Stream Sequential Pattern Mining with Precise Error Bounds

Luiz F. Mendes [1,2]   Bolin Ding [1]   Jiawei Han [1]

[1] University of Illinois at Urbana-Champaign   [2] Google Inc.

lmendes@google.com   {bding3, hanj}@uiuc.edu

Abstract

Sequential pattern mining is an interesting data mining problem with many real-world applications. This problem has been studied extensively in static databases. However, in recent years, emerging applications have introduced a new form of data called data stream. In a data stream, new elements are generated continuously. This poses additional constraints on the methods used for mining such data: memory usage is restricted, the infinitely flowing original dataset cannot be scanned multiple times, and current results should be available on demand.

This paper introduces two effective methods for mining sequential patterns from data streams: the SS-BE method and the SS-MB method. The proposed methods break the stream into batches and only process each batch once. The two methods use different pruning strategies that restrict the memory usage but can still guarantee that all true sequential patterns are output at the end of any batch. Both algorithms scale linearly in execution time as the number of sequences grows, making them effective methods for sequential pattern mining in data streams. The experimental results also show that our methods are very accurate in that only a small fraction of the patterns that are output are false positives. Even for these false positives, SS-BE guarantees that their true support is above a pre-defined threshold.

1 Introduction

The problem of mining sequential patterns from a large static database has been studied extensively by data mining researchers. This is an important problem with many real-world applications such as customer purchase behavior analysis, DNA sequence analysis, and analysis of scientific experiments, to name just a few [10]. However, in recent years, new applications such as network traffic analysis, web click-stream mining, and sensor data analysis have emerged and introduced a new kind of data referred to as *data stream* [8]. A data stream is an unbounded sequence in which new elements are generated continuously. Any mining method for data streams must consider the additional constraints that memory usage is restricted (meaning that we cannot store all the stream data in memory) and that we can only look at each stream component once, without performing any blocking operation.

Because of these constraints, traditional data mining methods developed for static databases cannot be applied for mining data streams. As a result, since the emergence of data streams, many new methods have been proposed for frequent pattern mining [5] [7], classification [1] [11], and clustering [2] [4] in data streams. However, there has been little work on finding effective methods for mining sequential patterns from data streams. There are many important real-world applications of this problem. For example, consider sequences generated by sensor networks, financial tickers, and web click-streams. In each of these cases, the data is generated as a stream, yet we would still like to find sequential patterns from the data for analysis.

In this paper we propose two effective methods for mining sequential patterns from data streams: *SS-BE* (Stream Sequence miner using Bounded Error) and *SS-MB* (Stream Sequence miner using Memory Bounds). The general framework for the two methods is similar, as both break up the stream into fixed-sized batches and perform sequential pattern mining on each batch. Both algorithms consist of storing the potentially frequent subsequences in a lexicographic tree structure. The main difference between our two methods involves the way they use pruning mechanisms to limit the amount of memory needed.

The *SS-BE* method ensures that all truly frequent sequences remain in the tree and are output when the user requests the list of frequent sequential patterns seen up to that point. It has another advantage of guaranteeing that the true support of the false positives is above some pre-defined threshold. To the best of our knowledge, *SS-BE* is the first method for solving the stream sequential pattern mining problem that guarantees no false negatives while also placing a bound on the error of the false positives.

The *SS-MB* method has the advantage that the maximum memory usage after processing any batch can be controlled explicitly. Although theoretically this method may only be able to guarantee that there are no false negatives after execution, it can also place a lower bound on the true count of

any sequence that is output.

The running time of each algorithm scales linearly as the number of sequences grows and the amount of memory used is bounded in both cases through the pruning strategies employed. These properties make both methods effective choices for stream sequential pattern mining.

2 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_j\}$ be a set of j items. A sequence is an ordered list of items from I denoted by $\langle s_1, s_2, \ldots, s_k \rangle$. A sequence $s = \langle a_1, a_2, \ldots, a_p \rangle$ is a subsequence of a sequence $s' = \langle b_1, b_2, \ldots, b_q \rangle$ if there exist integers $i_1 < i_2 < \cdots < i_p$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots, a_p = b_{i_p}$.

A *data stream* of sequences is an arbitrarily large list of sequences. A sequence s *contains* another sequence s' if s' is a subsequence of s. The *count* of a sequence s, denoted by $count(s)$, is defined as the number of sequences that contain s. The *support* of a sequence s, denoted by $supp(s)$, is defined as $count(s)$ divided by the total number of sequences seen. If $supp(s) \geq \sigma$, where σ is a user-supplied minimum support threshold, then we say that s is a frequent sequence, or a sequential pattern.

Our goal in solving this problem is to find all the sequential patterns in our data stream.

Example 1: Suppose the length of our data stream is only 3 sequences: $S_1 = \langle a, b, c \rangle$, $S_2 = \langle a, c \rangle$, and $S_3 = \langle b, c \rangle$. Let us assume we are given that $\sigma = 0.5$. The set of sequential patterns and their corresponding counts are as follows: $\langle a \rangle$:2, $\langle b \rangle$:2, $\langle c \rangle$:3, $\langle a, c \rangle$:2, and $\langle b, c \rangle$:2. □

While we stated above that the goal of the problem is to find all the sequential patterns in our data stream, this is not actually possible. Recall that in a data stream environment we cannot store all the sequences we have seen along with their respective exact counts. Thus, the goal is to find the set of *approximate* sequential patterns.

3 The SS-BE Method

The *SS-BE* method is inspired by the *Lossy Counting* algorithm [7], a method for solving a different, simpler problem: frequent *item* mining from data streams. Because it solves a different problem, the *SS-BE* method is significantly different from the *Lossy Counting* algorithm. However, both methods guarantee no false negatives and place the same bound on the true support of the false positives.

3.1 Algorithm

SS-BE uses a lexicographic tree T_0 to store the subsequences seen in the data stream. The tree nodes denote items and each path from the root to a node represents a sequence where the items are the nodes on that path in the root-leaf order. Thus, we can associate with each node n in our tree a corresponding sequence s_n and a path P_n from the root to the node. Each node will have a *count* value that will denote the count of the sequence corresponding to

it. The *count* of the root node will always be the number of sequences seen in the data stream. The tree T_0 will be stored in memory at all times.

For the *SS-BE* method, each node below the root in the tree T_0 will have four attributes:

- *item*: denotes an item represented by this node;
- *count*: denotes the count of the sequence corresponding to the path from the root to this node;
- *TID*: denotes the timestamp of the batch in our data stream that introduced this node to the tree;
- *batchCount*: indicates how many batches have contributed to the count of this node since the last time it was introduced to the tree.

The algorithm takes the following input values: (1) a stream of sequences $D = S_1, S_2, \ldots$; (2) minimum support threshold σ; (3) significance threshold ϵ where $0 \leq \epsilon < \sigma$; (4) batch length L; (5) batch support threshold α where $0 \leq \alpha \leq \epsilon$; and (6) pruning period δ.

The first step is to break the data stream into fixed-sized batches, where each batch contains L sequences. For each batch B_k that arrives, we apply PrefixSpan [10] to it, using support α. Each frequent sequence s_i that is found, having count c_i, is inserted to the tree T_0. If a path corresponding to this sequence does not exist in the tree, then one must be created, setting the $batchCount$ and $count$ values of the new nodes to 0 and the TID values to k. In either case, once the node corresponding to this sequence exists in T_0, we increment its $count$ by c_i and its $batchCount$ by 1.

Every time the number of batches seen is a multiple of the pruning period, δ, we prune the tree T_0. For each node in the tree, we define B to be the number of batches elapsed since the last pruning before it was inserted in the tree, and let $B' = B - batchCount$. If $count + B'(\lceil \alpha L \rceil - 1) \leq \epsilon BL$ for a given node, then we delete the entire subtree rooted at that node from the tree.

The above steps are repeated until the user requests the set of sequential patterns. At that point, we compute the number of sequences seen, $N = kL$, and output all sequences corresponding to nodes in the tree having $count \geq (\sigma - \epsilon)N$. This guarantees that all true sequential patterns are output and that any false positives have real support count at least $(\sigma - \epsilon)N$, as proven in the next subsection.

3.2 Correctness

Lemma 1. *In Algorithm 1, $U = count + B'(\lceil \alpha L \rceil - 1)$, which is computed for a node n during the pruning phase, is an upper bound of the true count over the last B batches of the sequence s corresponding to node n in the tree.*

Proof. For each batch B_i in the last B batches, U is increased by $\max\{count_{B_i}(s), \lceil \alpha L \rceil - 1\}$, where $count_{B_i}(s)$ is the true count of s in batch B_i. □

Theorem 1. *The algorithm produces no false negatives, and all false positives are guaranteed to have true support count of at least $(\sigma - \epsilon)N$, where N is the stream length.*
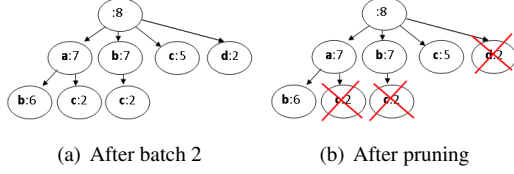
(a) After batch 2   (b) After pruning

Figure 1. States of T_0 in SS-BE

Proof. Suppose a sequence s is pruned n times, after batches $B_{f1}, B_{f2}, \ldots, B_{fn}$. Suppose the last pruning operation before the sequence was inserted each of these n times occurred at batches $B_{i1}, B_{i2}, \ldots, B_{in}$. Suppose $(B_{f1} - B_{i1}) = d_1, (B_{f2} - B_{i2}) = d_2, \ldots, (B_{fn} - B_{in}) = d_n$. Let $d_1 + d_2 + \cdots + d_n = p$. By definition of our pruning scheme and by Lemma 1, the true count of the sequence s in these p batches is less than or equal to $\sum_{i=1}^{n}(\epsilon d_i L) = \epsilon p L$. The only other source of undercounting in the remaining $(N/L - p)$ batches occurs when a sequence is not frequent in its batch and thus does not have its true count from that batch inserted to the tree. But in each such batch, the sequence has true count below $\alpha L \leq \epsilon L$. Thus, the maximum amount of undercounting here is $(N/L - p)\epsilon L$. Adding the two sources of undercounting gives $p\epsilon L + (N/L - p)\epsilon L = N\epsilon$. Because our algorithm outputs all sequences having *count* in the tree greater than or equal to $(\sigma - \epsilon)N$, we are guaranteed to get all the true sequential patterns. Furthermore, every node that is output is guaranteed to have true support count least $(\sigma - \epsilon)N$ because our algorithm does not perform any overcounting. $\square$

3.3   Example Execution

Suppose the batch length L is 4, the minimum support threshold σ is 0.75, the significance threshold ϵ is 0.5, the batch support threshold α is 0.4, and the pruning period δ is 2. Let batch B_1 be composed of sequences $\langle a, b, c \rangle, \langle a, c \rangle, \langle a, b \rangle$, and $\langle b, c \rangle$. Let batch B_2 be composed of sequences $\langle a, b, c, d \rangle, \langle c, a, b \rangle, \langle d, a, b \rangle$, and $\langle a, e, b \rangle$. Assume the stream length $N = 8$, so we only process these two batches and then output the sequential patterns found.

The algorithm begins by applying PrefixSpan [10] to the first batch with minimum support 0.4. The frequent sequences found, followed by their support counts, are: $\langle a \rangle$:3, $\langle b \rangle$:3, $\langle c \rangle$:3, $\langle a, b \rangle$:2, $\langle a, c \rangle$:2, and $\langle b, c \rangle$:2. Nodes corresponding to each of these sequences are inserted into the tree T_0 with the respective counts and a $batchCount$ of 1.

The algorithm then moves on to the next batch, applying PrefixSpan [10] to B_2, again with support 0.4. The frequent sequences found, followed by their support counts, are: $\langle a \rangle$:4, $\langle b \rangle$:4, $\langle c \rangle$:2, $\langle d \rangle$:2, and $\langle a, b \rangle$:4. This causes the nodes corresponding to sequences $\langle a \rangle, \langle b \rangle, \langle c \rangle$, and $\langle a, b \rangle$ to have their counts incremented by 4, 4, 2, and 4, respectively. In addition, each of these nodes has its $batchCount$ variable incremented to 2. A new node is created corresponding to the sequence $\langle d \rangle$, having *count* of 2 and $batchCount$ of 1. The state of the tree at this point is shown in Figure 1(a).

Because the pruning period is 2, we must now prune the tree. For each node, B is the number of batches elapsed since the last pruning before that node was inserted in the tree. In this case, the last pruning can be thought of as occurring at time 0. Thus, $B = 2$ for all nodes. For each node, $B' = B - batchCount$. In this case, some nodes have $B' = 0$, whereas others have $B' = 1$. According to the algorithm, we prune from the tree all nodes satisfying:

$$count + B'(\lceil \alpha L \rceil - 1) \leq B\epsilon L$$

In this case, that reduces to: $count + B' \leq 4$. The state of the tree after pruning is shown in Figure 1(b).

If the user requests the set of sequential patterns now, the algorithm outputs all sequences corresponding to nodes having count at least $(\sigma - \epsilon)N = (0.75 - 0.5) * 8 = 2$. The output sequences and counts are: $\langle a \rangle$:7, $\langle b \rangle$:7, $\langle c \rangle$:5, and $\langle a, b \rangle$:6. As guaranteed by the algorithm, there is no false negative. In this case, there is only one false positive: $\langle c \rangle$.

4   The SS-MB Method

The *SS-MB* method is inspired by the *Space-Saving* algorithm [9], which, like *Lossy Counting* [7], solves the simpler problem of frequent *item* mining from data streams.

4.1   Algorithm

Like *SS-BE*, the *SS-MB* method also uses a lexicographic tree T_0 to store the subsequences seen in the data stream. The only difference in the structure of the tree from that used in the previous method is that here each node below the root will have only three attributes:
- *item*: denotes an item represented by this node;
- *count*: denotes the count of the sequence corresponding to the path from the root to this node;
- *over_estimation*: denotes an upper bound on the over-estimation of the count of this node in the tree compared to the true count of the sequence corresponding to this node.

Like the *Space-Saving* algorithm, *SS-MB* limits the number of monitored sequences by ensuring that the tree T_0 will never have more than a fixed number of nodes, m, after processing any given batch.

The algorithm takes the following input values: (1) a stream of sequences $D = S_1, S_2, \ldots$; (2) minimum support threshold σ; (3) significance threshold ϵ where $0 \leq \epsilon < \sigma$; (4) batch length L; and (5) maximum number of nodes in the tree m. There will also be a variable min, initially set to 0, to keep track of the largest count of any node that has been removed from our tree.

The first step is to break the data stream into fixed-sized batches, where each batch contains L sequences. For each batch B_k that arrives, we apply PrefixSpan [10] to it, using support ϵ. Each frequent sequence s_i that is found, having count c_i, is inserted to the tree T_0. If a path corresponding to this sequence does not exist in the tree, then one must be created, setting the *over_estimation* and *count* values of

the new nodes to min. In either case, once the node corresponding to this sequence exists in the tree, we increment its $count$ by c_i.

After processing each batch, we check whether the number of nodes in the tree exceeds m. While this is true, we remove from the tree the node of minimum count. We then set min to equal the count of the last node removed.

The above steps are repeated until the user requests the set of sequential patterns. At that point, we compute the number of sequences seen, $N = kL$, and output all sequences corresponding to nodes in the tree having $count > (\sigma - \epsilon)N$. The sequences corresponding to nodes having $count - over\_estimation \geq \sigma N$ are guaranteed to be frequent. Furthermore, if $min \leq (\sigma - \epsilon)N$, then the algorithm guarantees that all true sequential patterns are output, as proven in the next subsection.

4.2 Correctness

For a sequence s, let $real\_count(s)$ denote the true support count of s in the data stream. Let $tree\_count(s)$ denote the $count$ value of the node corresponding to s in the tree T_0. Let $infinite\_count(s)$ denote what the value of the $count$ variable of the node corresponding to s in the tree T_0 would be if the tree had infinite capacity ($m = \infty$).

Lemma 2. *For every sequence s, $real\_count(s) - infinite\_count(s) < \epsilon N$, where N is the number of sequences seen in the data stream.*

Proof. The only source of undercounting if the tree has infinite capacity is from when a sequence is not frequent in a batch and thus does not get added to the tree. Suppose this occurs in B' batches. For each of these batches, the sequence has true count below ϵL. Thus, the total amount of undercounting is less than $B'\epsilon L \leq \epsilon N$. $\square$

Lemma 3. *Every sequence s appearing in the tree T_0 has $tree\_count(s) \geq infinite\_count(s)$.*

Proof. The only way $tree\_count(s)$ would become less than $infinite\_count(s)$ would be through pruning. However, if a sequence s is removed from the tree, then $min \geq tree\_count(s)$. The value of min never becomes smaller. If s is ever re-inserted to the tree, then its new value for $tree\_count(s)$ is incremented by the current value of min, which is at least the value of $tree\_count(s)$ when it was last removed. This means $tree\_count(s) \geq infinite\_count(s)$. $\square$

Lemma 4. *Any sequence s_i with $infinite\_count(s_i) > min$ must exist in the tree.*

Proof. Let us prove this theorem by contradiction: suppose there exists a sequence s_i with $infinite\_count(s_i) > min$ that does not exist in the tree. Because this sequence had to have been inserted to the tree at some point, we know that it has been removed since then. Suppose its count in the tree when it was last removed, $tree\_count(s_i)$, was c_f. By Lemma 4, we know that $c_f \geq infinite\_count(s_i)$ at

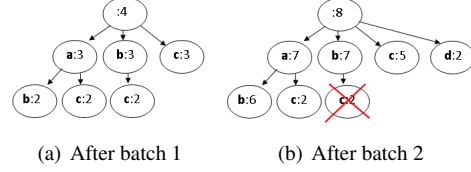(a) After batch 1   (b) After batch 2

Figure 2. States of T_0 in SS-MB

that point. Because s_i was removed, c_f was the minimum count value in the tree. The fact that the value of min never gets smaller means that $min \geq c_f$. Based on this, we know that $infinite\_count(s_i) \leq c_f \leq min$. This contradicts the original assumption, concluding our proof. $\square$

Theorem 2. *If at the end of the algorithm $min \leq (\sigma - \epsilon)N$, where N is the stream length, then all true sequential patterns are output.*

Proof. Any true sequential pattern s' has $real\_count(s') \geq \sigma N$. By Lemma 3, $infinite\_count(s') > (\sigma - \epsilon)N$. If $min \leq (\sigma - \epsilon)N$, then by Lemma 5 we know that s' must occur in our tree. Lemma 4 shows that $tree\_count(s') \geq infinite\_count(s')$, which means that $tree\_count(s') > (\sigma - \epsilon)N$ and s' is output at the end. $\square$

4.3 Example Execution

Suppose the batch length L is 4, the minimum support threshold σ is 0.75, the significance threshold ϵ is 0.5, and the maximum number of nodes allowed in the tree after processing any given batch is 7. Let batch B_1 be composed of sequences $\langle a, b, c \rangle, \langle a, c \rangle, \langle a, b \rangle$, and $\langle b, c \rangle$. Let batch B_2 be composed of sequences $\langle a, b, c, d \rangle, \langle c, a, b \rangle, \langle d, a, b \rangle$, and $\langle a, e, b \rangle$. Let the stream length $N = 8$, so we only process these two batches and output the sequential patterns found.

The algorithm begins by applying PrefixSpan [10] to the first batch with minimum support $\epsilon = 0.5$. The frequent sequences found are: $\langle a \rangle$:3, $\langle b \rangle$:3, $\langle c \rangle$:3, $\langle a, b \rangle$:2, $\langle a, c \rangle$:2, and $\langle b, c \rangle$:2. Nodes corresponding to each of these sequences are inserted into the tree T_0 with the respective counts and an $over\_estimation$ of 0. The state of the tree after processing this batch is shown in Figure 2(a).

The algorithm then moves on to the next batch, applying PrefixSpan [10] to B_2, again with support 0.5. The frequent sequences found, followed by their support counts, are: $\langle a \rangle$:4, $\langle b \rangle$:4, $\langle c \rangle$:2, $\langle d \rangle$:2, and $\langle a, b \rangle$:4. So the nodes corresponding to sequences $\langle a \rangle, \langle b \rangle, \langle c \rangle$, and $\langle a, b \rangle$ have their counts incremented by 4, 4, 2, and 4, respectively. A new node is created corresponding to the sequence $\langle d \rangle$, having $count$ as $2 + min = 2$ and $over\_estimation$ as $min = 0$.

Because there are now 8 nodes in the tree and the maximum is 7, we must remove the sequence having minimum count from the tree. Breaking ties arbitrarily, the node corresponding to the sequence $\langle b, c \rangle$ is removed. The variable min is set to this sequence's count before being deleted, 2. The current state of the tree is shown in Figure 2(b).

Suppose now the user requests the set of sequential patterns. The algorithm outputs all sequences corresponding to

nodes having count above $(\sigma - \epsilon)N = (0.75 - 0.5) * 8 = 2$. The output sequences and counts are: $\langle a \rangle$:7, $\langle b \rangle$:7, $\langle c \rangle$:5, and $\langle a, b \rangle$:6. Because $min = 2 \leq (\sigma - \epsilon)N = 2$, the algorithm *guarantees* that there are no false negatives. In this case, there is only one false positive: $\langle c \rangle$.

5   Experimental Results

We implemented both algorithms *SS-BE* and *SS-MB* in C++ on a PC with an Intel Core2 Quad CPU at 2.4 GHz and 4 GB of memory. The operating system was Linux Fedora version 2.6.9-67. All experiments were performed on synthetic sequence data obtained through a sequential data generator in the IlliMine system package, which can be retrieved from the following website: http://illimine.cs.uiuc.edu/.

Exp-1 (Varying the number of sequences): For the following experiments, the number of distinct items in the input data was 100, the average sequence length was 10, the minimum support threshold (σ) was 0.01, the significance threshold (ϵ) was 0.00999, and the batch length (L) was 50,000. These values were used for both methods. The *SS-BE* method had a batch support threshold (α) of 0.00995, and a prune period (δ) of 4 batches. For *SS-MB*, the value for the maximum number of nodes in the tree (m) was chosen to be the smallest possible value such that the algorithm still guaranteed that all true sequential patterns were output. On average, the ratio of m to the number of true sequential patterns was 1.115, with 1.208 being the highest value.

The running time, maximum memory usage, and accuracy, defined as the percentage of the output sequences that are actually frequent, for our two methods are shown in Figures 3(a), 3(b), and 3(c), respectively. Notice that the running time of the proposed algorithms scales linearly as the number of sequences grows, and that the maximum memory usage is very small. This feature is not only desirable, but also necessary for mining sequential patterns from data streams, where the number of sequences is unbounded.

Exp-2 (Varying the average sequence length): For the following experiments, the number of distinct items in the input data was 100, the total number of sequences was 100,000, the minimum support threshold (σ) was 0.01, the significance threshold (ϵ) was 0.0099, and the batch length (L) was 50,000. These values were used for both methods. The *SS-BE* method had a batch support threshold (α) of 0.0095, and a prune period (δ) of 1 batch. For *SS-MB*, the value for the maximum number of nodes in the tree (m) was chosen to be the smallest possible value such that the algorithm still guaranteed that all true sequential patterns were output. On average, the ratio of m to the number of true sequential patterns was 1.054, with 1.082 the highest value.

In this experiment, we also compare the performance of our methods with that of a naive algorithm, which behaves as follows: for each sequence that arrives in the data stream,

the algorithm finds all of its possible subsequences and inserts each one into a global lexicographic tree T_0, like the one used by our methods, with a count of 1. There is also a pruning period δ and a significance threshold ϵ. After every δ sequences, the algorithm prunes the tree in a similar manner as our *SS-BE* method. Similarly, at the end all sequences having count at least $(\sigma - \epsilon)N$ are output, where σ is the minimum support threshold and N is the stream length. This algorithm also guarantees that all true sequential patterns are output and that the false positives have true support at least $(\sigma - \epsilon)N$. We set ϵ to 0.0099 and δ to 20.

The execution time, maximum memory usage, and accuracy, defined as in the previous experiment, for all three methods are shown in Figures 3(d), 3(e), and 3(f), respectively, as functions of the average sequence length. Notice that both of our proposed methods scale extremely well as the average sequence length increases. In contrast, the weakness of the naive method is exposed, i.e., the number of sequences it generates grows exponentially with the average sequence length. In fact, when the average sequence length is 10, as in experiment 1, this naive method cannot complete in a reasonable amount of time. For this reason, we did not include it in experiment 1.

6   Discussion

Comparing Methods: The main advantage of the *SS-BE* method is that it always *guarantees* that there are no false negatives, while also placing a bound on the support of the false positives. However, since there is no precise relationship between the significance threshold parameter ϵ and the maximum memory usage, the user may pick a value for ϵ that forces the algorithm to remove nodes from the tree to free up space before all the available memory is used. Another possibility is that the user may pick a value for ϵ that is too small and causes the system to run out of memory as a result of insufficient pruning. Neither of these can happen in the *SS-MB* method because the user can set the parameter for the maximum number of nodes in the tree based on the amount of memory available. Thus, by exploiting all of the available memory in the system, the *SS-MB* method may be able to achieve greater accuracy than the *SS-BE* method.

Related Work: Several algorithms have been proposed for stream sequential pattern mining, but, unlike *SS-BE*, none of them places a precise bound on the approximation.

a) SMDS Algorithm: One of the proposed methods is the Sequence Mining in Data Streams (SMDS) algorithm [8], which uses sequence alignment techniques for mining approximate sequential patterns in data streams. Unlike our *SS-BE* method, this method is unable to guarantee a precise level of "closeness" to the true set of sequential patterns.

b) SSM Algorithm: The Sequential Stream Mining (SSM) algorithm [3] breaks the data stream into variable-sized batches. It keeps track at all times of the frequency counts
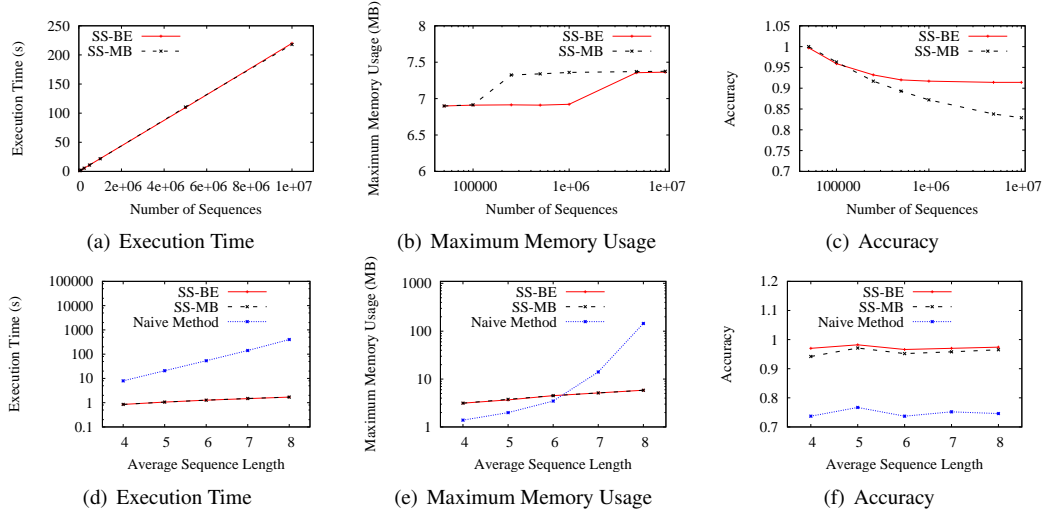
| (a) Execution Time | (b) Maximum Memory Usage | (c) Accuracy |
| (d) Execution Time | (e) Maximum Memory Usage | (f) Accuracy |

Figure 3. Experimental Results

of single items using a D-List structure. For each batch that arrives, it performs sequential pattern mining on that batch using an algorithm for static databases and stores the results in a compact tree. The paper claims that all true frequent sequences are found, but this may not be the case. The D-List only stores items whose support count are greater than or equal to the total number of sequences seen so far multiplied by the percentage maximum tolerance error, e. Before applying the sequential pattern miner on each batch, items not in the D-List are removed from the batch, causing the algorithm to possibly miss some true frequent sequences.

c) GraSeq Algorithm: The GraSeq algorithm [6] generates all length-1 and length-2 subsequences of each sequence in the stream, and keeps track of the support counts of these sequences. When the user requests the set of sequential patterns, the algorithm outputs all sequences having all length-1 and length-2 subsequences being frequent. The main problem of this algorithm is that it will often generate a large number of false positives. Furthermore, these false positives may not even occur in the stream at all.

7 Conclusions

We have introduced two algorithms that can be used for mining sequential patterns when the number of input sequences is unbounded, as in a data stream. The *SS-BE* method ensures that there are no false negatives, while also guaranteeing that the true support of the false positives is above some pre-defined threshold. The *SS-MB* method is only guaranteed to output all true sequential patterns if at the end of the algorithm $min \le (\sigma - \epsilon)N$, where N is the total number of sequences seen in the stream. Its main advantage is that the amount of memory used at the end of processing any batch can be controlled explicitly.

The running time of each algorithm scales linearly as the number of sequences grows, and the maximum memory us-

age is restricted in both cases through the pruning strategies adopted. In addition, our experiments show that both methods produce a very small number of false positives. Based on these properties, our proposed methods are effective solutions to the stream sequential pattern mining problem.

References

[1] C. Aggarwal, J. Han, J. Wang, and P. S. Yu. On demand classification of data streams. In *KDD'04*.

[2] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB'03*.

[3] C. I. Ezeife and M. Monwar. SSM: A frequent sequential data stream patterns miner. In *CIDM'07*.

[4] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *FOCS'00*.

[5] R. M. Karp, C. H. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in streams and bags. In *ACM Trans. Database Systems*, pages 51–55, March 2003.

[6] H. Li and H. Chen. *GraSeq* : A novel approximate mining approach of sequential patterns over data stream. In *ADMA'07*.

[7] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB'02*.

[8] A. Marascu and F. Masseglia. Mining sequential patterns from temporal streaming data. In *MSTD'05*.

[9] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT'05*.

[10] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE'01*.

[11] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *KDD'03*.