Mining Sequential Patterns Across Data Streams

Gong Chen, Xindong Wu, and Xingquan Zhu

Department of Computer Science, University of Vermont,
Burlington VT 05405, USA
{gchen,xwu,xqzhu}@cs.uvm.edu

Abstract. There are extensive endeavors toward mining frequent items or itemsets in a single data stream, but rare efforts have been made to explore sequential patterns among literals in different data streams. In this paper, we define a challenging problem of mining frequent sequential patterns across multiple data streams. We propose an efficient algorithm MILE[1] to manage the mining process. The proposed algorithm recursively utilizes the knowledge of existing patterns to make new patterns' mining fast. We also apply a state-of-the-art sequential pattern mining algorithm PrefixSpan which was designed for transaction databases to solve our problem. Extensive empirical results show that MILE is significantly faster than PrefixSpan. One unique feature of our algorithm is when some prior knowledge of the data distribution in the data streams is available, it can be incorporated into the mining process to further improve the performance of MILE. As MILE consumes more memory than PrefixSpan, we also propose a solution to balance the memory usage and time efficiency in memory limited environments.

1 Introduction

Many real-world applications involve data streams. Examples include data flows in medical ICU (Intensive Care Units), network traffic data, stock exchange rates, and Web interface actions. Discovering structures of interest in multiple data streams is an important problem, because such structures are useful for further analysis. For example, the knowledge from data streams in ICU (such as the oxygen saturation, chest volume and heart rate) may indicate or predicate the state of a patient's situation, and an intelligent agent with the ability to discover knowledge in the data from multiple sensors can automatically acquire and update its environment model [11].

In this paper, we assume that real-valued data has been discretized into tokens and we deal with categorical data only. A token stands for an event at a certain abstraction level, for example, a steady heart rate or a rising stock price. One discretization method proposed by Gautam et al. [3] is to cluster subsequences in a sliding window at first and then assign the cluster identifiers to these subsequences. In this paper, we are interested in knowledge in the form of frequent sequential patterns across data streams. Such a pattern can look like

[1] MIning from muLtiple strEams

“the price of Sun stock and the price of IBM stock go up at the same time, and within two days Microsoft stock’s price goes down and one day later Intel stock’s fall as well.” Mining such a sequential pattern across multiple data streams (e.g., the stock prices of different companies) is a more challenging task than previous studies of frequent itemset mining and is also distinct from sequential pattern mining from supermarket basket data. The challenges come from the following three aspects. (1) When it comes to sequential pattern mining, there are too many candidates to be dealt with in multiple streams. A single data stream with 10 distinct tokens can result in $\sum_{i=1}^{10} P_{10}^{i}$ possible patterns. One can imagine how large this number could be if we increased the number of streams to 10 as well. (2) In the data stream scenario, the occurrence of a sequential pattern can complicate the mining procedure too, even if the order of the pattern literals is the same. That is, a matching instance of a pattern can occur with noisy tokens at different time points involved, which makes it hard to count the numbers of patterns’ occurrences. (3) Steaming data never ends and always arrives in a continuous manner. It can easily reach a larger number of patterns for the data at hand. We must provide a practical and efficient solution to find out frequent patterns which make sense to real-world users. We start our work to deal with static streams or a period history of data streams (for example, one day or one hour) like [3] and [11]. One future direction is to explore our work to handle dynamic streams.

The contributions of this paper are as follows.

- We define a challenging problem of mining sequential patterns across data streams.
- We design an efficient algorithm MILE to solve this problem.
- One unique feature of MILE is that it can incorporate prior knowledge of the data distribution in the streams into the mining process to further improve the efficiency when the knowledge is available.
- We apply a state-of-the-art sequential pattern mining algorithm PrefixSpan (which was designed for transaction databases) to solve our problem. Extensive empirical results show that MILE is significantly faster than PrefixSpan.
- We also propose a solution to balance the memory usage and time efficiency in memory limited environments.

The remainder of the paper is organized as follows. In Section 2 we review related work and discuss the difference between our problem and previous studies. The problem is formally defined in Section 3. In Section 4, we describe the design of our MILE algorithm. In Section 5 empirical comparative results are presented. Finally, we conclude our work and discuss some future directions in Section 6.

2 Related Work

Sequential pattern mining in transaction databases has been well studied in [1], [13], [16] and [12]. The most recent report in [12] shows that the PrefixSpan algorithm is significantly faster than other sequential pattern mining algorithms. The merits of PrefixSpan come from the fact that it recursively projects the original dataset into smaller and smaller subsets, from which patterns can be progressively mined out. PrefixSpan does not need to generate candidate patterns and identify their occurrences but grows patterns as long as the current item is frequent in the projected dataset. This property makes PrefixSpan extremely efficient. However, when PrefixSpan recursively projects the original dataset into overlapping subsets, it is very likely that PrefixSpan scans the same part of data again and again. This disadvantage, however, can be overcome by our proposed approach, namely suffix appending (embedded in MILE). In the next section we will use PrefixSpan to solve our problem, and will also conduct extensive comparisons between PrefixSpan and MILE in the context of multiple data streams in Section 5. One can see the semantic difference between sequential pattern mining in transaction databases and data streams. For example, there might be no transactions, customer-ids and items purchased in data streams. However, if we assume that we deal with a period history of data streams and treat each time window of data as one customer's transactions (and each time point of the data as one transaction), then the problem of sequential pattern mining in data streams can be generalized as sequential pattern mining in transaction databases and any sequential pattern mining algorithm can be used to solve the problem. That is why we can employ PrefixSpan to solve our problem and do fair comparisons between PrefixSpan and MILE. It is natural that the suffix appending approach we will propose in MILE can also be adopted for sequential pattern mining in transaction databases though MILE is designed to handle sequential pattern mining in data streams.

Mannila et al. [10] dealt with mining frequent episodes in a sequence of events while we are dealing with multiple sequences of events. There are also extensive studies on mining frequent items or itemsets which do not have sequential (temporal) order among items from data streams. Manku et al. [9] computed approximate frequency information for items or itemsets over data streams with provably small memory footprints. Charikar et al. [2] introduced a 1-pass algorithm to estimate the most frequent item in a data stream. Giannella et al. [5] developed an algorithm based on the frequent-pattern tree to find frequent itemsets from data streams. Jin et al. [7] maintained frequent items over a data stream with a small bounded memory in a dynamic environment where insertion and deletion of items are allowed. Das et al. [3] considered the problem of rule discovery from discretized data streams. A rule here is in the form of the occurrence of event A indicating the occurrence of event B within time T. We can treat this type of causal rule as a simplified sequential pattern of two events while a pattern in our problem involves an arbitrary number of events which make the problem much more complicated.

The most relevant work to our problem in this paper was introduced by Oates et al. [11]. They tried to search rules in the form of x indicating y within time δ where x is a set of events within a window and y is also a set of events within the window. However, in each of x and y, the order in which events happen is fixed. For example, an x is like: after event A happens, *exactly* two time points later event B happens, and *exactly* three time points later event C happens. In our problem definition in Section 3, after event A happens, *within* two time points event B happens, and *within* three time points later event C happens. This loose temporal order makes our mining problem more challenging. Also, the rule form in [11] is only a special case of our patterns. Their search for rules in the restricted form is unable to find our patterns.

Zhu et al. [17] found high correlations between all pairs of data streams based on Discrete Fourier Transforms. Yi et al. [15] studied an entire set of sequences as a whole to predict for the last "current" values based on a multi-variate linear regression. These two studies tried to build global models between two entire streams or among the entire set of streams while our focus in this paper is on mining local patterns across data streams. By local, we mean that we are interested in the patterns of events in different streams that happen within a time window in a loose temporal order. Another line of related research is to efficiently identify a pattern out of a set of patterns when that pattern appears in the data streams. Gao et al. [4] proposed Fast Fourier Transforms based optimization techniques for this pattern evaluation process. Keogh et al. [8] attacked fast pattern matching with a probabilistic approach. Wang et al. [14] monitored the occurrences of patterns in the form of conjunctive correlations among multiple data streams. We can see that before the pattern matching process the users need data mining algorithms to discover interesting patterns, which is the topic of this paper.

3 Problem Statement

A stream of categorical data is an infinite sequence of literals. At each time point n, however, the stream of categorical data takes the form of a finite sequence, assuming the last literal is the one that arrived at the time point n. We adopt the notations as follows. Data entries, which are called *tokens*, in a stream of categorical data are in the triple form of (*streamID, timePoint, value*). Each stream consists of all tokens with the same streamID.

Each stream has a value available at every time step, for example, every second. We call the step index the *timePoint*. For example, if the time-step unit is a second and the current timePoint is i, after one second, all the streams will have new values at timePoint $i + 1$. Let s_i denote the value of stream s at timePoint i, $s_{i...j}$ denote the subsequence of stream s from timePoint i through j inclusive, and s^j denote the stream with streamID j. We use n to denote the latest timePoint. We also assume that all of the tokens occurring at a given timePoint in the streams were recorded synchronously.

Assuming a finite amount of space for frequent patterns, we only consider patterns that span no more than a constant number of consecutive time steps.

We allow a pattern to span at most w time steps, i.e., a time window of width w for each pattern. We are interested in a pattern if the number of its occurrences is more than a threshold *minSup*. *minSup* and w are both user-specified parameters. Consider the following example with 3 data streams and 12 time points. If *minSup*=3 and w=4, we can find the pattern {(33 22 *)(* * 11)}. We put pattern literals at the same time point in parentheses and put all pattern literals in {}.

	1	2	3	4	5	6	7	8	9	10	11	12
s^3	33	.	.	.	.	33	.	.	.	33	.	.
s^2	22	.	.	.	.	22	.	.	.	22	.	.
s^1	.	.	11	.	.	.	11	.	.	.	.	11

We call pattern literals at the same time point (p^m p^{m-1} ...p^j... p^1) an **intra-pattern** where m is the number of data streams. p^j is a token[2] or a wild card which can match any token. A **pattern** consists of intra-patterns. The maximum number of intra-patterns in a pattern cannot be greater than w. Any two intra-patterns in a pattern cannot occur at the same time point in the data streams. Intra-patterns in a pattern have a *loose temporal order* between them. In the above example, the pattern {(33 22 *)(* * 11)} requires intra-pattern (* * 11) appear after intra-pattern (33 22 *). But (* * 11) can either happen immediately after (33 22 *) or several time points later within the same window.

Given

- a set of streams S={s^1, s^2, ..., s^j, ..., s^m} where s^j=$s^j_1 s^j_2 . .. s^j_i ... s^j_{n-1} s^j_n$ is a stream of categorical data, and s^j_i is shorthand for a token (j, i, s^j_i) where $s^j_i \in V^j$ ($\bigcup_j V^j = \sum$) which is the set of categorical values for stream s^j;
- the width of time window w, and
- the threshold value *minSup*,

a complete set of patterns satisfying the following conditions is discovered at timePoint n:

- each pattern is in the form of $\{(p^m_i p^{m-1}_i ... p^j_i ... p^1_i) \| i \in [0, w-1]\}$ where p^j_i is either a token in V^j or a wild card *;
- for any two tokens from different intra-patterns in a pattern, $p^{j_1}_{i_{k_1}}$ and $p^{j_2}_{i_{k_2}}$ ($1 \leq j_1 \leq m$, $1 \leq j_2 \leq m$, $i_{k_1} < i_{k_2}$), $s^{j_1}_{t+i'_{k_1}}$ and $s^{j_2}_{t+i'_{k_2}}$, the corresponding matching tokens ($s^{j_1}_{t+i'_{k_1}} = p^{j_1}_{i_{k_1}}$ and $s^{j_2}_{t+i'_{k_2}} = p^{j_2}_{i_{k_2}}$) at timePoint t, should preserve the **temporal** condition $i'_{k_1} < i'_{k_2}$; and
- the number of each pattern's occurrences in S is greater than *minSup*.

[2] If not explicitly explained as a triple (*streamID, timePoint, value*), a token means the value of that token.

To be concise, we ignore wild cards in a pattern description. For example, we use {(33 22)(11)} instead of {(33 22 *)(* * 11)}. Since we can always encode tokens in such a way that different streams have different sets of tokens, this representation causes no confusion. For example, we encode tokens in s^1 starting with 1, tokens in s^2 starting with 2 and so on. In the above example, 11 can only appear in s^1 so that {(11)} contains the same position information as {(* * 11)}.

For an arbitrary pattern P=$\alpha\tilde{t}\beta$ where α and β are subpatterns of P and $\tilde{t}$ is a token in P, we define $suffix(\tilde{t})$=β, and $prefix(\tilde{t})$=α. $\tilde{t}$ can occur in many patterns which have α as a prefix. We define $suffixes(\tilde{t})$=$\cup suffix(\tilde{t})$. Note that when we talk about $suffixes(\tilde{t})$, these suffixes should share the same prefix although we may not explicitly show it. For example, assuming two patterns {(33 20 10)(22 15)(32 21 11)} and {(33 20 10)(22 16)(34 25 11)}, $suffixes$(22)={(_ 15)(32 21 11), (_ 16)(34 25 11)} for the shared prefix (33 20 10). Assuming we have $suffixes(\tilde{t}_1)$, $suffixes(\tilde{t}_2)$,..., $suffixes(\tilde{t}_n)$ for the shared prefix α, we define $suffixesSet(t)$ where t is the last token of α in the form of {$\tilde{t}_1$:$suffixes(\tilde{t}_1)$; $\tilde{t}_2$: $suffixes(\tilde{t}_2)$;...; $\tilde{t}_n$:$suffixes(\tilde{t}_n)$}. Again, when we mention $suffixesSet(t)$, these suffixes should share some prefix α. For example, if we have two more patterns {(33 20 10)(21 18)(32 27 11)} and {(33 20 10)(21 19)(34 25 11)}, $suffixesSet$(10)={21:{(_ 18)(32 27 11), (_ 19)(34 25 11)}; 22:{(_ 15)(32 21 11), (_ 16)(34 25 11)}} for the prefix (33 20 10).

4 Algorithm Description

4.1 Description of PrefixSpan

We now use an example to explain how we can apply the general idea of PrefixSpan (PseudoProjection) [12] to solve our problem. First we outline the basic steps of PrefixSpan in our multi-stream context.

1. Scan data streams to locate tokens whose frequency is greater than minSup, and output them (each of which is a frequent pattern of a single value). If no frequent token exists, return.

2. For each pattern a, from each of its ending locations (the time point when the last token occurs) scan data streams at the same window to locate token b whose frequency is greater than *minSup*; append b to a; output ab; let a = ab, and goto step 2. If no frequent token exists, return.

Now let us apply PrefixSpan on the following example which has 3 data streams and 11 time points . w = 3 and *minSup* = 2. According to the parameters, we have 4 windows of data (the last window has only two time points of data), we want to find every sequential pattern that appears in at least 3 windows.

	1	2	3	4	5	6	7	8	9	10	11
s^3	33	32	39	33	31	38	33	30	35	36	37
s^2	21	22	23	24	22	25	26	22	27	28	29
s^1	10	12	11	13	14	11	15	16	11	17	18

Scan data once, 11, 22 and 33 are found to be frequent patterns of a single value. {(11)}, {(22)} and {(33)} are output. Scan data after[3] {(11)}, no frequent token is found since there is no data after {(11)} at the same window. Scan data after {(22)}, 11 is found to be frequent so {(22)(11)} is output. Scan data after {(22)(11)}, no frequent token is found. Scan data after {(33)}, 22 is found to be frequent so {(33)(22)} is output. Scan data after {(33)(22)}, 11 is found to be frequent so {(33)(22)(11)} is output. Scan data after {(33)(22)(11)}, no frequent token is found.

In the above example, PrefixSpan mines patterns with 11 as prefix first, then patterns with 22 as prefix and finally patterns with 33 as prefix. A prefix is growing gradually till it cannot grow due to infrequency. For example, patterns with 22 as prefix are mined in the order:{(22)}→{(22)(11)}; patterns with 33 as prefix are mined in the order: {(33)}→{(33)(22)}→{(33)(22)(11)}. Each time the prefix grows by one token.

4.2 Description of MILE

From the above example, we can see that when PrefixSpan mines {(33)(22)}, {(22)(11)} has been mined out at the previous stage as a pattern with 22 as prefix. Can we append this pattern directly to {(33)(22)} to form the pattern {(33)(22)(11)} without scanning the data after {(33)(22)}? In more general cases, can we append some mined patterns with b as prefix to pattern cb to form all the patterns with c as prefix without scanning the data after cb? If this is possible, we can avoid scanning data over and again and speed up the mining process. In the above example, the data after 22 has been scanned twice: once to mine patterns with 22 as prefix, and once to mine patterns with (33)(22) as prefix. Another advantage is when a very long pattern β with b as prefix is mined out and β also appears after cb, we will get long patterns with cb as prefix directly by appending β to c rather than recursively scanning the data after cb. That is, we want to let a prefix grow to the point that it cannot grow, and then get patterns starting with that prefix directly. How can we recursively utilize the knowledge from mined patterns to speed up the mining process? We describe below our algorithm MILE to manage this process efficiently.

We explain the mining process of MILE with a part of a pattern tree in Figure 2. One concatenation of tokens on edges from the root to any node forms a pattern. For example, {11} is a pattern and so are {11 44}, {11 44 β_1} and {33 22 11 55 44 β_2}. Here we can ignore the parentheses in patterns to understand the main idea of MILE smoothly. Due to limited space, we use β_i to denote a

[3] At the same time point, a token in the stream with a lower streamID is after a token in the stream with a higher streamID; and at different time points, a token at a later time point is after a token at an earlier time point.

```
MILE(){
1  token t = ();
2  t.endLoc←start time points of every window;
3  suffixesSet(t) = ();
4  index idx = ();
5  pattern set←PrefixExtend(t, suffixesSet(t), idx);
}

PrefixExtend(token t, suffixesSet s, index idx){
1  index nIdx = ();
2  suffixesSet(t) = ();
3  for e in t.endLoc
4    /**scanning process**/
5    scan from e to the end of window starting at e,
     register locations for every token t̃ at t̃.endLoc,
     update the frequency for t̃ at t̃.freq;
6  for token t̃ in ∑ and if(t̃.freq>minSup)
7    if(suffixes(t̃) in s)
8      suffixesSet(t)←SuffixAppend(t̃, suffixes(t̃), idx);
9    else
10     suffixesSet(t)←PrefixExtend(t̃, suffixesSet(t),
       nIdx);
11   suffixes(t)←append t̃ to (_);
12   suffixes(t)←append suffixes(t̃) in suffixesSet(t)
     to (_ t̃);
13 return suffixes(t);
}

SuffixAppend(token t̃, suffixes s_t̃, index idx){
1  if(idx has no idx_t̃ for s_t̃)
2    /**building index**/
3    idx←build idx_t̃ for s_t̃ with information in s_t̃;
4  /**hitting process**/
5  Use every e in t̃.endLoc to hit idx_t̃,
   update frequency for a hitted suffix in s_t̃,
   register the hitted location for a hitted suffix;
6  /**choosing the desired suffixes**/
7  suffixes(t̃)←suffixes in s_t̃ whose frequency>minSup;
8  return suffixes(t̃);
}
```

Fig. 1: Pseudo code for MILE

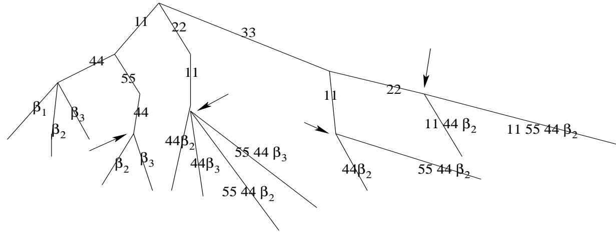

Fig. 2: Part of a pattern tree showing the mining process of MILE

suffix of a pattern which contains tokens on the corresponding edge. Similarly, we use 55 44 β_i to label the edge which denotes the concatenation of tokens 55, 44 and the suffix β_i. From the description of PrefixSpan, we can see that it performs a depth-first search along this pattern tree. It mines patterns in such an order: $\{11\}\rightarrow\{11\ 44\}\rightarrow\{11\ 44\ \beta_1\}\rightarrow\{11\ 44\ \beta_2\}\rightarrow\{11\ 44\ \beta_3\}\rightarrow\{11\ 55\ 44\ \beta_2\}\rightarrow\{11\ 55\ 44\ \beta_3\}$ and $\{22\}\rightarrow\{22\ 11\}\rightarrow$...and $\{33\}\rightarrow...\rightarrow\{33\ 22\ 11\ 55\ 44\ \beta_2\}$. MILE uses PrefixExtend to perform a similar process. But when it comes to $\{11\ 55\ 44\}$, it finds that $suffixes(44)$ for prefix 11 has been mined, so it calls SuffixAppend to select the desired suffixes (which will be explained in the next paragraph) from $suffixes(44)$ and append them directly to $\{11\ 55\ 44\}$ instead of performing a depth-first search to scan data in PrefixSpan. Similarly, when it comes to $\{22\ 11\}$, it finds that $suffixes(11)$ for prefix $\{\}$ (actually these are all patterns with 11 as prefix) has already been discovered so it calls SuffixAppend to select the desired suffixes from $suffixes(11)$ and append them to $\{22\ 11\}$. We use arrows to mark each place where SuffixAppend occurs in the pattern tree. From Figure 2, we can see that SuffixAppend is embedded in the mining process and avoids costly depth-first search (for redundant data scanning) so it speeds up the mining process significantly.

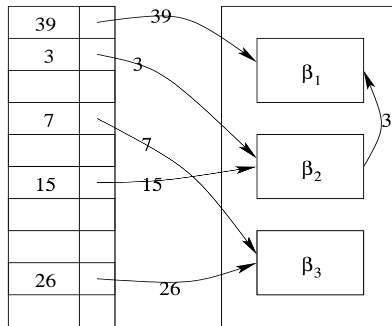

Fig. 3: Hash index of $suffixes(44)$ for prefix 11

Now we describe the selection process of SuffixAppend. It has three steps which are commented in the pseudo code in Figure 1: building index (optional), hitting process and choosing the desired suffixes. Now we use one part of the pattern tree in Figure 2 to show how these three steps work. Assuming MILE is currently running at point {11 55 44} with ending locations (time points when 44 in this pattern occurs in the data streams) (3, 7, 15, 26) and finds that $suffixes(44)$ for prefix 11 has been mined, it calls SuffixAppend. Ending locations are collected in the scanning process commented in the pseudo code of PrefixExtend.

Assume *minSup*=1 and start locations (time points when 44 in the corresponding suffixes occurs) of suffixes in $suffixes(44)$ for prefix 11 are as follows: (_ β_1): (3, 39); (_ β_2): (3, 15); (_ β_3): (7, 26). In this case, no index has been built for $suffixes(44)$ so the building process is started. To speed up the hitting process at a later stage, we use a hash table indexed by start locations of suffixes in $suffixes(44)$. Scan suffixes $suffixes(44)$ and their start locations once to insert each suffix to the corresponding bucket according to their start locations. Since (_ β_1) and (_ β_2) share the same start location 3, put them into a linked list indexed by 3. The resulting hash table is shown in Figure 3.

Now the hitting process begins. Every ending location of {11 55 44} is hashed into the hash table. When 3 is hashed, the frequencies of (_ β_1) and (_ β_2) are increased by 1. When 7 is hashed, the frequency of (_ β_3) is increased by 1. After all ending locations have been hashed, the choosing process will store every suffix whose frequency is greater than *minSup* in $suffixes(44)$ for prefix {11 55} for possible future appending. Also, the selected suffixes will be appended to prefix {11 55 44} in PrefixExtend. In this case, (_ β_2) and (_ β_3) are selected for appending to prefix {11 55 44} which also can be seen from Figure 2. Note that the constructed index for $suffixes(44)$ with 11 as prefix is stored for future use to avoid a repeated building process. For example, if we have a pattern {11 66 44}, then this index will be used again for appending suffixes to that pattern. This index will be dropped when all patterns with {11} as prefix are discovered. At this point, the readers might think that if no suffixes can be appended, this building process will be pure overhead. Actually, this is not true. If no suffixes can be appended, we only need to hash ending locations of a prefix to decide whether there is any suffix to be appended if we have this index in hand. Otherwise, we need to scan data during the scanning process in PrefixExtend to make the decision. In the case of relatively small numbers of ending locations, suffixes and their start locations, and a relatively large amount of data to be scanned, this indexing can still speed up the mining process which will be demonstrated in the experimental results.

4.3 Techniques

In this section, we will first present the need for merging suffixes and the keyword tree technique to speed up this process. Then we will discuss further the optimization of the mining process when some prior knowledge about the data

distribution is available. Finally, we will provide a solution to balance the MILE algorithm's performance and memory usage when memory is limited.

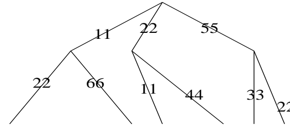

Fig. 4: A keyword tree

Merging Suffixes Why do we need to merge suffixes in the mining process? Assume we have two patterns discovered by PrefixExtend {(55 33)(22)(11)} and {(55)(33)(22)(11)}. For the first pattern $suffixes$(33) contains {(_)(22)(11)} with start locations (12, 24, 35) for prefix 55, and for the second pattern $suffixe$-s(33) contains {(_)(22)(11)} with start locations (15, 39, 43) for prefix 55. When MILE comes to {(55 44)(33)}, it will hash ending locations (12, 24, 39) of 33 in this pattern to get the desired suffixes appended. It can be easily seen from the two sets of start locations {(_)(22)(11)} and ending locations of 33 that {(_)(22)(11)} is to be appended (assuming *minSup*=2) with the appending locations (12, 24) in one set of start locations and (39) in the other set of start locations. However, since {(55 33)(22)(11)} and {(55)(33)(22)(11)} are independent patterns mined separately by PrefixExtend, these two sets of start locations are separately associated with their own {(_)(22)(11)}. If the hitting process in SuffixAppend starts with this situation, the frequencies of two separate suffixes would be 2 and 1 respectively. Neither would be appended, which is not what we expect. How can we merge these two {(_)(22)(11)}'s into one and also their start positions into one set before MILE calls SuffixAppend? To avoid getting too many details involved, let us directly use the fact that in MILE patterns starting with {(55 33)} are mined first in a depth-first style and patterns starting with {(55)(33)} are mined at a later stage. So when PrefixExtend comes to get the suffix {(_)(22)(11)} from {(55)(33)(22)(11)}, another {(_)(22)(11)} is buried by many suffixes from patterns starting with {(55 33)} such as {(_)(11)(22)}, {(_)(11)(66)}, {(_)(22)(44)}, {(_)(55)(33)} and {(_)(55)(22)}. How can we merge the {(_)(22)(11)} into them? First, we need to decide whether there exists {(_)(22)(11)} in the mined suffixes. In a simple way, we do pattern matching suffix by suffix in an $O(nml)$ time where n is the number of mined suffixes, m is the maximum length of a suffix in the mined suffixes, and l is the length of the suffix which needs to be merged. Since this matching process is in the inner loop of MILE, it directly affects the efficiency of MILE.

Instead of using the above naive pattern matching, we use a keyword tree to do a dictionary look-up so that the $O(nml)$ time will be reduced to $O(nm + l)$ [6]. First, we insert all mined suffixes {(_)(11)(22)}, {(_)(11)(66)}, {(_)(22)(44)}, {(_)(55)(33)} and {(_)(55)(22)} into a keyword tree showed in Figure 4 which is similar to the pattern tree in Section 4.2. The insertion involves token comparison

from the root till the edge where differences between token values happen. In the parent node of that edge, insert a new edge with the different token in the inserting suffix labeled on it. Then, we do token comparison of the suffix needed to be merged and one path from the root of the built keyword tree. If a leaf node is reached with exhausting all tokens of the suffix, its start locations will be merged into the mined start location set. If a new edge is generated, this suffix is a new suffix to be put into the set of mined suffixes. After the merging process finishes, SuffixAppend can be called without any problem.

Incorporating Prior Knowledge If some prior knowledge of the data distribution in data streams is available, we can further improve the efficiency of the mining process based on our suffix appending approach. Assume that the users know in advance the frequency of one token's occurrence in some data stream is higher than others'. That means it will have more chance to get more suffixes appended if the mining process of patterns with this token as prefix can be delayed to a later stage. In this way, MILE will avoid more expensive depth-first search. The strategy we employ is to encode such a stream with larger values and the largest value is assigned to the token with the highest frequency. We show this encoding strategy with the following example.

s^1	x	y	z	z	y	x	y	x	z
s^2	e	f	g	e	f	e	g	f	g
s^3	a	a	a	b	c	a	a	a	a

In s^3, token a occurs more frequently than the other two tokens (and tokens in the other data streams are random). So we encode a with the largest value 33 and the streams as follows.

s^3	33	33	33	32	31	33	33	33	33
s^2	20	21	22	20	21	20	22	21	22
s^1	10	11	12	12	11	10	11	10	12

In PrefixExtend, we can control MILE in such a way that patterns with a smaller value as prefix are mined earlier than the ones with a larger value as prefix. It is understandable that subtrees starting with smaller values are searched first in the pattern tree and those subtrees with larger values will use SuffixAppend to explore instead of depth-first search. In general cases, we assign higher encoding values to the tokens of higher frequencies in one data stream and assign a higher encoding value to the stream that contains the token of the highest frequency. Empirical results in Section 5 show that this heuristic can further improve the performance of MILE. Actually, if such frequency information is not available in advance, it can be collected by a straightforward

counting method and be utilized later. Another direction we are now exploring is to collect statistic information from the previous mining procedure and use it to decide which token should be mined earlier to get more benefits from our SuffixAppend approach.

Balancing Memory Usage and Performance MILE uses more memory than PrefixSpan since it records down previously mined suffixes and builds corresponding indices if needed. With advances in computer engineering, the sizes of main memory for computers are growing fast and the price of memory is cheap. Several gigabytes are simply normal with a regular computing server. If the users are more concerned with time efficiency, MILE is clearly a good choice. If the users are also concerned with the memory a data mining system consumes, we now describe a solution to balance the memory usage and time performance of MILE.

In a normal situation, the number of shorter patterns is larger than the number of longer patterns, and the locations (frequencies) of shorter patterns are much higher than the locations (frequencies) of relatively longer patterns. Similar situations exist for mined suffixes. If MILE only records down and builds indices for mined suffixes whose length exceeds a predefined parameter l, and uses PrefixExtend to grow shorter patterns which will not be mined by SuffixAppend due to unrecorded short suffixes, it will use less memory than the original algorithm although the efficiency will degrade at the same time. For example, if the predefined parameter l=1, suffix $\{_ \ 44\}$ for prefix 11 will not be recorded down in the pattern tree in Figure 2, but $\{_ \ 44 \ \beta_1\}$ will (assuming that β_1 contains at least one token). Since the information about suffix $\{_ \ 44\}$ for prefix 11 is not available at a later stage, patterns {22 11 44} and {33 11 44} will be mined in PrefixExtend rather than in SuffixAppend in the original design. Empirical results in Section 5 show that this solution can significantly save memory and in the meanwhile, maintain reasonable efficiency. After all, we can see that the longer suffixes are appended, the more benefits the mining process gets from our suffix appending approach. So when only relatively short suffixes are not used, MILE still works well.

5 Experimental Evaluation

In this section, we compare PrefixSpan and MILE with data sets under different parameter settings. We also analyze those factors that impact the efficiency of MILE.

Experiment Environment. All experiments are performed on a server of four 1GHz SPARC CPUs with 8 gigabyte main memory, running with Solaris 9. We have implemented MILE and PrefixSpan (according to [12]) in Java. Although the server is a multi-user environment, we are interested in a comparison of the CPU time of these two algorithms to see what is the computational bottleneck for sequential pattern mining across multiple data streams. So other running programs on the server do not affect our experimental results. We turn off all outputs of the two programs in our experiments.

Data Generation. We generate data sets with uniform distribution and also multinomial distribution with specified probabilities. Unless explicitly explained otherwise, the data distribution is uniform. Three parameters are used in the name of each data set to indicate the data set's settings. s denotes the number of streams, t denotes the number of time points, and v denotes the number of different tokens per stream. For example, *s3t200v3* means that the data set has 3 streams, 200 time points, and each stream has 3 different tokens.

Performance Comparison with Different Time Points and Window Sizes. First we compare the performance of PrefixSpan and MILE on small (s9t200v4), medium (s9t2000v4) and large (s9t20000v4) data sets with a fixed window size of 4 and various *minSup* values. (Here we use relative values. For example, if we have 50 windows of data and *minSup*=50%, we require the frequency of a pattern to be greater than 25.) Data streams have more values in the time dimension than in other dimensions. So this group of comparisons reflects the normal situation. Figure 5, Figure 6 and Figure 7 show that MILE runs consistently faster than PrefixSpan. Note that when the *minSup* is increased, the number of patterns is decreased and the performance of MILE becomes similar to PrefixSpan. When the number of patterns is very small (for example, less than 10), MILE may be less efficient than PrefixSpan. However, when the *minSup* becomes less and less, the number of patterns becomes more and more and the performance of MILE is consistently much better than PrefixSpan. In the largest data set, MILE can achieve a 46.01% improvement (by (PrefixSpan's CPU time-MILE's CPU time) / PrefixSpan's CPU time, which is denoted as (Pt-Mt) / Pt hereafter) over PrefixSpan when *minSup*=7%. When the *minSup* is small, the number of patterns becomes very large and much more computation is involved than when the *minSup* is large. It is in that point the difference between MILE and PrefixSpan becomes important. When we vary the window size and fix the other factors, Figure 8 shows the consistent performance of MILE.

The Relationship between Efficiency and the Number of Patterns. Intuitively, the larger the number of patterns formed by suffix appending, the faster MILE runs in comparison with PrefixSpan. Figure 9 illustrates this by putting two ratios together: one is (Pt-Mt)/Pt (explained in the last paragraph) standing for the efficiency of MILE; and the other is Sn/Tn which is the ratio of the number of patterns formed by suffix appending over the number of all patterns. From this figure, we see two points. First, the trends of the two curves show that when suffix appending occurs more frequently, the mining process will be faster. Second, even if no suffix appending happens (Sn/Tn=0), the constructed index is not just pure overhead and can actually speed up the mining process as explained in Section 4.2.

Performance Comparison with Prior Knowledge on Data Distributions. Figure 10 demonstrates the performance of MILE when some prior knowledge about data distributions is incorporated into the mining process as described in Section 4.3. We generate data sets in such a way that (1) data set *Mult1* has one stream containing a token (with a probability of 0.55) that happens more frequently than others (each of which is associated with a probability of 0.15); (2) data

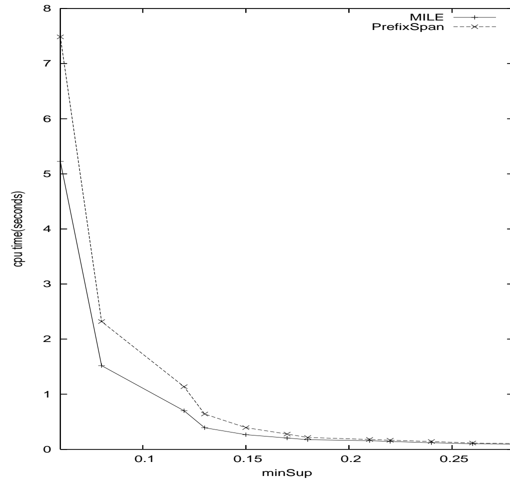

Fig. 5: CPU time comparison, data set s9t200v4, window size=4

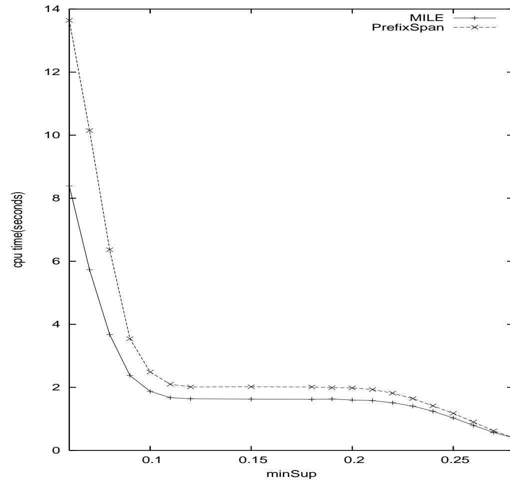

Fig. 6: CPU time comparison, data set s9t2000v4,window size=4

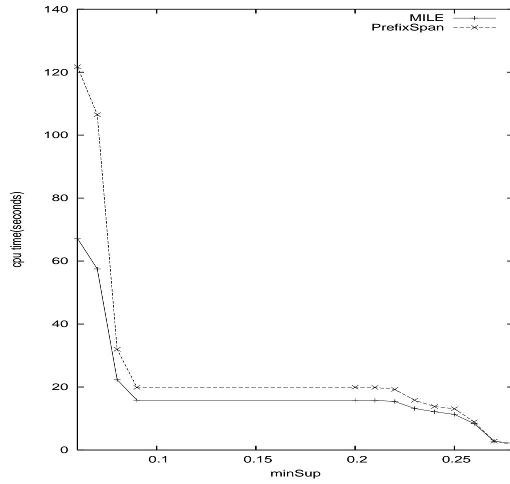

Fig. 7: CPU time comparison, data set s9t20000v4,window size=4

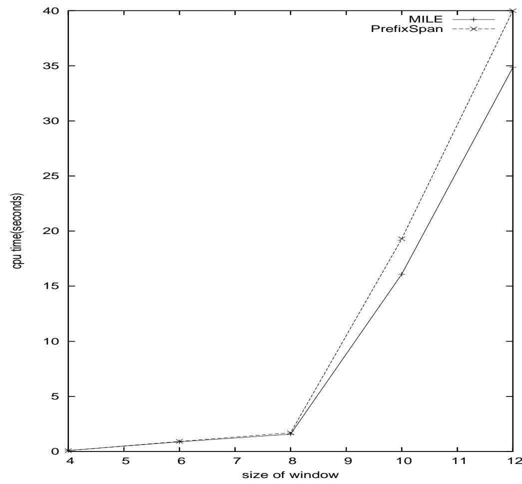

Fig. 8: CPU time comparison when window size is varied with data set s6t2000v6 and minSup=20%

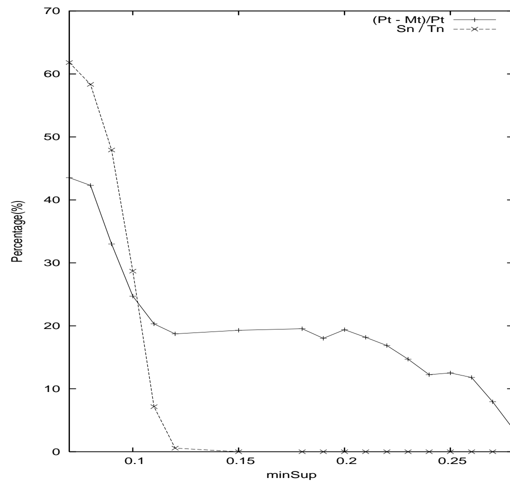

Fig. 9: Relationship between efficiency and the number of patterns formed by suffix appending, data set s9t2000v4, window size=4

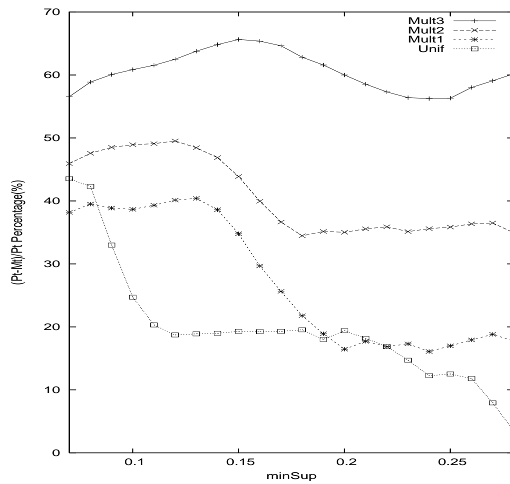

Fig. 10: Efficiency ((Pt-Mt)/Pt) of MILE with incorporated prior knowledge, data sets s9t2000v4 with different distributions, window size=4

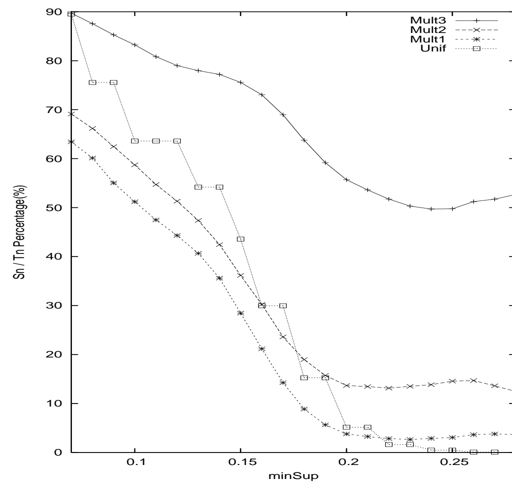

Fig. 11: The ratio Sn / Tn, data sets s9t2000v4 with different distributions, window size=4

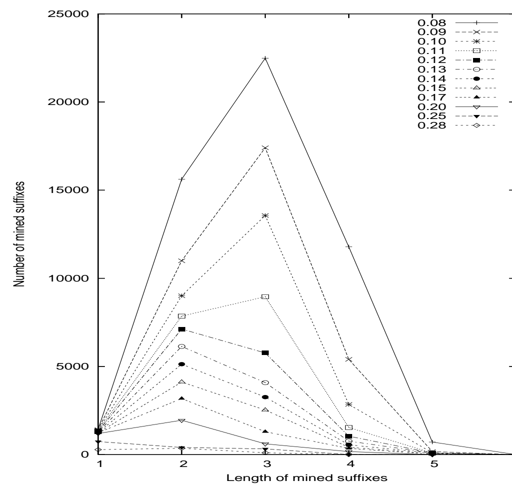

Fig. 12: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s9t2000v4 with Mult3 distribution, window size=4

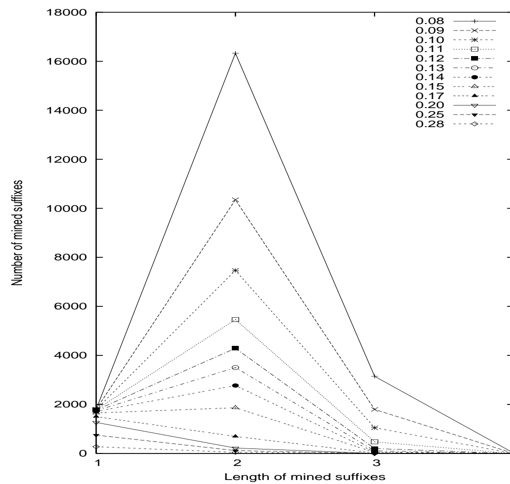

Fig. 13: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s9t2000v4 with Mult2 distribution, window size=4

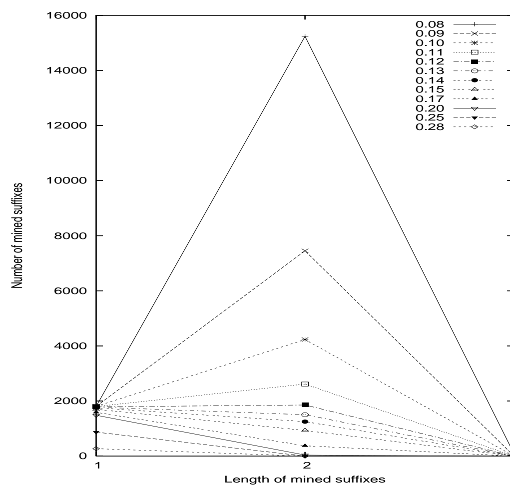

Fig. 14: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s9t2000v4 with Mult1 distribution, window size=4

set *Mult2* has two streams each of which contains a token (with a probability of 0.55) that happens more frequently than others (each of which is associated with a probability of 0.15); and (3) data set *Mult3* has two streams each of which contains a token (with a probability of 0.75) that happens more frequently than others (each of which is associated with a probability of 0.05). From Figure 10, we can see that the performance of MILE in these three data sets is in the order of *Mult3>Mult2>Mult1*. This result shows that when prior knowledge of data distributions is available, we can use the encoding mechanism in Section 4.3 to get more benefits from our suffix appending approach. From Figure 11 the performance is consistent with the ratio Sn/Tn (which indicates how often suffix appending happens). That is, Sn/Tn in these three data sets is in the order of *Mult3>Mult2>Mult1*. Tokens in data set *Unif* are uniformly distributed. This type of data set is the base line. In this case, the average performance of MILE is minimized when no prior knowledge is incorporated. However, the discussion from the previous paragraphs in this section shows that MILE still consistently outperforms PrefixSpan when dealing with data sets of a totally random distribution. Note that although in Figure 11 the ratio Sn/Tn in data set *Unif* is sometimes greater than both *Mult2* and *Mult1* and is even close to the ratio Sn/Tn in *Mult3*, MILE's performance in *Unif* is the lowest (still better than PrefixSpan). Figures 12, 13, 14 and 15 show why this happens. In these four figures, statistics on suffixes of different lengths were collected at the first level of a pattern tree (the suffixes with a single pattern literal as prefix). From these four figures, we can see that *Mult3*, *Mult2* and *Mult1* have more mined suffixes of longer lengths than *Unif*, which roughly indicates that more expensive depth-first search is avoided by our suffix appending approach.

Balance between Memory Usage and Efficiency of MILE. Figure 16 illustrates that the efficiency of our proposed solution in Section 4.3 when memory usage is of the concern of the users. If we need to save memory, MILE does not record down short suffixes nor builds their corresponding location indices. We use MILEM to denote this version of MILE. In Figure 16, the information about suffixes shorter than 2 is not recorded. Since patterns are mostly short in the data set of uniform distribution and this distribution does not hold for most situations, we use multinomial distribution and various lengths of patterns to show the performance of the proposed memory saving solution. Figure 16 shows that the performance of MILE, MILEM and PrefixSpan is in the order of MILE > MILEM > PrefixSpan. Figure 17 compares the amount of memory saved by MILEM over MILE ((Memory used by MILE - Memory used by MILEM) / Memory used by MILE) and the efficiency of MILEM ((Pt-Mt)/Pt). We can see that in most cases MILEM can save a significant amount of memory while maintaining reasonable efficiency. On average, it can save 64% memory over MILE and maintain a 21% improvement over PrefixSpan. When *minSup* is increased, the number of relatively long suffixes becomes less and the performance of MILEM degrades. However, usually the users are more interested in patterns across several data streams to find correlations among them, and these patterns are relatively long in multiple data streams like the distribution indicated by

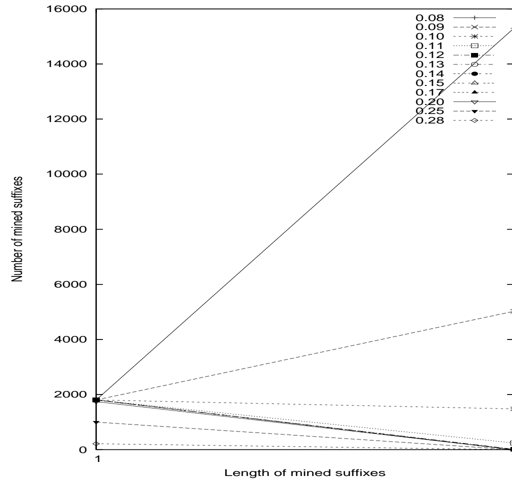

Fig. 15: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s9t2000v4 with uniform distribution, window size=4

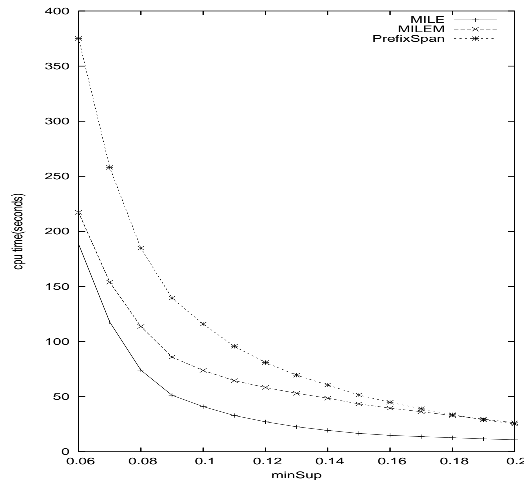

Fig. 16: CPU time comparison, data set s15t2000v4, Mult3 distribution, window size=4

Figure 12 rather than Figure 15. So we can conclude that in a normal situation MILEM works well.

Performance Comparison with Different Numbers of Streams. Figure 18 shows the scalability of MILE with the number of data streams. The results show that MILE runs consistently faster than PrefixSpan. Furthermore, the efficiency of MILE compared with PrefixSpan will become more significant when the number of streams is increased. Actually, from the previous discussions, we can see that the performance of MILE is related with the ratio of the number of patterns formed by suffix appending over the number of all patterns, and also related with the length of suffixes appended. For the first factor, Figure 19 illustrates that the ratio Sn/Tn is increased when the number of streams is increased. For the second factor, we can see from Figures 20, 21 and 22 that the increase in the number of streams does not change much the length of suffixes appended.

6 Conclusions

Discovering frequent patterns over multiple data streams is a nontrivial task for many real-world applications. These patterns can be used to explore event correlations across data streams and assess their causal relationships. Existing studies have concentrated on mining frequent items or itemsets in individual data streams. In this paper, we have defined a challenging problem of mining frequent sequential patterns across multiple data streams. We have proposed an efficient algorithm MILE to solve the problem. The proposed algorithm recursively utilizes the knowledge of the mined patterns from the previous mining procedures to make new patterns' discovery fast. We have also applied a state-of-the-art sequential pattern mining algorithm PrefixSpan to solve our problem. Extensive empirical results show that MILE is significantly faster than PrefixSpan, especially when prior knowledge of the data distribution in the streams is available. To the best of our knowledge, MILE is the only algorithm that can incorporate prior knowledge of the data distribution into the mining process for efficiency. In memory limited environments, we have also proposed a solution to balance the memory usage and time efficiency.

In this paper, we use *consecutive* or *non-overlapping* time windows to model locations of patterns. That is, each window of data is not overlapped by other windows of data. Another option could be *sliding* or *overlapping* time windows. In this case, the users can specify the length of every sliding step. For example, after each time point a time window may slide one step (time point), which drops off the first time point of data in the original window and appends a new time point of data to the rest of data in the original window to form a new window of data. Likewise, a time window can slide two steps by updating two new time points of data. If the length of sliding step equals the window size, a sliding window becomes a consecutive window. A sliding window allows a mining algorithm to discover patterns across two consecutive windows, while consecutive windows may break a pattern into two shorter patterns. One problem of a sliding window is that the redundancy of overlapping data may cause a mining algorithm to return patterns of inaccurate frequency. That is, a short pattern may exist in

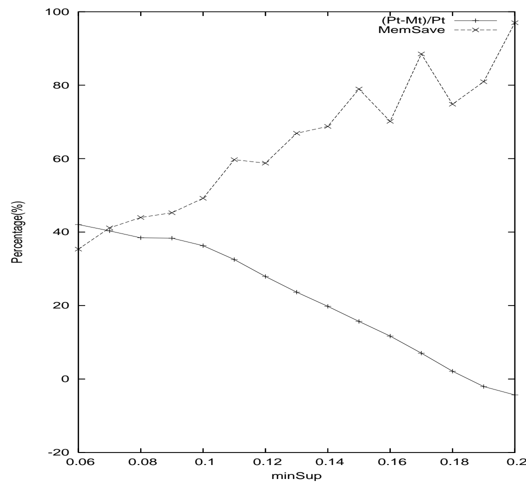

Fig. 17: CPU time vs. memory usage, data set s15t2000v4, Mult3 distribution, window size=4

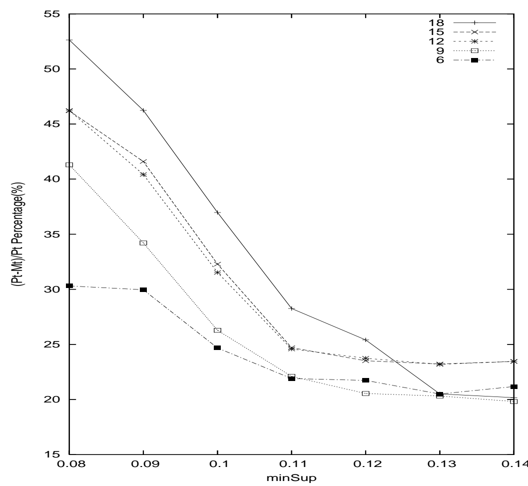

Fig. 18: Efficiency comparison of MILE, with various numbers of streams, data set sXt2000v4 (X is the number of streams labeled in the figure), window size=4.

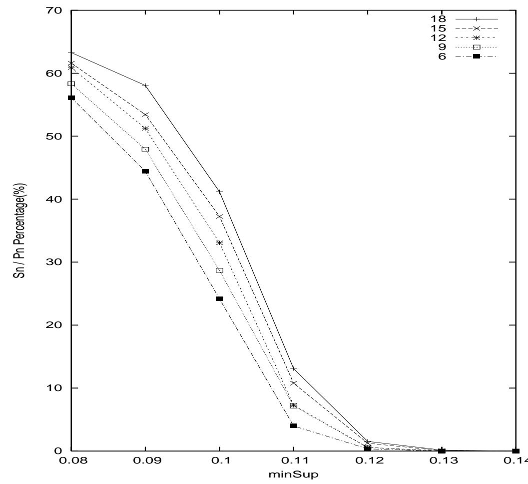

Fig. 19: The ratio Sn / Tn, with various numbers of streams, sXt2000v4 (X is the number of streams labeled in the figure), window size=4.

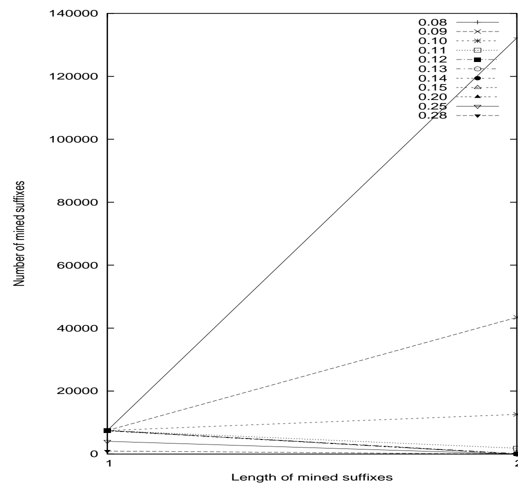

Fig. 20: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s18t2000v4 with uniform distribution, window size=4

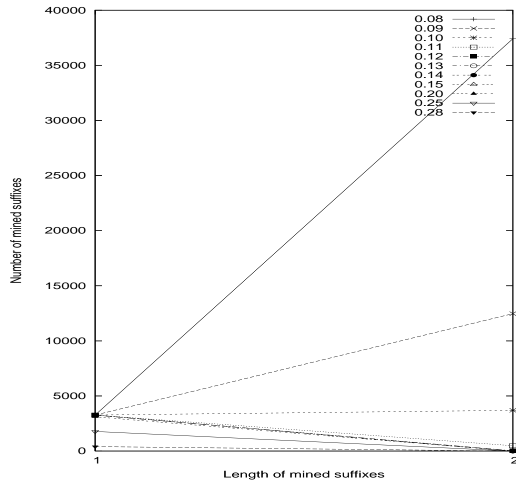

Fig. 21: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s12t2000v4 with uniform distribution, window size=4

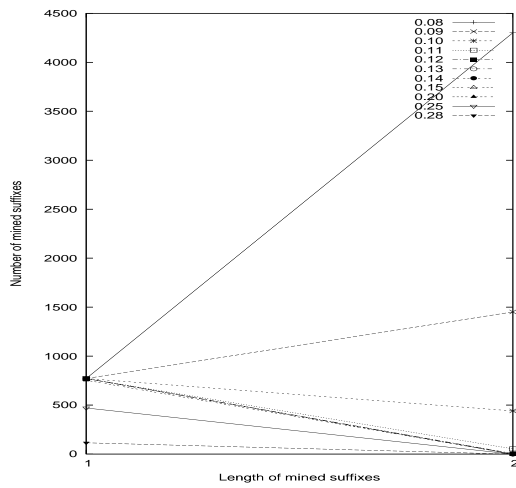

Fig. 22: Approximate distribution of lengths of mined suffixes (at the first level of a pattern tree), data set s6t2000v4 with uniform distribution, window size=4

the overlapping part of several sliding windows and its one occurrence may be counted several times. One possible solution is to push the association of pattern literals into the discovery process. For example, if the length of sliding step is 1, then for the data of a sliding window, only patterns starting at the first time point of that window are considered and other patterns will be considered in next sliding windows. In this case, the overlapping part of data can only be counted once as a part of some independent pattern (the overlapping part of the data can be counted many times as parts of different patterns). Note that different types of time windows may return different sets of patterns and may have impacts on the counting or scanning process of a mining algorithm but do not affect the application of our suffix appending approach. In every situation, the spirit of suffix appending can be used to utilize the knowledge from discovered patterns to speed up the mining process. Other techniques employed in MILE can be used as well under different types of time windows.

We are currently exploring the direction of collecting statistics from previous mining procedures to guide our oncoming mining process in order to maximize the power of our suffix appending approach. Since sequential pattern mining is a very hard combinatorial problem, most (if not all) existing work ([1], [13], [16] and [12]) stays with static data environments. Although [3] and [11] have dealt with searching sequential structures from data streams, they assumed that the whole set of data streams was available in advance. We plan to extend our current work to mine frequent sequential patterns from dynamic data streams.

7 Acknowledgments

We would like to thank Dr. Craig A. Damon for helpful advice on memory-efficient data structures for Java program implementations, Dr. Byung S. Lee for his help with hash indexing, and Dr. Xiaoyang Sean Wang for his enlightenment on the generalization of our problem.

References

1. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, 1995.
2. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of International Colloquium on Automata,Languages, and Programming*, pages 508–515, 2002.
3. G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *Proceedings of the 4th International Conference of Knowledge Discovery and Data Mining*, pages 16–22, 1998.
4. L. Gao and X. S. Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 370–381, 2002.
5. C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. *Mining Frequent Patterns in Data Streams at Multiple Time Granularities-Chapter 3 of Next Generation Data Mining*. AAAI/MIT, 2003.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences–Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.

7. C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of the 12th international conference on Information and knowledge management*, pages 287–294, 2003.
8. E. Keogh and P. Smyth. A probabilistic approach to fast pattern matching in time series databases. In *Proceedings of the 3rd International Conference of Knowledge Discovery and Data Mining*, pages 16–22, 1997.
9. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.
10. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.
11. T. Oates and P. R. Cohen. Searching for structure in multiple streams of data. In *Proceedings of the 13th International Conference on Machine Learning*, pages 346–354, 1996.
12. J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. Knowl. Data Eng.*, 16(11):1424–1440, 2004.
13. R. Srikant and R. Agrawal. Mining sequential patterns: Generalized and performance improvements. In *Proceedings of 5th International Conference on Extending Database Technology*, pages 3–17, 1996.
14. M. Wang and X. S. Wang. Efficient evaluation of composite correlations for streaming time series. In *Proceedings of 4th International Conference on Web-Age Information Management*, pages 369–380, 2003.
15. B. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for co-evolving time sequences. In *Proceedings of the 16th International Conference on Data Engineering*, pages 13–22, 2000.
16. M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Mach. Learn.*, 42(1-2):31–60, 2001.
17. Y. Zhu and D. Shasha. Stastream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 358–369, 2002.