

Chapter 3

Mining Frequent Patterns in Data Streams at Multiple Time Granularities

*Chris Giannella**, *Jiawei Han*[†], *Jian Pei*[‡], *Xifeng Yan*[†],
Philip S. Yu[‡]

*Indiana University, cgiannel@cs.indiana.edu

[†]University of Illinois at Urbana-Champaign, {hanj,xyan}@cs.uiuc.edu

[‡]State University of New York at Buffalo, jianpei@cse.buffalo.edu

[‡]IBM T. J. Watson Research Center, psyu@us.ibm.com

Abstract:

Although frequent-pattern mining has been widely studied and used, it is challenging to extend it to data streams. Compared with mining a static transaction data set, the streaming case has far more information to track and far greater complexity to manage. Infrequent items can become frequent later on and hence cannot be ignored. The storage structure need be dynamically adjusted to reflect the evolution of itemset frequencies over time.

In this paper, we propose an approach based on computing and maintaining all the frequent patterns (which is usually more stable and smaller than the streaming data) and dynamically updating them with the incoming data stream. We extended the framework to mine *time-sensitive* patterns with *approximate* support guarantee. We incrementally maintain *tilted-time* windows for each pattern at multiple time granularities. Interesting

queries can be constructed and answered under this framework.

Moreover, inspired by the fact that the **FP-tree** provides an effective data structure for frequent pattern mining, we develop **FP-stream**, an **FP-tree**-based data structure for maintaining time sensitive frequency information about patterns in data streams. The **FP-stream** can be scanned to mine frequent patterns over multiple time granularities. An **FP-stream** structure consists of (a) an in-memory *frequent pattern-tree* to capture the frequent and sub-frequent itemset information, and (b) a *tilted-time window table for each frequent pattern*. Efficient algorithms for constructing, maintaining and updating an **FP-stream** structure over data streams are explored. Our analysis and experiments show that it is realistic to maintain an **FP-stream** in data stream environments even with limited main memory.

Keywords: frequent pattern, data stream, stream data mining.

3.1 Introduction

Frequent-pattern mining has been studied extensively in data mining, with many algorithms proposed and implemented (for example, **Apriori** [1], **FP-growth** [10], **CLOSET** [17], and **CHARM** [19]). Frequent pattern mining and its associated methods have been popularly used in association rule mining [1], sequential pattern mining [2], structured pattern mining [13], iceberg cube computation [4], cube gradient analysis [12], associative classification [14], frequent pattern-based clustering [18], and so on.

Recent emerging applications, such as network traffic analysis, web click stream mining, power consumption measurement, sensor network data analysis, and dynamic tracing of stock fluctuation, call for study of a new kind of data, *stream data*. Stream data takes the form of continuous, potentially infinite data streams, as opposed to finite, statically stored data sets. Stream data management systems and continuous stream query processors are under intense investigation and development. Besides querying data streams, another important task is to mine data streams for interesting patterns.

There are some recent studies on mining data streams, including classification of stream data [7, 11] and clustering data streams [9, 16]. However, it is challenging to mine frequent patterns in data streams because mining frequent itemsets is essentially a set of join operations as illustrated in **Apriori** whereas join is a typical *blocking operator*, i.e., computation for any itemset cannot complete before seeing the past and future data sets. Since one can only maintain a limited size window due to the huge amount of stream data, it is difficult to mine and update frequent patterns in a dynamic, data stream environment.

In this paper, we study this problem and propose a new methodology: *mining time-sensitive data streams*. Previous work [15] studied the *landmark model*, which mines frequent patterns in data streams by assuming that patterns are measured from the start of the stream up to the current moment. The landmark model may not be desirable since the set of frequent patterns usually are time-sensitive and in many cases, *changes of patterns and their trends* are more interesting than patterns themselves. For example, a shopping transaction stream could start long time ago (e.g., a few years ago), and

the model constructed by treating all the transactions, old or new, equally cannot be very useful at guiding the current business since some old items may have lost their attraction; fashion and seasonal products may change from time to time. Moreover, one may not only want to fade (e.g., reduce the weight of) old transactions but also to find changes or evolution of frequent patterns with time. In network monitoring, the changes of the frequent patterns in the past several minutes are valuable and can be used for detection of network intrusions [6].

In our design, we actively maintain pattern frequency histories under a tilted-time window framework in order to answer time-sensitive queries. A collection of patterns along with their frequency histories are compressed and stored using a tree structure similar to FP-tree [10] and updated incrementally with incoming transactions. In [10], the FP-tree provides a base structure to facilitate mining in a static batch environment. In this paper, an FP-tree is used for storing transactions for the current time window; in addition, a similar tree structure, called *pattern-tree*, is used to store collections of itemsets and their frequency histories. Our time-sensitive stream mining data structure, FP-stream, includes two major components: (1) an *pattern-tree*, and (2) *tilted-time windows*.

We summarize the contributions of the paper. First, we develop a data structure, FP-stream, supporting time-sensitive mining of frequent patterns in a data stream. Next, we develop an efficient algorithm to incrementally maintain an FP-stream. Third, we describe how time-sensitive queries can be answered over data streams with an error bound guarantee.

The remainder of the paper is organized as follows. Section 3.2 presents the problem definition and provides a basic analysis of the problem. Section 3.3 presents the FP-stream data structure. Section 3.4 introduces the maintenance of tilted-time windows, while Section 3.5 discusses the issues of minimum support. The algorithm is outlined in Section 3.6. Section 3.7 reports the results of our experiments and performance study. Section 3.8 discusses how the FP-stream can be extended to include fading time windows. Section 3.9 discusses some of the broader issues in stream data mining and how our approach applies.

3.2 Problem Definition and Analysis

Our task is to *mine frequent patterns over arbitrary time intervals in a data stream* assuming that one can only see the set of transactions in a limited size window at any moment.

To study frequent pattern mining in data streams, we first examine the same problem in a transaction database. To justify whether a single item i_a is frequent in a transaction database DB , simply scan DB and count the number of transactions in which i_a appears (the frequency). The frequency of every single item can be computed in one scan of DB . However, it is too costly to compute, in one scan, the frequency of every possible combination of single items because of the huge number of such combinations. An efficient alternative proposed in the Apriori algorithm [1] is to count only those itemsets *whose every proper subset is frequent*. That is, at the k -th scan of DB , derive the frequent itemsets of length k (where $k \geq 1$), and then derive the set of length

$(k + 1)$ candidate itemsets (i.e. those whose every length k subset is frequent) for the next scan.

There are two difficulties in using an Apriori-like algorithm in a data stream environment. Frequent itemset mining by Apriori is essentially a set of join operations as shown in [1]. However, join is a typical *blocking operator* [3] which cannot be performed over stream data since one can only observe at any moment a very limited size window of a data stream.

To ensure the completeness of frequent patterns for stream data, it is necessary to store not only the information related to frequent items, but also that related to infrequent ones. If the information about the *currently infrequent items* were not stored, such information would be lost. If these items become frequent later, it would be impossible to figure out their correct overall support and their connections with other items. However, it is also unrealistic to hold all streaming data in the limited main memory. Thus, we divide patterns into three categories: *frequent patterns*, *subfrequent patterns*, and *infrequent patterns*.

Definition 1 The frequency of an itemset I over a time period T is the number of transactions in T in which I occurs. The support of I is the frequency divided by the total number of transactions observed in T . Let the *min-support* be σ and the *relaxation ratio* be $\rho = \epsilon/\sigma$, where ϵ is the *maximum support error*. I is frequent if its support is no less than σ ; it is sub-frequent if its support is less than σ but no less than ϵ ; otherwise, it is infrequent. ■

We are only interested in frequent patterns. But we have to maintain subfrequent patterns since they may become frequent later. We want to discard infrequent patterns since the number of infrequent patterns are really large and the loss of support from infrequent patterns will not affect the calculated support too much. The definition of frequent, subfrequent, and infrequent patterns is actually relative to period T . For example, a pattern I may be subfrequent over a period T_1 , but it is possible that it becomes infrequent over a longer period T_2 ($T_1 \subset T_2$). In this case, we can conclude that I will not be frequent over period T_2 . In our design, the complete structure, FP-stream, consists of two parts: (1) a global frequent pattern-tree held in main memory, and (2) tilted-time windows embedded in this pattern-tree. Incremental updates can be performed on both parts of the FP-stream. Incremental updates occur when some infrequent patterns become (sub)frequent, or vice versa. At any moment, the set of frequent patterns over a period can be obtained from FP-stream residing in the main memory (with a support error bounded above by ϵ).

3.3 Mining Time-Sensitive Frequent Patterns in Data Streams

The design of the *tilted-time window* [5] is based on the fact that people are often interested in recent changes at a fine granularity, but long term changes at a coarse granularity. Fig. 3.1 shows such a tilted-time window: the most recent 4 quarters of an hour, then the last 24 hours, and 31 days. Based on this model, one can compute

frequent itemsets in the last hour with the precision of a quarter of an hour, the last day with the precision of an hour, etc. This model registers only $4 + 24 + 31 = 59$ units of time, with an acceptable trade-off of lower granularity at distant times.

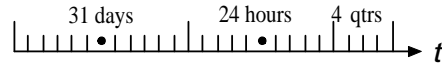


Figure 3.1: Natural Tilted-Time Window Frames

As shown in Figure 3.2, for each tilted-time window, a collection of patterns and their frequencies can be maintained. Assuming these collections contain the frequent patterns (and possibly more), the following queries can be answered: (1) what is the frequent pattern set over the period t_2 and t_3 ? (2) what are the periods when (a, b) is frequent? (3) does the support of (a) change dramatically in the period from t_3 to t_0 ? and so on. That is, one can (1) mine frequent patterns in the current window, (2) mine frequent patterns over time ranges with granularity confined by the specification of window size and boundary, (3) put different weights on different windows to mine various kinds of weighted frequent patterns, and (4) mine evolution of frequent patterns based on the changes of their occurrences in a sequence of windows. Thus we have the flexibility to mine a variety of frequent patterns associated with time.

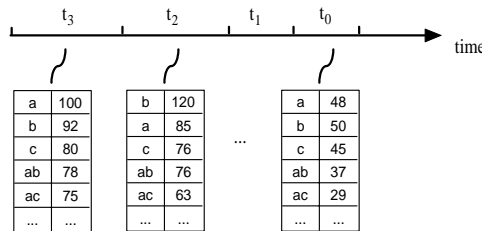


Figure 3.2: Pattern Frequencies for Tilted-Time Windows

A compact tree representation of the pattern collections, called **pattern-tree**, can be used. Figure 3.3 shows an example. Each node in the pattern tree represents a pattern (from root to this node) and its frequency is recorded in the node. This tree shares a similar structure with an **FP-tree**. The difference is that it stores patterns instead of transactions. In fact, we can use the same **FP-tree** construction method in [10] to build this tree by taking the set of patterns as input.

The patterns in adjacent time windows will likely be very similar. Therefore, the tree structure for different tilted-time windows will likely have considerable overlap. Embedding the tilted-time window structure into each node, will likely save considerable space. Thus we propose to use only one pattern tree, where at each node, the frequency for each tilted-time window is maintained. Figure 3.4 shows an example of a

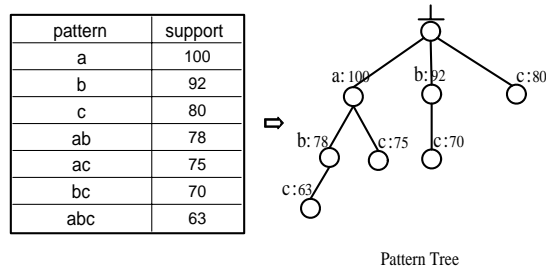


Figure 3.3: Pattern Tree

pattern tree with tilted-time windows embedded. We call this structure an FP-stream.

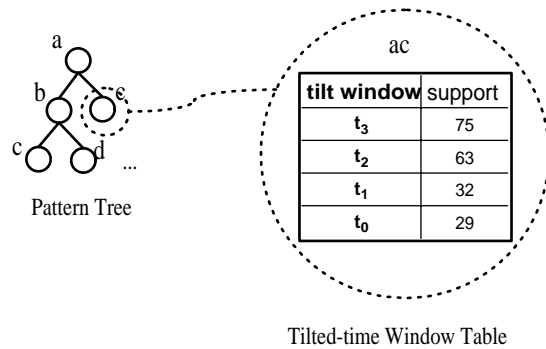


Figure 3.4: Pattern-Tree with Tilted-Time Windows Embedded

3.4 Maintaining Tilted-Time Windows

With the arrival of new data, the tilted-time window table will grow. In order to make the table compact, tilted-time window maintenance mechanisms are developed based on a tilted-time window construction strategy.

3.4.1 Natural Tilted-Time Window

For the natural tilted-time window discussed before (shown in Figure 3.1), the maintenance of windows is straightforward. When four quarters are accumulated, they merge together to constitute one hour. After 24 hours are accumulated, one day is built. In the natural tilted-time window, at most 59 tilted windows need to be maintained for a

period of one month. In the following section, we introduce a logarithmic tilted-time window schema which will reduce the number of tilted-time windows used.

3.4.2 Logarithmic Tilted-Time Window

As an alternative, the tilted-time window frame can also be constructed based on a logarithmic time scale as shown in Figure 3.5. Suppose the current window holds the transactions in the current quarter. Then the remaining slots are for the last quarter, the next two quarters, 4 quarters, 8 quarters, 16 quarters, etc., growing at an exponential rate of 2. According to this model, one year of data will require $\log_2(365 \times 24 \times 4) + 1 \approx 17$ units of time instead of $366 \times 24 \times 4 = 35,136$ units. As we can see, the logarithmic tilted-time window schema is very space-efficient.

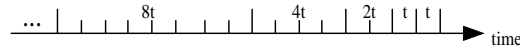


Figure 3.5: Tilted-Time Window Frame with Logarithmic Partition

Formally, we assume that the stream of transactions is broken up into fixed sized batches $B_1, B_2, \dots, B_n, \dots$, where B_n is the most current batch and B_1 the oldest. For $i \geq j$, let $B(i, j)$ denote $\bigcup_{k=j}^i B_k$. For a given itemset, I , let $f_I(i, j)$ denote the frequency of I in $B(i, j)$ (I is omitted if clear from context). A logarithmic tilted-time window is used to record frequencies for itemset I . The following frequencies are kept

$$f(n, n); f(n-1, n-1); f(n-2, n-3); f(n-4, n-7), \dots$$

The ratio r between the size of two neighbor tilted-time windows reflects the growth rate of window size, which usually should be larger than 1. The above example illustrates a logarithmic tilted-time window with ratio of 2. Note that there are $\lceil \log_2(n) \rceil + 1$ frequencies. So even for a very large number of batches, the maximum number of frequencies is reasonable (e.g., 10^9 batches requires 31 frequencies).

However, in a logarithmic tilted-time window, intermediate buffer windows need to be maintained. These intermediate windows will replace or be merged with tilted-time windows when they are full.

3.4.3 Logarithmic Tilted-Time Window Updating

Given a new batch of transactions B , we describe how the logarithmic tilted-time window for I is updated. First, replace $f(n, n)$, the frequency at the finest level of time granularity (level 0), with $f(B)$ and shift $f(n, n)$ back to the next finest level of time granularity (level 1). $f(n, n)$ replaces $f(n-1, n-1)$ at level 1. Before shifting $f(n-1, n-1)$ back to level 2, check if the intermediate window for level 1 is full. If not, $f(n-1, n-1)$ is not shifted back; instead it is placed in the intermediate window and the algorithm stops (in the example in the previous sub-section, the intermediate window for all levels is empty). If the intermediate window is full (say with a frequency f), then $f(n-1, n-1) + f$ is shifted back to level 2. This process continues

until shifting stops. Consider the following example over batches B_1, \dots, B_8 . The tilted-time window initially looks like

$$f(8, 8); f(7, 7); f(6, 5); f(4, 1).$$

$f(8, 8)$ resides in the window for granularity level 0, $f(7, 7)$ for level 1, $f(6, 5)$ for level 2, $f(4, 1)$ for level 3. The intermediate windows at each level are empty and thus not shown. Upon arrival of B_9 we update the tilted-time window

$$f(9, 9); f(8, 8)[f(7, 7)]; f(6, 5); f(4, 1).$$

$f(9, 9)$ replaces $f(8, 8)$ at level 0 which is shifted back to level 1 replacing $f(7, 7)$. Since the intermediate window for level 1 is empty, $f(7, 7)$ is put into the window and the shifting stops ([. . .] denotes an intermediate window). Upon arrival of B_{10} , updating requires several steps. First, we replace $f(9, 9)$ by $f(10, 10)$ and shift $f(9, 9)$ back. The intermediate window at level 1 is full, so the frequencies at level 1 are merged (producing $f(8, 7) = f(8, 8) + f(7, 7)$). $f(8, 7)$ is shifted back to level 2 replacing $f(6, 5)$. Since the intermediate window at that level is empty, $f(6, 5)$ is put into the intermediate window and the shifting stops. The result is

$$f(10, 10); f(9, 9); f(8, 7)[f(6, 5)]; f(4, 1).$$

Upon arrival of B_{11} we update and get

$$f(11, 11); f(10, 10)[f(9, 9)]; f(8, 7)[f(6, 5)]; f(4, 1).$$

Finally, upon arrival of B_{12} we get

$$f(12, 12); f(11, 11); f(10, 9); f(8, 5)[f(4, 1)].$$

Notice that *only one* entry is needed in intermediate storage at any granularity level. Hence, the size of the tilted-time window can grow no larger than $2\lceil \log_2(N) \rceil + 2$ where N is the number of batches seen thus far in the stream. There are two basic operations in maintaining logarithmic tilted-time windows: One is frequency merging; and the other is entry shifting. For N batches, we would like to know how many such operations need to be done for each pattern. The following claim shows the amortized number of shifting and merging operations need to be done, which shows the efficiency of logarithmic scale partition. For any pattern, the amortized number of shifting and merging operations is the total number of such operations performed over N batches divided by N .

Claim 3.4.1 *In the logarithmic tilted-time window updating, the amortized number of shifting and merging operations for each pattern is $O(1)$.*

3.5 Minimum Support

Let t_0, \dots, t_n be the tilted-time windows which group the batches seen thus far in the stream, where t_n is the oldest (be careful, this notation differs from that of the B 's in

the previous section). We denote the window size of t_i (the number of transactions in t_i) by w_i . Our goal is to mine all frequent itemsets whose supports are larger than σ over period $T = t_k \cup t_{k+1} \cup \dots \cup t_{k'}$ ($0 \leq k \leq k' \leq n$). The size of T is $W = w_k + w_{k+1} + \dots + w_{k'}$. If we maintained all possible itemsets in all periods no matter whether they were frequent or not, this goal could be met.¹ However, this will require too much space, so we only maintain $f_I(t_0), \dots, f_I(t_{m-1})$ for some m ($0 \leq m \leq n$) and drop the remaining tail sequences of tilted-time windows. Specifically, we drop tail sequences $f_I(t_m), \dots, f_I(t_n)$ when the following condition holds,

$$\forall i, m \leq i \leq n, f_I(t_i) < \sigma w_i \text{ and } \sum_{j=0}^i f_I(t_j) < \epsilon \sum_{j=0}^i w_j. \quad (3.1)$$

As a result, we no longer have an exact frequency over T , rather an *approximate frequency* $\hat{f}_I(T) = \sum_{i=k}^{\min\{m-1, k'\}} f_I(t_i)$ if $m > k$ and $\hat{f}_I(T) = 0$ if $m \leq k$. The approximation is less than the actual frequency

$$f_I(T) - \epsilon W \leq \hat{f}_I(T) \leq f_I(T). \quad (3.2)$$

Thus if we deliver all itemsets whose approximate frequency is larger than $(\sigma - \epsilon)W$, we will not miss any frequent itemsets in period T ([15] discussed the landmark case). However, we may return some itemsets whose frequency is between $(\sigma - \epsilon)W$ and σW . This is reasonable when ϵ is small.

Based on inequality (3.2), we draw the following claim that the pruning of the tail of a tilted-time window table does not compromise our goal.

Claim 3.5.1 *Consider itemset I . Let m be the minimum number satisfying the condition (3.1). We drop the tail frequencies from $f_I(t_m)$ to $f_I(t_n)$. For any period $T = t_k \cup \dots \cup t_{k'}$ ($0 \leq k \leq k' \leq n$), if $f_I(T) \geq \sigma W$, then $\hat{f}_I(T) \geq (\sigma - \epsilon)W$.*

The basic idea of Claim 3.5.1 is that if we prune I 's tilted-time window table to t_0, \dots, t_{m-1} , then we can still find all frequent itemsets (with support error ϵ) over any user-defined time period T . We call this pruning *tail pruning*.

Itemsets and their tilted-time window tables are maintained in the FP-stream data structure. When a new batch B arrives, mine the itemsets from B and update the FP-stream structure. For each I mined in B , if I does not appear in the structure, add I if $f_I(B) \geq \epsilon|B|$. If I does appear, add $f_I(B)$ to I 's table and then do tail pruning. If all of the windows are dropped, then drop I from FP-stream.

This algorithm will correctly maintain the FP-stream structure, but not very efficiently. We have the following anti-monotone property for the frequencies recorded in tilted-time window tables.

Claim 3.5.2 *Consider itemsets $I \subseteq I'$ which are both in the FP-stream structure at the end of a batch. Let $f_I(t_0), f_I(t_1), \dots, f_I(t_k)$ and $f_{I'}(t_0), f_{I'}(t_1), \dots, f_{I'}(t_k)$ be*

¹Maintaining only frequent tilted-time window entries will not work. As the stream progresses, infrequent entries may be needed to account for itemsets going from infrequent to frequent.

the entries maintained in the tilted-time window tables for I and I' , respectively. The following statements hold.

1. $k \geq l$.
2. $\forall i, 0 \leq i \leq l, f_I(t_i) \geq f_{I'}(t_i)$.

Claim 3.5.2 shows the property that the frequency of an itemset should be equal to or larger than the support of its supersets still holds under the framework of approximate frequency counting and tilted-time window scenario. Furthermore, the size of the tilted-time window table of I should be equal to or larger than that of its supersets. This claim allows for some pruning in the following way. If I is found in B but is not in the FP-stream structure, then by Claim 3.5.2 part 1, no superset is in the structure. Hence, if $f_I(B) < \epsilon|B|$, then none of the supersets need be examined. So the mining of B can prune its search and not visit supersets of I . We call this type of pruning *Type I Pruning*.

By Claim 3.5.1 and 3.5.2, we conclude the following anti-monotone property which can help in efficiently cutting off infrequent patterns.

Claim 3.5.3 Consider a pattern $I \subseteq I'$, the following statements hold.

1. if the tail frequencies $f_I(t_m) \dots f_I(t_n)$ can be safely dropped based on Claim 3.5.1, then I' can safely drop any frequency among $f_{I'}(t_m) \dots f_{I'}(t_n)$ if it has.
2. if all the frequencies $f_I(t_0) \dots f_I(t_n)$ can be safely dropped based on Claim 3.5.1, then I' together with all its frequencies can be safely dropped.

Claim 3.5.3 part 2 essentially says that if all of I 's tilted-time window table entries are pruned (hence I is dropped), then any superset will also be dropped. We call this type of pruning *Type II Pruning*.

3.6 Algorithm

In this section, we describe in more detail the algorithm for constructing and maintaining the FP-stream structure. In particular we incorporate the pruning techniques into the high-level description of the algorithm given in the previous section.

The update to the FP-stream structure is *bulky*, done only when enough incoming transactions have arrived to form a new batch B_i . The algorithm treats the first batch differently from the rest as an initialization step. As the transactions for B_1 arrive, the frequencies for all items are computed, and the transactions are stored in main memory. An ordering, f_list , is created in which items are ordered by decreasing frequencies (just as done in [10]). This ordering remains fixed for all remaining batches. Once all the transactions for B_1 have arrived (and stored in memory), the batch in memory is scanned creating an FP-tree pruning all items with frequency less than $\epsilon|B_1|$. Finally, an FP-stream structure is created by mining all ϵ -frequent itemsets from the FP-tree (the batch in memory and transaction FP-tree are discarded). All the remaining batches B_i , for $i \geq 2$, are processed according to the algorithm below.

Algorithm 1 (FP-streaming) (*Incremental update of the FP-stream structure with incoming stream data*)

INPUT: (1) An FP-stream structure, (2) a *min_support* threshold, σ , (3) an error rate, ϵ , and (4) an incoming batch, B_i , of transactions (these actually are arriving one at a time from a stream), (5) an item ordering *f_list*.

OUTPUT: The updated FP-stream structure.

METHOD:

1. Initialize the FP-tree to empty.
2. Sort each incoming transaction t , according to *f_list*, and then insert it into the FP-tree without pruning any items.
3. When all the transactions in B_i are accumulated, update the FP-stream as follows.
 - (a) Mine itemsets out of the FP-tree using FP-growth algorithm in [10] modified as below. For each mined itemset, I , check if I is in the FP-stream structure. If I is in the structure, do the following.
 - i. Add $f_I(B)$ to the tilted-time window table for I as described in Section 3.4.3.
 - ii. Conduct tail pruning.
 - iii. If the table is empty, then FP-growth stops mining supersets of I (Type II Pruning). Note that the removal of I from the FP-stream structure is deferred until the scanning of the structure (next step).
 - iv. If the table is not empty, then FP-growth continues mining supersets of I .

If I is not in the structure and if $f_I(B) \geq \epsilon|B|$, then insert I into the structure (its tilted-time window table will have only one entry, $f_I(B_i)$). Otherwise, FP-growth stops mining supersets of I (Type I Pruning).

- (b) Scan the FP-stream structure (depth-first search). For each itemset I encountered, check if I was updated when B was mined. If not, then insert 0 into I 's tilted-time window table (I did not occur in B)². Prune I 's table by tail pruning.

Once the search reaches a leaf, if the leaf has an empty tilted-time window table, then drop the leaf. If there are any siblings of the leaf, continue the search with them. If there were no siblings, then return to the parent and continue the search with its siblings. Note that if all of the children of the parent were dropped, then the parent becomes a leaf node and might be dropped. ■

3.7 Performance Study and Experiments

In this section, we report our performance study. We describe first our experimental set-up and then our results.

²By recording some additional time-stamp information, these zero tilted-time window entries could be dropped. However, in the interests of simplicity, we did not do so and leave it for future work.

3.7.1 Experimental Set-Up

Our algorithm was written in C and compiled using gcc with the -lm switch. All of our experiments are performed on a SUN Ultra-5 workstation using a 333 MHz Sun UltraSPARC-III processor, 512 MB of RAM, and 1350 MB of virtual memory. The operating system in use was SunOS 5.8. All experiments were run without any other users on the machine.

The stream data was generated by the IBM synthetic market-basket data generator, available at “www.almaden.ibm.com/cs/quest/syndata.html/#assocSynData” (managed by the Quest data mining group). In all the experiments 3M transactions were generated using 1K distinct items. The average number of items per transaction was varied as described below. The default values for all other parameters of the synthetic data generator were used (*i.e.*, number of patterns 10000, average length of the maximal pattern 4, correlation coefficient between patterns 0.25, and average confidence in a rule 0.75).

The stream was broken into batches of size 50K transactions and fed into our program through standard input. The support threshold σ was varied (as described below) and ϵ was set to 0.1σ .³ Note that the underlying statistical model used to generate the transactions does not change as the stream progresses. We feel that this does not reflect reality well. In reality, seasonal variations may cause the underlying model (or parameters of it) to shift in time. A simple-minded way to capture some of this shifting effect is to periodically, randomly permute some item names. To do this, we use an item mapping table, M . The table initially maps all item names to themselves (*i.e.* $M(i) = i$). However, for every five batches 200 random permutations are applied to the table⁴.

3.7.2 Experimental Results

We performed two sets of experiments. In the first set of experiments, σ was fixed at 0.005 (0.5 percent) and ϵ at 0.0005. In the second set of experiments σ was fixed at 0.0075 and ϵ at 0.00075. In both sets of experiments three separate data sets were fed into the program. The first had an average transaction length 3, the second 5, and the third 7. At each batch the following statistics were collected: the total number of seconds required per batch (TIME),⁵ the size of the FP-stream structure at the end of each batch in bytes (SIZE),⁶ the total number of itemsets held in the FP-stream structure at the end of the batch (NUM ITEMSETS), and the average length of an itemset in the FP-stream at the end of each batch (AVE LEN). In all graphs presented the x axis represents the batch number. Moreover “Support” is used to denote σ .

Figures 3.6 and 3.7 show TIME and SIZE results, respectively. In each figure the top graph shows the results for average transaction length 3, the middle one shows

³Not all 3M transactions are processed. In some cases only 41 batches are processed (2.05M transactions), in other cases 55 batches (2.75M transactions).

⁴A random permutation of table entries i and j means that $M(i)$ is swapped with $M(j)$. When each transaction $\{i_1, \dots, i_k\}$ is read from input, before it is processed, it is transformed to $\{M(i_1), \dots, M(i_k)\}$.

⁵Includes the time to read transactions from standard input.

⁶Does not include the temporary FP-tree structure used for mining the batch.

average transaction length 5, and the bottom one shows average transaction length 7.

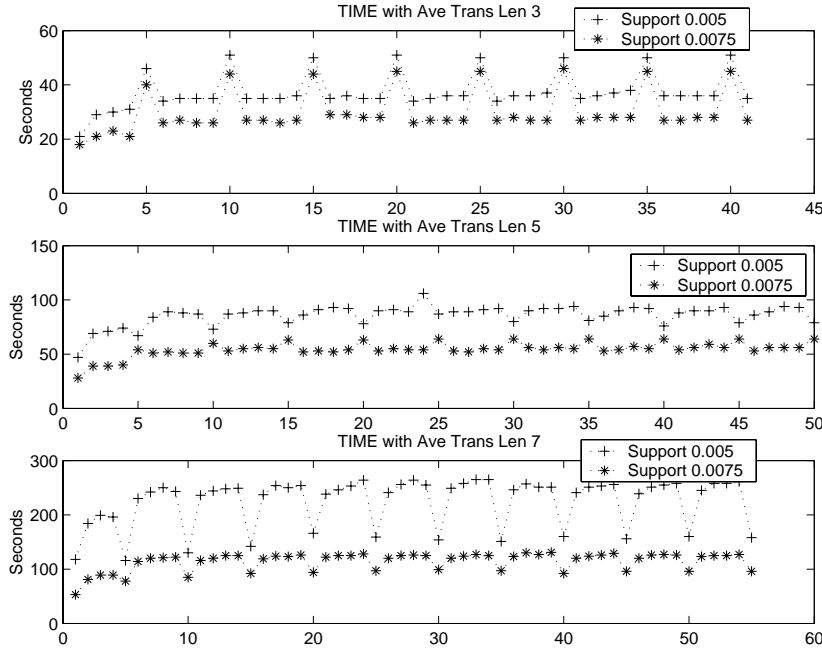


Figure 3.6: FP-stream time requirements

As expected, the item permutation causes the behavior of the algorithm to jump at every five batches. But, stability is regained quickly. In general, the time and space requirements of the algorithm tend to stabilize or grow very slowly as the stream progresses (despite the random permutations). For example, the time required with average transaction length 5 and support 0.0075 (middle graph figure 3.6) seems to stabilize at 50 seconds with very small bumps at every 5 batches. The space required (middle graph figure 3.7) seems to stabilize at roughly 350K with small bumps. The stability results are quite nice as they provide evidence that the algorithm can handle long data streams.

The overall space requirements are very modest in all cases (less than 3M). This can easily fit into main memory. To analyze the time requirements, first recall that the algorithm is to be used in a batch environment. So, we assume that while the transactions are accumulating for a batch, updates to the FP-stream structure from the previous batch can be commencing. The primary requirement, in our opinion, is that the algorithm not fall behind the stream. In other words, as long as the FP-stream structure can be updated before the next batch of transactions is processed, the primary requirement is met. Consider the case of average transaction length three and $\sigma = 0.0075$ (top graph in figure 3.6). The time stabilizes to roughly 25 seconds per batch. Hence, the algorithm can handle a stream with arrival rate 2000 transaction per second (batch size divided by time). This represents the best case of our experiments. In

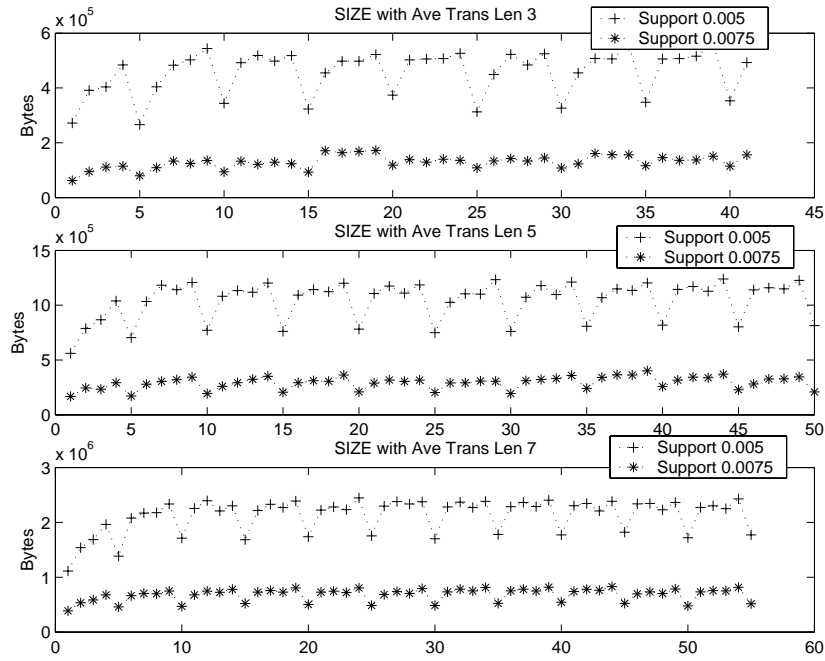


Figure 3.7: FP-stream space requirements

the worst case (average transaction length 7 and $\sigma = 0.0075$) the rate is roughly 180 transactions per second. While this rate is not as large as we would like, we feel that considerable improvement can be obtained since the implementation is currently simple and straight-forward with no optimizations.

In some circumstances it is acceptable to only mine small itemsets. If the assumption is made that only small itemsets are needed, then the algorithm can prune away a great deal of work. Figure 3.8 shows the time performance of the algorithm when the length of the itemsets mined is bounded by two. We see that the times for average transaction length 3 (figure 3.8 top graph) are not much smaller than those where all itemsets were mined (figure 3.6 top graph). But the difference is significant for average transaction length 7. Here the algorithm with itemsets of length bounded by two at support 0.005 can handle a stream with arrival rate 556 transactions per second (the unbounded itemset lengths algorithm could handle a rate of 180).

An interesting observation can be made concerning the “spikes” and “troughs” in figures 3.6 and 3.7. Considering SIZE we see that the random permutations cause a narrow trough (drop) in space usage. We conjecture that the permutations cause some itemsets in the tree to be dropped due to a sharp decrease in their frequency. Considering TIME we see that the permutations cause a narrow spike (increase) in the top graph at both support thresholds. In the middle graph the spiking behavior persists for threshold 0.0075 but switches to troughs for threshold 0.005. Finally, in the bottom

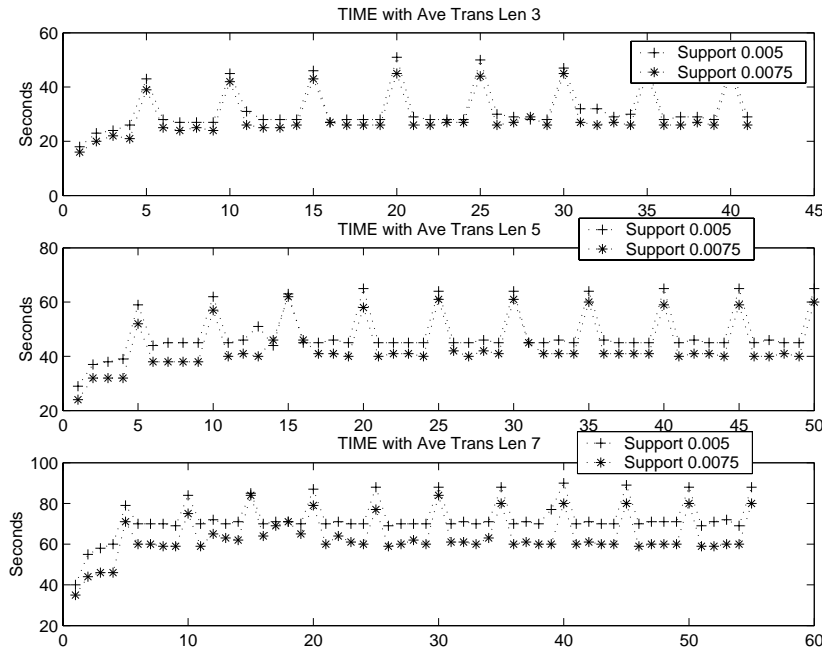


Figure 3.8: FP-stream time requirements—itemset lengths mined are bounded by two

graph, troughs can be seen for both thresholds.

The switching from spikes to troughs is an interesting phenomena. As of yet we do not know its cause but do put forth a conjecture. When an item permutation occurs, many itemsets that appear in the FP-stream structure no longer appear in the new batch and many itemsets that do not appear in the structure appear in the new batch. This results in two competing factors: (1) mining the batch requires less work because itemsets in the structure that do not appear in the batch need not be updated; and (2) mining the batch requires more work because itemsets not in the structure that were sub-frequent in the current batch need be added. When the average transaction length is small (say 3), condition (2) dominates—resulting in a spike. When it is large (say 7), condition (1) dominates—resulting in a trough.

Finally, we describe some results concerning the nature of the itemsets in the FP-stream structure. Figures 3.9 and 3.10 show the average itemset length and the total number of itemsets, respectively.⁷

Note that while the average itemset length does not seem to increase with average transaction length, the number of itemsets does. This is consistent with our running the Apriori program of C. Borgelt⁸ on two datasets consisting of 50K transactions, 1K items, and average transaction lengths 5 and 7, respectively. The support threshold

⁷The maximum itemset length was between 8 and 11 in all experiments.

⁸fuzzy.cs.uni-magdeburg.de/borgelt/software.html#assoc

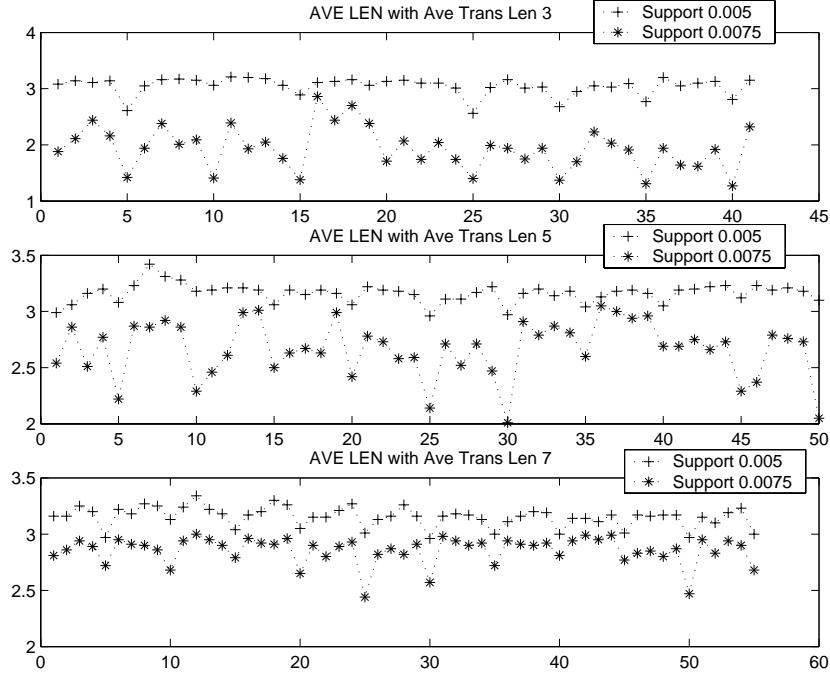


Figure 3.9: FP-stream average itemset length

in each case was 0.0005 (corresponding to ϵ in our $\sigma = 0.005$ experiments). The itemsets produced by Apriori should be exactly the same as those in the FP-stream after the first batch (the leftmost point in middle and bottom graphs in figure 3.10). We observed that the make-up of the itemset lengths from Apriori was nearly the same for both datasets: $\approx 3\%$ size one, $\approx 33\%$ size two, $\approx 23\%$ size three, $\approx 18\%$ size four, $\approx 12\%$ size five, $\approx 7\%$ size six, $\approx 3\%$ size seven, and $\approx 1\%$ sizes eight, nine, and ten combined.

3.8 Time Fading Framework

In the previous discussion, we introduced natural and logarithmic tilted-time window partitions. Both of them give finer granularity to the recent and coarser granularity to the past. However, they do not discount the support of past transactions. In order to discount the past transactions, we introduce a fading factor ϕ . Suppose we have fixed sized batches B_1, B_2, \dots, B_n , where B_n is the most current batch and B_1 the oldest. For $i \geq j$, let $B(i, j)$ denote $\bigcup_{k=j}^i B_k$. For $B(i, j)$, the actual window size is $\sum_{k=j}^i |B_k|$. In a fading framework, the faded window size for $B(i, j)$ is $\sum_{k=j}^i \phi^{i-k} |B_k|$ and its faded support is $\sum_{k=j}^i \phi^{i-k} f_I(B_k)$. We do not change Algorithm 1, that means, we still drop infrequent patterns whose support is less than ϵ . Assume the real faded sup-

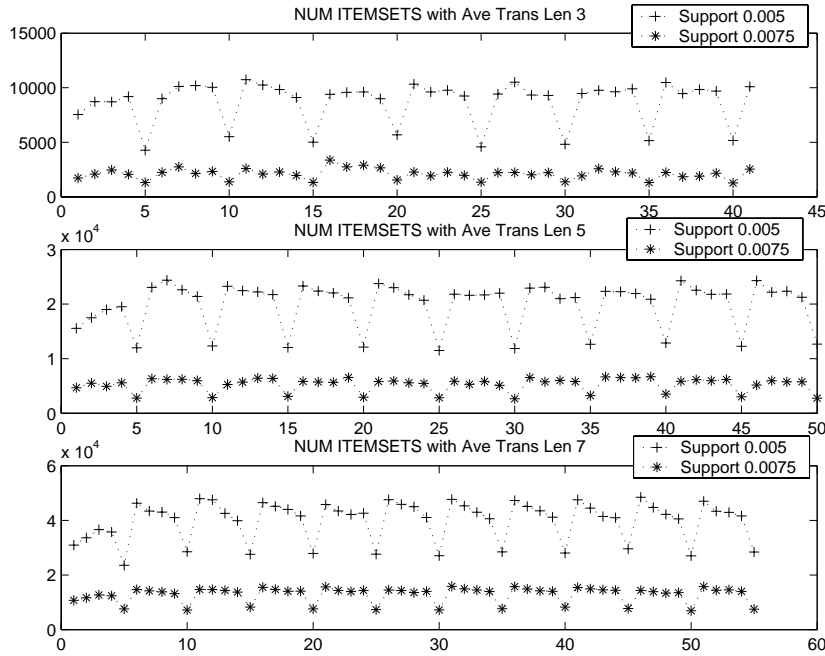


Figure 3.10: FP-stream total number of itemsets

port of I for $B(i, j)$ is $f_I = \sum_{k=j}^i \phi^{i-k} f_I(B_k)$, the approximate support we get for I is \hat{f}_I , then we have

$$f_I - \epsilon \sum_{k=j}^i \phi^{i-k} |B_k| \leq \hat{f}_I \leq f_I \quad (3.3)$$

Inequality (3.3) is consistent with inequality (3.2) if actual support is replaced with faded support and the actual window size is replaced with the faded window size. When we merge two tilted-time windows, t_i and t_{i+1} , the merged frequency is $\hat{f}_I(t_i) + \hat{f}_I(t_{i+1}) \times \phi^{l_i}$, where l_i is the number of batches contained in tiled-time window t_i . As we can see, our tilted-time window framework also works for time fading model by changing the definition of merging operation. The claims discussed before also hold for the time fading model.

3.9 Broader Stream Mining Issues

In the last few years a great deal of work has been conducted on the managing and mining of stream data (see [3] for a good survey). One of the broader issues addressed is the development of systems for processing queries on data streams. For example,

the data stream management system (DSMS) at Stanford aims to serve the analogous role of a relational DBMS on data streams. Also, the issue of stream data mining has been addressed by extending static data mining models to a stream environment: classification [7, 11], clustering [9, 16], and frequent itemset discovery [15].

Dong *et al.* [8] argue that “online mining of the changes in data streams is one of the core issues” in stream data mining and that the previously mentioned studies have not addressed this issue substantially. Dong *et al.* describe three categories of research problems: modeling and representation of changes, mining methods, and interactive exploration of changes.

Modeling and representation of changes refers to the development of query languages for specifying mining queries on changes in data streams and the development of methods of summarizing and representing the discovered changes. Mining methods refers to the development of efficient algorithms for evaluating specific change mining queries as well as general queries specified by a change mining query language. Finally, interactive exploration of changes refers to the development of methods to support a user’s evaluation of changes. For example, a user may initially want to monitor changes at a high level, then more closely inspect the details of interesting high-level changes.

We envision the **FP-stream** model as a foundation upon which frequent itemset change mining queries can be answered. For example, the change in frequency of itemsets across multiple time granularities can be computed.

Acknowledgments

The authors express their thanks to An-Hai Doan for his constructive comments on a draft of the paper. The work was supported in part by U.S. National Science Foundation (NSF) IIS-02-09199, the Univ. of Illinois, and an IBM faculty award. C. Giannella thanks the NSF for their support through grant IIS-0082407.

Bibliography

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.
- [2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.
- [3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 2002 ACM Symp. Principles of Database Systems (PODS'02)*, pages 1–16, Madison, WI, June 2002.
- [4] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.
- [5] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, pages 323–334, Hong Kong, China, Aug. 2002.
- [6] P. Dokas, L. Ertöz, V. Kumar, A. Lazarevic, J. Srivastava, and P.-N. Tan. Data mining for network intrusion detection. In *Proc. 2002 NSF Workshop on Data Mining*, pages 21–30, Baltimore, MD, Nov. 2002.
- [7] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. 2000 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'00)*, pages 71–80, Boston, MA, Aug. 2000.
- [8] G. Dong, J. Han, L. Lakshmanan, J. Pei, H. Wang, and P. Yu. Online Mining of Changes from Data Streams: Research Problems and Preliminary Results. In *Proc. 2003 ACM SIGMOD Workshop on Management and Processing of Data Streams*, 2003.
- [9] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS'00)*, pages 359–366, Redondo Beach, CA, 2000.
- [10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

- [11] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. 2001 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'01)*, San Francisco, CA, Aug. 2001.
- [12] T. Imielinski, L. Khachiyan, and A. Abdulghani. Cubegrades: Generalizing association rules. *Data Mining and Knowledge Discovery*, 6:219–258, 2002.
- [13] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 313–320, San Jose, CA, Nov. 2001.
- [14] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York, NY, Aug. 1998.
- [15] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, pages 346–357, Hong Kong, China, Aug. 2002.
- [16] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. High-performance clustering of streams and large data sets. In *Proc. 2002 Int. Conf. Data Engineering (ICDE'02)*, San Francisco, CA, April 2002.
- [17] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00)*, pages 11–20, Dallas, TX, May 2000.
- [18] H. Wang, J. Yang, W. Wang, and P. S. Yu. Clustering by pattern similarity in large data sets. In *Proc. 2002 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'02)*, pages 418–427, Madison, WI, June 2002.
- [19] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proc. 2002 SIAM Int. Conf. Data Mining*, pages 457–473, Arlington, VA, April 2002.