

Verifying and Mining Frequent Patterns from Large Windows over Data Streams

Barzan Mozafari, Hetal Thakkar, Carlo Zaniolo

Computer Science Department
University of California
Los Angeles, CA, USA

{barzan,hthakkar,zaniolo}@cs.ucla.edu

Abstract—Mining frequent itemsets from data streams has proved to be very difficult because of computational complexity and the need for real-time response. In this paper, we introduce a novel *verification* algorithm which we then use to improve the performance of monitoring and mining tasks for association rules. Thus, we propose a frequent itemset mining method for sliding windows, which is faster than the state-of-the-art methods—in fact, its running time that is nearly constant with respect to the window size entails the mining of much larger windows than it was possible before. The performance of other frequent itemset mining methods (including those on static data) can be improved likewise, by replacing their counting methods (e.g., those using hash trees) by our verification algorithm.

I. INTRODUCTION

Data streams have received much attention in recent years. Furthermore, interest in online stream mining has also dramatically increased [1], [2], [3], [4], [5], [6]. This interest is largely due to the growing set of streaming applications, such as credit card fraud detection, market basket data analysis, where data mining plays a critical role. In this paper, we focus on the problem of mining frequent itemsets on large windows defined over such data streams. This problem appears in many of the applications mentioned above in different forms.

Mining frequent itemsets for association rules has been studied extensively since it was first introduced by Agrawal et al. [1]. Since then many faster algorithms have been proposed [2], [3], [6], [7]. Furthermore, this problem appears as a subproblem in many other mining contexts such as finding sequential patterns [7], [3], clustering [8], and classification [9], [10].

The recent growth of interest in data stream systems and data stream mining is due to the fact that, in many applications, data must be processed continuously, either because of real time requirements or simply because the stream is too massive for a store-now & process-later approach. However, mining of data streams brings many challenges not encountered in database mining, because of the real-time response requirement and the presence of bursty arrivals and concept shifts (i.e., changes in the statistical properties of data). In order to cope with such challenges, the continuous stream is often divided into windows, thus reducing the size of the data that need to be stored and mined. This allows detecting concept drifts/shifts by monitoring changes between subsequent windows. Even so, association rule mining over

such large windows remains a computationally challenging problem requiring algorithms that are faster and lighter than those used on stored data. Thus, algorithms that make multiple scans of the data should be avoided in favor of single-scan, incremental algorithms. In particular, the technique of partitioning large windows into slides (a.k.a. panes) to support incremental computations has proved very valuable in DSMS [11], [12] and will be exploited in our approach.

We will also make use of the following observation: in real-world applications there is an obvious difference between the problem of (i) finding new association rules, and (ii) verifying the continuous validity of existing rules.

Normally, finding new rules requires both machines and domain experts, since size of the data is too large to be mined by a person and importance of new rules with respect to the application can only be validated by domain experts. In this situation, delays by the mining algorithms in detecting new frequent itemsets are also acceptable, provided that they add little to the typical time required by the domain experts to validate new rules. Thus, we propose an algorithm for incremental mining of frequent itemsets that compares favorably with existing algorithms when real-time response is required. Furthermore, the performance of the proposed algorithm improves when small delays are acceptable.

Although a real-time introduction of new association rules is neither sensible nor feasible, the on-line verification of old rules is highly desirable in most application scenarios: we need to determine immediately when old rules no longer hold to stop them from pestering customers with improper recommendations. Therefore, in this paper we propose fast algorithms, called *verifiers* henceforth, for verifying the frequency of previously frequent itemsets over newly arriving windows. Toward this goal, we use sliding windows, whereby a large window is partitioned into smaller panes [11] and a response is returned promptly at the end of each slide (rather than at the end of each large window). This also leads to a more efficient computation since the frequency of the itemsets in the whole window can be computed incrementally by counting itemsets in the new incoming (and old expiring) panes. Thus to make this counting efficient, we introduce a novel concept of *conditional counting*, a.k.a. verification, which can be realized efficiently by the proposed verifiers. Thus, the proposed incremental algorithm for finding frequent

itemsets utilizes the proposed verifiers to efficiently compute the frequency of a set of mined itemsets over the whole window¹. Therefore, the contributions of the paper are as follows:

1. We introduce the novel concept of conditional counting, which results in an efficient verification technique, called verifier. We propose two verifiers and a hybrid verifier, which combines and outperforms both. Our verifiers are an order of magnitude faster than current state-of-the-art counting methods (Sections IV,V).
2. We also propose a delta-maintenance based method that utilizes the fast verifier for incremental frequent itemset mining over very large sliding windows, called Sliding Window Incremental Miner(SWIM)(Section III). SWIM is an exact algorithm (with no false positives or false negatives) and achieves high performance and scalability with respect to the window size. Performance of SWIM further improves when a small fraction of the patterns are allowed to be reported in a delayed fashion. As shown in Section V, more than 99% of the patterns are reported with no delay. Furthermore, our method allows the user to control this delay, which can also be set to 0. Decreasing the delay decreases the efficiency of our method, however our method is faster than state-of-the-art methods even when the delay is set to 0.
3. Finally, we discuss other important applications that will significantly benefit from the proposed fast verifiers. An order of magnitude performance gain over hash-trees means that our verifiers can replace the counting algorithms in current mining algorithms, to improve the efficiency of such algorithms. As we discuss in Section VI, monitoring patterns and randomization-based privacy preserving algorithms are two other applications that benefit from the proposed verifiers.

The rest of this paper is organized as follows. Section II, overviews recent related work, followed by introduction of the incremental mining algorithm for large sliding windows (Section III). In Section IV we present novel verification algorithms that make the proposed incremental mining algorithm efficient. In Section V we validate our claims through comprehensive experiments over synthetic and real-world datasets followed by additional advantages and a conclusion in Section VI and Section VII.

II. RELATED WORK

Many algorithms have been proposed for mining of frequent itemsets [1], [5], [13], but because of space limitations we only discuss those that are most relevant to this paper. In fact, several of those algorithms are not suitable for data streams [14], since they require many passes through the data, or they are computationally too expensive.

Han et al. [2], introduced an efficient data structure, called *fp-tree*, for compactly storing transactions given a minimum support threshold. Then they proposed an efficient algorithm

(called FP-growth) for mining an *fp-tree* [2]. This efficient algorithm, however, requires two passes over the window (one for finding the frequent items and second for finding the frequent itemsets), and becomes prohibitively expensive for large windows. This situation has motivated a considerable amount of research interest in online frequent itemsets mining over data stream windows [15], [16], [17]. Chi et al. [17] propose the Moment algorithm for maintaining closed frequent itemsets over sliding windows, whereas Cats Tree [15] and CanTree [16] support the incremental mining of frequent itemsets.

While a significant amount of work has focused on skipping candidate generation phase [2], condense representations [18] and a variety of other optimization aspects of frequent pattern mining [1], [5], [19], there are only a few that have addressed the counting phase, often assumed to be a trivial phase. In addition to hash tree [1] which has been widely used in almost all the existing work, the most related ones to our verification approach are as follows. One category is hash-based counting methods originally proposed in Park et al. [19], whereas Brin et al. [20], proposed a dynamic algorithm, called DIC, for efficiently counting itemsets' frequencies.

The fast verifiers proposed in this paper utilize the *fp-tree* data structure and the conditionalization idea (which have been used for mining in [2]) and significantly outperform current state-of-the-art hash-based counting techniques. Achieve much faster counting of itemsets allows a much faster delta-maintenance of frequent itemsets on large sliding windows.

III. MINING LARGE SLIDING WINDOWS

SWIM (Sliding Window Incremental Miner) algorithm relies on a verifier function for which an efficient algorithm is proposed in Section IV. SWIM is an exact and efficient algorithm for mining very large sliding windows over data streams. The performance of SWIM improves when small delays are allowed in reporting new frequent patterns, however this delay can be set to 0 with a small performance overhead.

A. Problem Statement and Notations

Let D be the dataset to be mined (a window in our case); D contains several transactions (baskets), where each basket contains one or more items. Let $I = i_1, i_2, \dots, i_n$ be the set of all such distinct items in D . Each subset of I is called an itemset, and by k -itemset we mean an itemset containing k different items. The *frequency* of an itemset p is the number of transactions in D that contain itemset p , and is denoted as $\text{Count}(p, D)$. The support of p , $\text{sup}(p, D)$, is defined as its frequency divided by the total number of transactions in D . Therefore, $0 \leq \text{sup}(p, D) \leq 1$ for each itemset p . The goal of frequent itemsets mining² is to find all such itemsets p , whose support is greater than (or equal to) some given minimum support threshold α . The set of frequent itemsets in D is denoted as $\sigma_\alpha(D)$.

¹As we will see in later sections the cost of the verification drops as larger delays are allowed.

²As typically done in the literature, the words 'pattern' and 'itemset' will be used in this paper as synonyms.

Here we consider frequent itemsets mining over a data stream, thus D is defined as a sliding window over the continuous stream. D moves forward by a certain amount³ by adding the new slide (δ^+) and dropping the expired one (δ^-). Therefore, the successive instances of D are shown as W_1, W_2, \dots . The number of transactions that are added to (and removed from) each window is called its slide size. In this paper, for the purpose of simplicity, we assume that all slides have the same size, and also each window consists of the same number of slides. Thus, $n = |W|/|S|$ is the number of slides (a.k.a. panes) in each window, where $|W|$ denotes the window size and $|S|$ denotes the size of the slides.

B. The SWIM Algorithm

Sliding Window Incremental Miner (SWIM) always maintains a union of the frequent patterns of all slides in the current window W , called Pattern Tree (PT), which is guaranteed to be a superset of the frequent patterns over W . Upon arrival of a new slide and expiration of an old one, we update the true count of each pattern in PT , by considering its frequency in both the expired slide and the new slide. To assure that PT contains all patterns that are frequent in at least one of the slides of the current window $\cup_i(\sigma_\alpha(S_i))$, we must also mine the new slide and add its frequent patterns to PT . The difficulty is that when a new pattern is added to PT for the first time, its true frequency in the whole window is not known, since this pattern wasn't frequent in the previous $n-1$ slides. To address this problem, SWIM uses an auxiliary array, aux_array , for each new pattern in the new slide. The aux_array stores the frequency of a pattern in each window starting at a particular slide in the current window. In other words, the aux_array stores frequency of a pattern for each window, for which the frequency is not known. The key point is that this counting can either be done eagerly (i.e., immediately) or lazily. Under the laziest approach, we wait until a slide expires and then compute the frequency of such new patterns over this slide and update the aux_arrays accordingly. This saves many additional passes through the window. The pseudo code for the SWIM algorithm is given in Figure 1. At the end of each slide, SWIM outputs all patterns in PT whose frequency at that time is $\geq \alpha \cdot n \cdot |S|$. However we may miss a few patterns due to lack of knowledge at the time of output, but we will report them as delayed when other slides expire. The following mini-example shows how SWIM works.

Example 1: Assume that our input stream is partitioned into slides S_1, S_2, \dots and we have 3 slides in each window.

Consider a pattern p which shows up as frequent in S_4 for the first time. Letting $p.f_i$ denote the frequency of p in the i^{th} slide and $p.freq$ denote p 's cumulative frequency in the current window, SWIM works as follows:

- $W_4 = \{S_2, S_3, S_4\}$: SWIM allocates an auxiliary array for p ; $p.freq = p.f_4$ and $p.aux_array = \langle p.f_4, p.f_4 \rangle$.

³Each window either contains the same number of transactions (count-based or physical window), or contains all transactions arrived in the same period of time (time-based or logical window).

Each entry in this array will restore the partial frequencies of p in the past $n-1$ slides that we did not count p in.

- $W_5 = \{S_3, S_4, S_5\}$: S_2 expires thus the algorithm computes $p.f_2$, also adds $p.f_5$ to the cumulative count of p ; $p.freq = p.f_4 + p.f_5$. Knowing $p.f_2$ and $p.f_5$, the auxiliary array will be also updated; $p.aux_array = \langle p.f_2 + p.f_4, p.f_4 + p.f_5 \rangle$.
- $W_6 = \{S_4, S_5, S_6\}$: S_3 expires thus the algorithm computes $p.f_3$, also adds $p.f_6$ to the cumulative count of p ; $p.freq = p.f_4 + p.f_5 + p.f_6$. Also knowing $p.f_3$, the auxiliary array will be updated; $p.aux_array = \langle p.f_2 + p.f_3 + p.f_4, p.f_3 + p.f_4 + p.f_5 \rangle$. At this point, the full frequencies for p in both W_4 and W_5 are stored in this array, and in each case can be reported as delayed if turn out to be larger than the minimum support. Also from this slide on, $p.freq$ will contain the full frequency of p in the current window and can be immediately reported as frequent if needed. Thus, aux_array is no longer needed and is discarded.
- $W_7 = \{S_5, S_6, S_7\}$: S_4 expires thus the algorithm computes $p.f_4$ and deducts it from $p.freq$; $p.aux_array = \langle p.f_5 + p.f_6 + p.f_7 \rangle$. SWIM also adds $p.f_7$ to this cumulative count; $p.freq = p.f_5 + p.f_6 + p.f_7$. Subsequent windows will be treated exactly the same as W_7 . SWIM simply updates cumulative count of p until none of the 3 slides in the current window have p as frequent and then p is pruned from the Pattern Tree.

For Each New Slide S

- 1: For each pattern $p \in PT$
update $p.freq$ over S
- 2: Mine S to compute $\sigma_\alpha(S)$
- 3: For each existing pattern $p \in \sigma_\alpha(S) \cap PT$
remember S as the last slide in which p is frequent
- 4: For each new pattern $p \in \sigma_\alpha(S) \setminus PT$
 $PT \leftarrow PT \cup \{p\}$
remember S as the first slide in which p is frequent
create auxiliary array for p and start monitoring it

For Each Expiring Slide S

- 5: For each pattern $p \in PT$
update $p.freq$, if S has been counted in
update $p.aux_array$, if applicable
report p as delayed, if frequent but not reported
at query time
delete $p.aux_array$, if p has existed since arrival of S
delete p , if p no longer frequent in any of the current slides

Fig. 1. SWIM pseudo code.

C. SWIM Analysis

Correctness. This follows immediately from the fact that a pattern p belongs to $\sigma_\alpha(W)$, only if it also belongs to

$\cup_i(\sigma_\alpha(S_i))$. Thus, every frequent pattern in W must show up after mining of at least one of the slides and then we add it to PT (detailed proof in [21]).

Max Delay. The maximum delay allowed by SWIM is $n - 1$ slides. Indeed, after expiration of $n - 1$ slides, SWIM will have a complete history of the frequency of all frequent patterns of W and can report them. Moreover, the case in which a pattern is reported after $(n - 1)$ slides of time, is rare. For this to happen, pattern's support in all previous $n - 1$ slides must be less than α but very close to it, say $\alpha \cdot |S| - 1$, and suddenly its occurrence goes up in the next slide to say β , causing the total frequency over the whole window to be greater than the support threshold. Formally, this requires that $(n - 1) \cdot (\alpha \cdot |S| - 1) + \beta \geq \alpha \cdot n \cdot |S|$, which implies $\beta \geq n + \alpha \cdot |S| - 1$. This is not impossible, but in real-world such events are very rare, especially when n is a large number (i.e., a large window spanning many slides). In fact our experiments (Section V) show that most patterns are reported without any delay.

Time and Memory Complexity. From Figure 1, the main steps of SWIM are as follows. (i) Count all patterns of PT in the new slide and the expired one (delta maintenance), lines 1 and 5, and (ii) Compute $\sigma_\alpha(S_{new})$ and insert them into PT , line 2. Lines 3 and 4 are for book-keeping purposes and they can be performed concurrently with line 2. Let us denote the average time for (i) as a function of $|PT|$ and slide size $|S|$, $f(|S|, |PT|)$, and that for (ii) as a function of the slide size and α , $M(|S|, \alpha)$. Then the total time to process each window will be (roughly) $2 \cdot f(|S|, |PT|) + M(|S|, \alpha)$. Interestingly, the only part of this running time that depends on window size ($|W|$) is $|PT|$. But as we show in Section V, $|\cup_i(\sigma_\alpha(S_i))|$ is significantly smaller than $n \cdot |\sigma_\alpha(S_i)|$, since most frequent patterns are common between several slides.

The memory usage, consists of an *fp-tree* containing the new slide⁴, and the pattern tree containing $\cup_i(\sigma_\alpha(S_i))$, which is significantly smaller than $n \cdot |\sigma_\alpha(S_i)|$, as discussed above. The only concern which remains is that we need an auxiliary array (of size $n - 1$) for each pattern which has been added to PT recently (i.e., within the last n slides). After that period we no longer need an auxiliary array for that pattern and we release its memory. Therefore, the worst case happens when all patterns need such an array resulting in $4 \cdot n \cdot |PT|$ bytes of extra memory (assuming we use 4-byte integers for storing the frequency); this is not prohibitive since the number of patterns is not supposed to be very large⁵. However, based on our experiments only 60% of the patterns require *aux_array* at any given point of time, e.g. for $n = 1000$ slides in each W , even when $|PT| = 10000$, we will only need *24MB* of extra memory (for 60% of the patterns) on the average and *40MB* (for all the patterns) in the worst case.

⁴In window-based streams, the current window is stored somewhere on disk or in memory in order to expire old slides. In either case, we can store/fetch each slide in *fp-tree* format.

⁵Indeed, having a very low minimum support results in huge number of patterns which are usually meaningless in many real-world applications where the rules should be approved by an expert.

D. SWIM with Adjusted Delay Bound

SWIM can be easily modified to only allow a given maximum delay of L slides ($0 \leq L \leq n - 1$) as described next. Upon arrival of each slide of W , besides verifying the frequency of each pattern of PT over the new slide (S_{new}) and the expired one (S_{new-n}), SWIM will also verify the frequency of the new patterns ($p \in \sigma_\alpha(S_{new}) \setminus PT$) over previous $(n - L - 1)$ slides, i.e. $S_{new-1}, \dots, S_{new-n+L+1}$. Thus, the Maximum delay of reporting a frequent pattern in SWIM(Delay= L) is at most L slides. Notice that after expiring L slides, we have counted the occurrence of any frequent pattern of W in all n slides as remaining slides were counted eagerly. Note, choosing $L = 0$ guarantees that all frequent patterns are reported immediately once they become frequent in W and choosing $L = n - 1$ leads to the lazy SWIM in Section III-B.

While SWIM(Delay= L) represents an efficient incremental mining algorithm, counting frequencies of itemsets over a given dataset ($n - L + 1$ slides in our case) remains a bottleneck. Therefore, faster algorithms are required to compute these counts efficiently. Thus, next we propose a new algorithm, called verifier, based on the novel concept of conditional counting, that can be seen as a more general form of counting. As shown in Section V the proposed algorithm outperforms state-of-the-art counting algorithms.

IV. VERIFICATION

In this section, we formally define verification and propose two novel verifiers and their hybrid version.

Definition 1: Let D be a transactional database, P be a given set of arbitrary patterns and *min_freq* a given minimum frequency. A function f is called a *verifier* if it takes D , P and *min_freq* as input and for each $p \in P$ returns one of the following: (i) p 's true frequency in D if it has occurred at least *min_freq* times or otherwise (ii) reports that it has occurred less than *min_freq* times (frequency not required in this case).

It is important to notice the subtle difference between verification and simple counting. In the special case of *min_freq* = 0 a verifier simply counts the frequency of all $p \in P$, but in general if *min_freq* > 0, the verifier can skip any pattern whose frequency will be less than *min_freq*. This early pruning can be done by the Apriori property or by visiting more than $|D| - \text{min_freq}$ transactions. Also, note that verification is different (and weaker) from mining. In mining the goal is to 'find' all those patterns whose frequency is at least *min_freq*, but verification simply verifies counts for a given set of patterns, i.e. verification does not discover additional patterns. Therefore, we can consider verification as a concept more general than counting, and different from (weaker than) mining. The challenge is to find a verification algorithm, which is faster than both mining and counting algorithms, since algorithms like SWIM will benefit from this efficiency.

A. Background

Verifiers proposed here use the *fp-tree* data structure [2] (with some modifications). Here we assume that the reader is familiar with this data structure [2] and simply discuss the differences.

The *fp-tree* data structure as proposed in [2] orders the items in a transaction, in decreasing order of their frequency, instead we simply order the items in a lexicographical order, since this avoids an additional pass over the data. We also maintain a header table, as discussed in [2]. As an example, Figure 3 (a) shows an *fp-tree* created from the transactional database of figure 2. Furthermore, we also utilize the concept of conditionalization, as discussed in [2]. In summary, conditionalizing an *fp-tree* with respect to a given item, results in another *fp-tree*, which only contains transactions that 'end' with the conditionalization item. Furthermore, this resulting *fp-tree* can be recursively conditionalized for other items. Figure 3 shows successive iterations of this process, which results in computing the frequency of pattern *gdb* (conditionalizing for *g*, followed by *d*, followed by *b*). Given this background, we use the following notation to describe our verifiers:

- *u.item* is the item represented by node *u*.
- *u.freq* is the frequency of *u*, or *NIL* when unknown.
- *u.children* represents the set of *u*'s children .
- *head(c)* is the set of all nodes holding item *c*.

We also use another data structure called pattern tree, which is just an *fp-tree*, but instead of DB transactions we insert patterns in it. Thus each node represents a unique pattern. A verifier algorithm computes the frequency of all patterns in a given pattern tree. Initially the frequency of each node in the pattern tree is zero, but after the verification it would contain the true frequency of the pattern it represents.

B. Double-Tree Verifier (DTV)

In this section we introduce our first verification algorithm, called Double-Tree Verifier (DTV). DTV first inserts all transactions into an *fp-tree*. Thus, we have two trees, a given pattern tree and the *fp-tree*. We utilize the conditionalization technique introduced in [2] to recursively count frequency of itemsets.

The DTV algorithm conditionalizes both the *fp-tree* and the pattern tree, in parallel, as shown in Figure 4. Performing this process for both *fp-tree* and pattern tree at the same time, allows us to prune many useless paths as follows. First items

TID	Items bought	Ordered chosen items
100	a,b,c,d,e,x,y,z	a,b,c,d,e
200	a,b,c,d,f,x,y,z	a,b,c,d,f
300	a,b,c,d,g,z	a,b,c,d,g
400	a,b,c,d,g,p	a,b,c,d,g
500	b,e,g,h,z	b,e,g,h
600	a,b,c,g	a,b,c,g

Fig. 2. A transactional database.

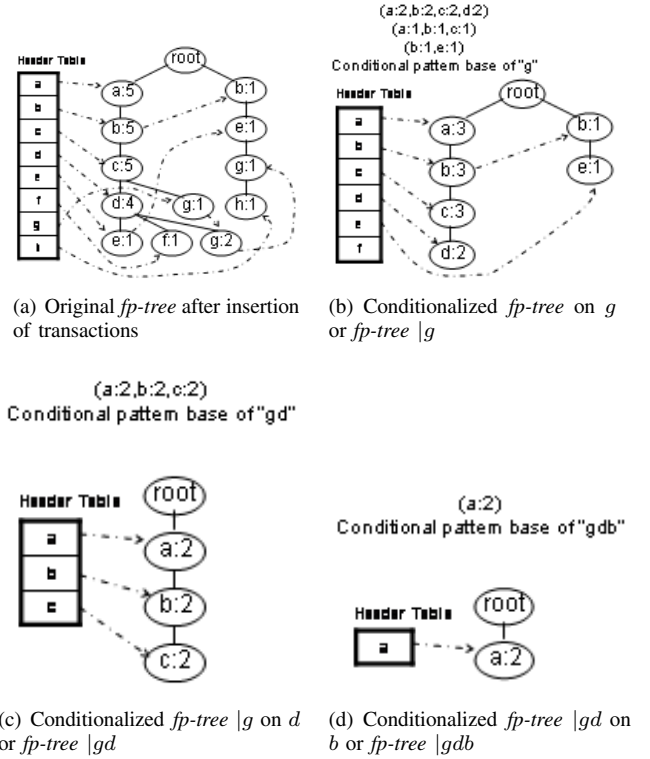


Fig. 3. A small *fp-tree* and its successive conditionalizations.

not present in conditionalized pattern tree can be pruned from the *fp-tree* (line 4, Figure 4). Secondly, the items not frequent in the *fp-tree* can be pruned from the pattern tree (line 6). Therefore, the next recursive conditionalization step will be performed over the pruned trees (line 7), resulting in significant computational reduction. The algorithm continuously updates itemset frequencies and book keeping pointers, which is omitted from the pseudo code for brevity. Next we present a simple example that illustrates this process.

Example 2: Again consider the database of Figure 2, which was inserted in an *fp-tree* in Figure (3.a). Let the initial pattern tree to be as given in Figure (5.a).

Let's say while running DTV, we decide to conditionalize based on item *g*. The resulting conditionalized pattern tree on *g* is shown in Figure (5.b). Notice that all frequencies in both the original pattern tree and the conditionalized tree are still unknown. Also notice that we keep a pointer between corresponding nodes (shown as solid double arrows), during the conditionalization process.

At this point, we conditionalize the *fp-tree* accordingly (on the same item) and obtain a conditionalized *fp-tree* (Figure (3.b)). Inductively, we can verify this smaller pattern tree against that smaller *fp-tree* shown in Figure (5.c). Now inferring the frequencies of *g* nodes in the original tree is straightforward, i.e. following back our return pointers. In Figure (3.c) these two nodes represent patterns *g* and *bdg* of the original pattern tree.

Correctness and Running Time. The correctness of DTV can be proved by induction on the total number of nodes in

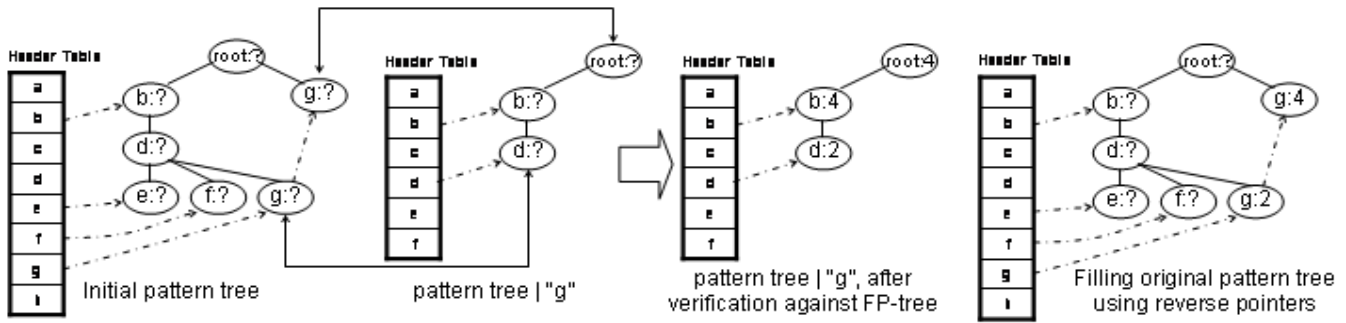


Fig. 5. An example of running DTV.

Algorithm DTV(min_freq, FP, PT)

Input:

min_freq is the minimum frequency threshold;
 FP is an fp -tree containing the DB transactions
 PT is a pattern tree containing given patterns

Description:

- 1: For all items $x \in PT.head$
- 2: $PT_x \leftarrow PT|x$ with no pruning
- 3: If $PT_x \neq \phi$
- 4: $FP_x \leftarrow FP|x$ with pruning all items not in PT_x or those whose frequency $< min_freq$
- 5: If $FP_x \neq \phi$
- 6: In PT_x prune all items not appearing in FP_x
- 7: DTV(min_freq, FP_x, PT_x)

Fig. 4. DTV pseudo code.

both fp -tree and pattern tree⁶.

Theoretically, the running time of DTV could be exponential in the worst case due to exponential nature of frequent pattern mining. But in practice, number of frequent patterns is small when min_freq is not very small. Therefore, DTV's running time compares favorably to the state-of-the-art frequent itemset mining algorithm (FP-growth).

Lemma 1: For a given fp -tree T and a fixed min_freq , let X be the set of conditionalizations performed by FP-growth to mine T and let Y be the set of conditionalizations performed by DTV in order to verify the frequent patterns of T . Now there exists an injective mapping Ψ from Y to X s.t. for each $x \in X$, the length of x is smaller than or equal to that of $\Psi(x)$.

Here the length of a conditionalization means the number of items that we have conditionalized, over from the original tree, until it becomes a single node. As an immediate result of having an injective mapping, it is clear that $|Y| < |X|$ holds too. In a nutshell, this lemma tells us that DTV algorithm under aforementioned assumptions, is guaranteed to be faster

⁶For space constraints, all the proofs are omitted in this paper but can be found in [21].

by having to perform less conditionalizations⁷. Notice that the number of different paths in the execution tree of DTV (corresponding to conditionalizations) is bounded by the total number of different patterns in the pattern tree. But for FP-growth as min_freq goes down the number of different possibilities for conditionalization goes up as we can prune fewer items in each step. In fact, for very small min_freq values, it becomes impossible to run FP-growth due to the exponential number of paths. Thus, the advantage of DTV increases when the minimum support decreases.

C. Depth-First Verifier (DFV)

The second algorithm we propose is called DFV. Similar to DTV, all transactions are initially inserted in an fp -tree and a pattern tree is given. The algorithm then traverses the given pattern tree in a depth first order and performs the following steps for each pattern tree node c to find the frequency of the pattern p represented by c . Notice that the only transactions which may contain p are those with a prefix ending in c 's item. Therefore clearly, by following the header table of the fp -tree for c 's item, we can reach all such candidate nodes. For each candidate node, s , we can easily determine if its represented transaction contains the pattern p , if the path from the root node to the candidate node s contains all items in p . Instead of this naive search in the fp -tree for each node (i.e., each pattern) of the pattern tree, DFV exploits the following optimizations:

- 1) **Ancestor Failure.** If a path in the fp -tree has already proved to not contain a prefix of the pattern p , then we know that it does not contain p itself either (apriori property). Therefore, DFV remembers such fp -tree nodes for any of the ancestors of c , while processing node c , which can then be marked.
- 2) **Smaller Sibling Equivalence.** If a path in the fp -tree has already been marked to (or not to) contain a smaller sibling of the pattern p , then we know that it does (or does not) contain p itself too. The reason is that sibling patterns are only different in their last item. Therefore, if a path contains the smaller sibling, it also contains

⁷In this lemma we have ignored the extra time to recursively conditionalize the pattern tree in DTV knowing that the number of patterns is much smaller than the size of the original database.

p itself provided that the path starts with c 's item. Similarly, since the items in the fp -tree are in order, if a prefix does not contain the smaller sibling it can not contain the larger sibling either.

- 3) **Parent Success.** If a path in the fp -tree has already been marked to contain the parent pattern of p (i.e., the pattern associated with node c), then we know that it also contains p , provided that this path starts with the c 's item.

DFV utilizes all of the above optimizations by marking/unmarking the fp -tree nodes in a depth-first order and processing children of each node in increasing order of items. Thus, while we are processing a pattern node c , all fp -tree nodes visited by any of its ancestors or by any of its smaller siblings, are already marked, thus allowing to avoid unnecessary traversals of the fp -tree, see the pseudo code for more details.

Algorithm DFV(pattern tree PT , fp -tree FP , min_freq)

```

1: FP.root.mark ← FAIL
2: For all children  $c$  of  $PT.root$  in ascending order
3:   For all nodes  $s$  in  $FP.head(c.item)$ 
4:     s.mark ← OK
5:     c.freq ← c.freq + s.freq
6:   If c.freq ≥ min_freq
7:     ProcessNode(c, min_freq, FP)

```

ProcessNode(pattern node u , min_freq , fp -tree FP)

```

1: For all children  $c$  of  $u$  in ascending order
2:   For all nodes  $s$  in  $FP.head(c.item)$ 
3:     Find  $t$ , the ‘smallest decisive ancestor’ of  $s$  w.r.t.  $u$ 
4:     If  $t.item < u.item$ 
5:       s.mark ← FAIL
6:     Else
7:       s.mark ← t.mark
8:       if s.mark = OK
9:         c.freq ← c.freq + s.freq
10:   If c.freq ≥ min_freq
11:     ProcessNode(c, min_freq, FP)
12: For all children  $c$  of  $u$ 
    For all FP-node  $s$  in  $FP.head(c.item)$ 
      s.mark ← NIL

```

Fig. 6. DFV pseudo code.

Correctness. The following lemma indicates how to decide a node's mark based on one of its ancestors.

Lemma 2: In DFV, when processing fp -tree node s containing $c.item$, for any ancestor t of s , the following facts about t 's mark hold (u is c 's parent):

- When $t.item < u.item$, regardless of t 's mark, if we haven't visited any other node between s and t containing $u.item$, it is guaranteed that there is not such a node in t 's ancestors too (line 5 in **ProcessNode**).

- If $t.item > u.item$ and $t.mark$ is not NIL then u must have another child p such that $p.item = t.item$. Also $p.item < c.item$.

As mentioned before, we can find s 's mark by following all its ancestors up to the root. However, the above lemma permits us to stop as soon as we visit a node t that meets the following criteria, called ‘smallest decisive ancestor’, which we summarize in the following definition.

Definition 2 (smallest decisive ancestor): For a given pattern node u and an fp -tree's node s , smallest decisive ancestor of s is its lowest ancestor t for which either $t.item < u.item$ or $t.item \neq NIL$.

Running Time. DFV processes each pattern tree node and the corresponding fp -tree nodes for that item. For each such fp -tree node, s , DFV goes up in the fp -tree only until it finds s 's ‘smallest decisive ancestor’, t , which is found at most after $f(s)$ (s 's depth in the tree) steps. Let the average distance of such a t to each fp -tree's node labeled with x be $f(x)$. Also assume that in the pattern tree there are $q(x)$ different nodes labeled with x , i.e. the number of patterns in which x has appeared. Therefore the total time complexity will be:

$$\sum_{x=1}^n q(x) \cdot f(x) \cdot |FP.head(x)|$$

where n is the total number of items and $|FP.head(x)|$ is the number of nodes in the fp -tree containing item x . Assuming that \tilde{f} and \tilde{q} are the average of f and q , respectively, and Z denotes the total number of nodes in the fp -tree, the above formula simplifies to:

$$\sum_{x=1}^n q(x) \cdot f(x) \cdot |FP.head(x)| = \tilde{q} \cdot \tilde{f} \cdot Z$$

By a worst-case estimation of \tilde{f} using the fact that $\tilde{f} \leq T$ where T is the average transaction length, the final running time will be $O(\tilde{q} \cdot T \cdot Z)$ and since for most configurations $\tilde{q} < 40$ and $5 < T < 20$, our running time is not prohibitive in terms of the given fp -tree size, Z (which is also a linear function of the original database size $|W|$, verified through our experiments).

D. Hybrid Version of Our Verifiers

According to experimental evidences, DTV is faster than DFV when there are many transactions in the fp -tree and many patterns in the pattern tree (Section V). However, when our trees are small, DFV is more efficient because conditionalization overhead is high. To exploit this fact, we can start with DTV until the conditionalized trees are ‘small enough’ and after that point switch to DFV. We can check the size of FP_x and PT_x and decide whether to call DTV or DFV. The best threshold for deciding on ‘small enough’ trees can be evaluated by experiments. In our experiments we have switched to DFV after second recursive call to DTV and our experiments show that this hybrid verifier is indeed faster than both of DTV and DFV, since it avoids unnecessary recursive calls when the trees are small enough.

Throughout our comparisons in Section V, we use this hybrid verifier unless DTV or DFV are explicitly mentioned.

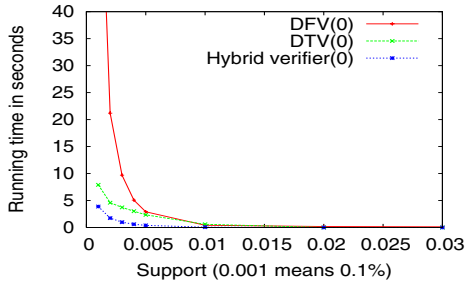


Fig. 7. Performance comparison of DFV, DTV, and hybrid verifier.

V. EXPERIMENTS

The goal of our experiments is to compare the proposed verifier and the incremental mining algorithm with their state-of-the-art counterparts.

All experiments were conducted on a P4 machine running Linux, with 1GB of RAM. Our algorithm was implemented⁸ in C. We use the IBM QUEST data generator[1] and Kosarak real-world dataset[22] for our experiments. The IBM dataset names, describe the data characteristics, where T is average transaction length, I is average pattern length, and D signifies the number of transactions.

A. Efficiency of Verification Algorithms

First, we compare the performance of DFV, DTV, and the hybrid verifier. The hybrid verifier initially uses the double tree verifier(DTV) and at runtime decides to switch to the depth-first verifier(DFV), as DFV is faster for small pattern trees. We test the efficiency of our algorithms with different support thresholds. As shown in Figure 7, the hybrid verifier is indeed faster than the two base verifiers; an order of magnitude faster when the support threshold is low, i.e. when there are many qualifying patterns. For support thresholds greater than 1%, the number of frequent patterns is low (< 400), thus all three verifiers perform about the same.

Next, we compare the proposed hybrid verifier with a state-of-the-art counting algorithm, namely hash-tree based algorithm, proposed in [1]. For this experiment, both algorithms, hybrid verifier and hash-tree based algorithm⁹, are provided a predefined set of patterns to verify over a given dataset (T20I5D50K). We vary the number of patterns given to the algorithms. As shown in Figure 8, the hybrid verifier outperforms the hash-tree based algorithm by an order of magnitude. Note Figure 8 uses log-scale for the Y-axis and the running time of the hybrid verifier includes the time to generate an *fp-tree* from the given dataset. This implies that our verifier can replace state-of-the-art hash-tree based counting algorithm and improve the performance of existing frequent itemsets mining algorithms.

Finally, we compare our hybrid verifier with a state-of-the-art mining algorithm, namely FP-growth. Again we use the

⁸All the implementations can be found at: <http://wis.cs.ucla.edu/swim/index.htm>

⁹The hash-tree based algorithm is implemented using hash_maps available in C++ standard template library.

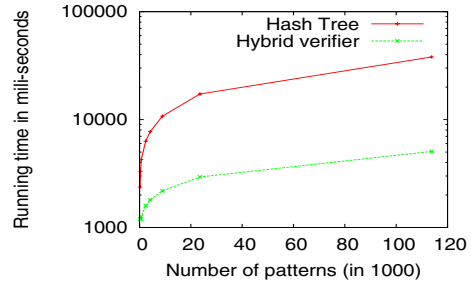


Fig. 8. Hybrid versus hash-tree verifiers

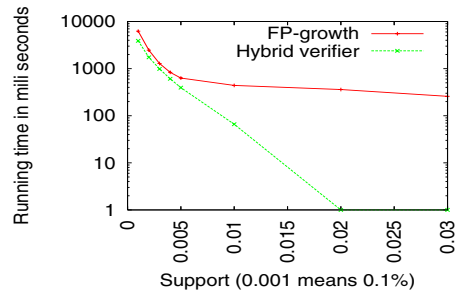


Fig. 9. Hybrid verifier versus FP-growth

T20I5D50K dataset and vary the support threshold. As shown in Figure 9, the hybrid verifier achieves better performance compared to FP-growth. Here we use 50K transactions as the window size. Note, the number of frequent patterns for support 0.5%, 1%, 2%, and 3% are 2400, 685, 384, and 217, respectively. While mining frequent itemsets does more than simple verification of patterns, this experiment simply shows that verification is faster than mining. Therefore, verification based approximate solutions are more suitable for data stream mining, since frequent itemsets mining is more expensive.

B. Efficiency of SWIM Algorithm

Our second set of experiments focus on the proposed incremental mining algorithm for large sliding windows. First, we compare our sliding window algorithm with Moment [17], a state-of-the-art incremental mining algorithm. We use the T20I5D1000K dataset and fix the window size to 10K transactions. Furthermore, we set the support threshold to 1% and vary the slide size to measure the scalability of these algorithms. As shown in Figure 10, SWIM is much more scalable compared to the Moment algorithm. In fact, both versions of our algorithm, one with maximum window size delay and one without any delay, are much faster than Moment. The Moment algorithm is intended for incremental maintenance of frequent patterns, but is not suitable for batch processing of thousands of tuples. The proposed algorithm however is aimed at maintaining frequent itemsets over large sliding windows. In fact, the proposed algorithm handles slide size of up to 1 million transactions. As shown in Figure 11, SWIM easily handles slide size of 10K transactions, whereas CanTree [16], a state-of-the-art algorithm shown to be faster than FELINE [15], AFPIM [23], and Cats Tree [15], struggles to handle such large slide size.

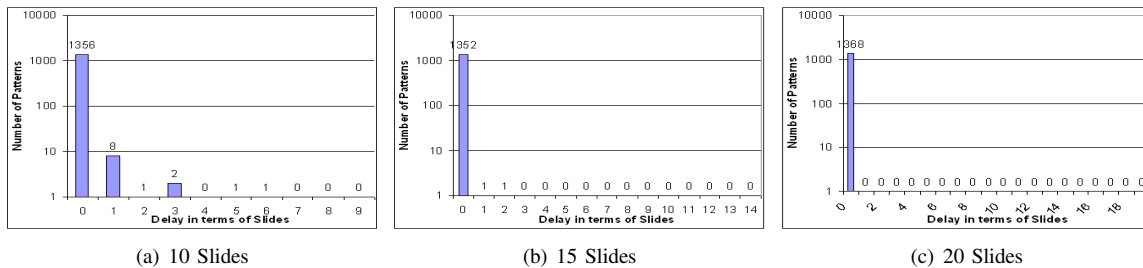


Fig. 12. Number of patterns experiencing the delay

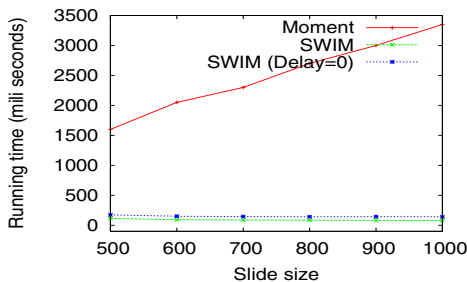


Fig. 10. Comparing SWIM and Moment

Next, we show that the delta-maintenance based approach presented here is scalable with respect to the window size. In fact, for a given slide size and support threshold, the running time of our proposed algorithm is independent of the window size. While this is intuitive given that the algorithm is based on delta-maintenance, we show that through a simple experiment here. For this experiment, we use the T20I5D1000K dataset. We fix the support threshold to 0.5% and the slide size is set to 10K transactions. We vary the window size from 20K to 400K transactions. Figure 11, shows that the running time of our algorithm is almost constant (the X-axis is in log-scale), whereas state-of-the-art algorithm, namely CanTree, does not scale well. Therefore, our algorithm is scalable with respect to window size and allows mining of very large windows, which was not possible before.

While our approach achieves greater performance compared to other incremental mining algorithms, it may also report frequent itemsets with a delay, bounded by the window size. While we allow the user to specify a maximum delay in lieu of increased processing time, in this experiment, we

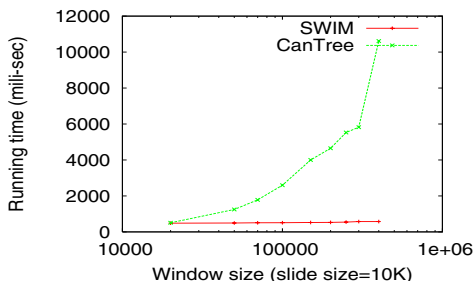


Fig. 11. Comparing with CanTree on various window sizes

study the percentage of itemsets that may be reported with a delay. We use the Kosarak dataset with window size 100K transactions and calculate average delay and the number of patterns that experience this delay. As shown in Figure 12(a), most patterns (> 99%) do not experience any delay, where X-axis denotes the delay in terms of number of slides and the Y-axis shows the number of patterns experiencing this delay (log-scale). This trend holds for different datasets, window sizes, slide sizes, and support thresholds, combinations, supporting the discussion in Section III-C.

Furthermore, as the number of slides in a window increases number of patterns experiencing the delay and the average delay reduces, as seen in Figures 12(a), 12(b), 12(c). This presents an additional advantage, since we process large windows spanning many slides, thus number of patterns reported in a delayed manner will be minimal. Thus, the proposed approach is efficient and only reports a small fraction of items in a delayed manner. Furthermore, a maximal bound over this delay can also be specified (for more analysis refer to Section III-D).

VI. ADDITIONAL ADVANTAGES OF THE VERIFIER

In this section, we outline other promising application scenarios for the proposed fast verifier.

A. Improving Existing Algorithms

As shown in Section V-A, our verifier is faster than state-of-the-art counting algorithms. Therefore, frequent itemset mining algorithms that use existing counting algorithms, can be improved by utilizing our verifier. For instance, Toivonen [6] proposes a sampling-based approach for frequent itemset mining over large databases, which mines a small sample of the dataset and then verifies the discovered patterns over the whole dataset using a hash-tree based counting algorithm. Our verifier is faster than the hash-tree based counting algorithm used by Toivonen and will thus improve the performance of that algorithm. Similarly, performance of Agrawal et al. [1], Zaki et al [5], and Park et al. [19] can also be improved, since they also use a hash-tree based counting algorithm.

B. Concept Shift

In many practical situations, including those where data arrival rate is very high, continuously mining the data set is either impractical or unfeasible. For such cases, we propose an approach whereby the data stream is monitored continuously

to (i) confirm the validity of existing patterns (using our fast verifiers), and (ii) detect any occurrence of concept-shift. In fact, our experiments suggest that concept-shift is always associated with a significant number ($> 5\text{-}10\%$) of frequent patterns becoming infrequent. Therefore, only when such changes are observed we need to call on a mining algorithm, which will discover the new patterns.

C. Privacy Preservation

We are also exploring applications of our verifiers to privacy preservation. Many privacy preserving association rule mining methods are based on data distortion [24]. In those methods, usually a randomization operator is applied to each original transaction by inserting lots of random false items and also removing some of the original ones [24]. An important bottleneck in such methods is the fact that the size of each randomized transaction is comparable to the overall number of single items, which may be few thousands: thus performance becomes a very serious concern [25]. We are currently investigating the possibility to mine such transactions by Toivonen method [6] enhanced by our DTV (or using DTV by itself to monitor patterns over those randomized transactions). Notice that since traditional hash trees [1] need to consider all subsets of each transaction in order to find the included patterns, their running time will increase exponentially with the transaction length. Such an approach can instead use DTV, exploiting the following lemma (proof in [21]).

Lemma 3: The maximum depth of the recursion in DTV (i.e., the largest length of the conditionalizations) is less than or equal to the maximum length of the patterns to be verified.

Since recursive conditionalizations count for the major part of the running time in DTV, lemma 3 implies that our running time depends almost exclusively on the pattern length, and thus it is independent of the transaction length.

VII. CONCLUSION

Mining data streams for association rules has proven to be a difficult problem, since techniques developed to mine frequent patterns on stored data result in excessive costs and time delays. This paper has made two important contributions to the solution of this problem. The first is the introduction of a very fast algorithm to verify the frequency of a given set of patterns. In fact, our algorithm outperforms the existing state-of-the-art counting algorithms by an order of magnitude. The second contribution is to use our fast verifier to solve the association-rule mining problem under the realistic assumption that we are mostly interested in the new/expiring patterns. This delta-maintenance approach effectively mines very large windows with slides, which was not possible before.

However, we also explored a second approach that further improves the performance by simply allowing a small reporting delay. Clearly this approach would become desirable in situations where modest delays in reporting new itemsets are acceptable. Such delays are negligible when compared to the time needed for the experts to validate the new rules before they are actually put into use. In summary we have proposed

an approach of great efficiency, flexibility, and scalability to solve the frequent pattern mining problem on data streams with very large windows.

VIII. ACKNOWLEDGMENTS

This work was supported in part by NSF-IIS award 0742267, “SGER: Efficient Support for Mining Queries in Data Stream Management Systems”.

REFERENCES

- [1] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *VLDB*, 1994.
- [2] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” in *SIGMOD*, 2000.
- [3] R. Srikant and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements,” in *EDBT*, 1996.
- [4] J. Yu, Z. Chong, H. Lu, Z. Zhang, and A. Zhou, “A false negative approach to mining frequent itemsets from high speed transactional data streams,” *Inf. Sci.*, 2006.
- [5] M. J. Zaki and C. Hsiao, “CHARM: An efficient algorithm for closed itemset mining,” 2002.
- [6] H. Toivonen, “Sampling large databases for association rules,” in *VLDB*, 1996, pp. 134–145.
- [7] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, “Mining sequential patterns by pattern-growth: The PrefixSpan approach,” *IEEE TKDE*, vol. 16, no. 11, pp. 1424–1440, November 2004.
- [8] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, “Automatic subspace clustering of high dimensional data for data mining applications,” in *SIGMOD*, 1998.
- [9] B. Liu, W. Hsu, and Y. Ma, “Integrating classification and association rule mining,” in *KDDM*, 1998.
- [10] K. Wang, S. Zhou, and S. C. Liew, “Building hierarchical classifiers using class proximity,” in *VLDB*, 1999, pp. 363–374.
- [11] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker, “No pane, no gain: efficient evaluation of sliding-window aggregates over data streams,” *SIGMOD Record*, 2005.
- [12] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo, “A data stream language and system designed for power and extensibility,” in *CIKM*, 2006, pp. 337–346.
- [13] N. Jiang and L. Gruenwald, “Cfi-stream: mining closed frequent itemsets in data streams,” in *SIGKDD*, 2006.
- [14] N. Jiang and L. Gruenwald, “Research issues in data stream association rule mining,” *SIGMOD Record*, 2006.
- [15] W. Cheung and O. R. Zaiane, “Incremental mining of frequent patterns without candidate generation or support,” in *DEAS*, 2003. [Online]. Available: citeseer.ist.psu.edu/622682.html
- [16] C.-S. Leung, Q. Khan, and T. Hoque, “Cantree: A tree structure for efficient incremental mining of frequent patterns,” in *ICDM*, 2005.
- [17] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, “Moment: Maintaining closed frequent itemsets over a stream sliding window,” in *Proceedings of the 2004 IEEE International Conference on Data Mining (ICDM'04)*, November 2004.
- [18] J. Pei, J. Han, and R. Mao, “CLOSET: An efficient algorithm for mining frequent closed itemsets,” in *SIGMOD*, 2000.
- [19] J. S. Park, M.-S. Chen, and P. S. Yu, “An effective hash-based algorithm for mining association rules,” in *SIGMOD*, 1995, pp. 175–186.
- [20] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, “Dynamic itemset counting and implication rules for market basket data,” in *SIGMOD*, 1997, pp. 255–264.
- [21] B. Mozafari, H. Thakkar, and C. Zaniolo, “Verifying and mining frequent patterns from large windows over data streams,” University of California Los Angeles, Tech. Rep. 070021, 2007.
- [22] “Frequent itemset mining dataset repository, <http://fimi.cs.helsinki.fi/data/>” [Online]. Available: <http://fimi.cs.helsinki.fi/data/>
- [23] J. Koh and S. Shieh, “An efficient approach for maintaining association rules based on adjusting fp-tree structures,” in *DASFAA*, 2004.
- [24] A. Evfimievski, J. Gehrke, and R. Srikant, “Limiting privacy breaches in privacy preserving data mining,” in *PODS*, 2003.
- [25] S. Agrawal, J. Krishnan, and J. Haritsa, “On addressing efficiency concerns in privacy-preserving mining,” in *DASFAA*, 2004, pp. 113–124.