $ND(\beta)$ under $ND(\alpha)$. If $ND(\beta)$ is already in the HUSP-Tree, the algorithm incrementally updates the tuples in SeqUtilList($\beta$) related to the new and oldest transactions as follows. Given the oldest transaction $S^{i-1}_2$ and the newest transaction $S^n_2$, according to Lemma 1 and Lemma 2, the SeqUtilList($\beta$) should be updated if it has a tuple whose SID is either $S_i$ or $S_i$'. These tuples (not all the tuples in SeqUtilList($\beta$)) are reconstructed by applying I-Step (if $\beta = \alpha \oplus \gamma$) or S-Step (if $\beta = \alpha \ominus \gamma$) on SeqUtilList($\alpha$) and ItemUtilLists($\gamma$). Then the algorithm updates TSWU of $\beta$ based on the updated SeqUtilList($\beta$). If TSWU of $\beta$ is less than the utility threshold, the algorithm removes $ND(\beta)$ and the sub-tree under $ND(\beta)$. Otherwise, if $\beta = \alpha \ominus \gamma$, the algorithm calls the procedure UpdateTree($ND(\beta)$) to update the sub-tree of $ND(\beta)$. If $\beta = \alpha \oplus \gamma$, the SPU of $\beta$ is updated using the updated SeqUtilList($\beta$). If SPU of $\beta$ 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($\{ab\}$) = \{ $\langle S_2, T_5, 23 \rangle$, $\langle S_3, T_5, 19 \rangle$ \}.

**3.3 HUSP Mining Phase**

HUSP mining phase is straightforward. After performing the update phase, HUSP-Tree maintains the information of the sequences in the current window. When users request 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 $\alpha$ in the current window. If the utility of $\alpha$ is no less than the minimum utility threshold, the algorithm outputs $\alpha$ as a HUSP.

### 4. EXPERIMENTS

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 [7]. BMS is obtained from SPMF [4] which contains sequences of clickstream data from an e-retailer. A synthetic dataset DS1: T32k1K1D100K is generated from the IBM data generator [1]. We follow previous studies [10] to generate internal and external utility of items for BMS and DS1. Table 1 shows characteristics of the datasets and parameter settings in the experiments. The $w$ column of Table 1 shows the default window size for each dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Seq</th>
<th>#Trans</th>
<th>#Items</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMS</td>
<td>6K</td>
<td>8K</td>
<td>3340</td>
<td>60K</td>
</tr>
<tr>
<td>DS1: T32k1K1D100K</td>
<td>400K</td>
<td>1000K</td>
<td>46,086</td>
<td>500K</td>
</tr>
</tbody>
</table>

Table 1: Details of parameter setting
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. 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.

5. CONCLUSIONS

We presented a novel framework for mining high utility sequential patterns over high velocity streaming data. We proposed an algorithm, 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 were proposed to dynamically maintain the essential information of potential HUSPs over data streams. We also defined a new over-estimated sequence utility measure named Suffix Utility (SFU), and used it to effectively prune the HUSP-Tree. Our experimental results show that our approach substantially outperforms the state-of-the-art algorithm in the number of generated potential HUSPs, run time and memory usage.

6. REFERENCES