Mining Frequent Itemsets Over Tuple-evolving Data Streams

Chongsheng Zhang
Henan University
Kaifeng, China
first.last@yahoo.com

Mirjana Mazuran
Politecnico di Milano
Milano,Italy
mazuran@elet.polimi.it

Hamid Mousavi
UCLA
Los Angeles,USA
hmousavi@cs.ucla.edu

Yuan Hao
University of Science and Technology Beijing
haoyuan227@gmail.com

Carlo Zaniolo
UCLA
Los Angeles,USA
zaniolo@cs.ucla.edu

Florent Masseglia
Zenith team, INRIA
Montpellier, France
first.last@inria.fr

ABSTRACT

In many data streaming applications today, tuples inside the streams may get revised over time. This type of data stream brings new issues and challenges to the data mining tasks. We present a theoretical analysis for mining frequent itemsets from sliding windows over such data. We define conditions that determine whether an infrequent itemset will become frequent when some existing tuples inside the streams have been updated. We design simple but effective structures for managing both the evolving tuples and the candidate frequent itemsets. Moreover, we provide a novel verification method that efficiently computes the counts of candidate itemsets. Experiments on real-world datasets show the efficiency and effectiveness of our proposed method.

1. INTRODUCTION

The number of applications that need to process data streams has been growing rapidly in the past few years. In these applications, transactions (tuples) are continuously arriving to the systems and users often expect real-time answers to their queries [2]. For accelerating the processing of data streams several data stream models have been proposed to approximately represent the unbounded sequence of streaming tuples and among them sliding windows [7] are commonly employed. In data stream management (and/or mining) systems (DSMSs) such as *SMM* [16], pane (a.k.a. slide) based sliding windows proved efficient and effective for query processing [12] and frequent pattern discovery [14].

Frequent pattern/itemset mining is one of the most important research issues in data mining with applications in market-basket analysis, recommender systems and so on. Mining frequent itemsets from sliding windows over streaming data has also received significant research focus [5, 14].

However, it has not been investigated on how to extract frequent itemsets from "tuple-evolving" data streams where tuples (transactions) are allowed to be revised. Here, "tuple-evolving" data streams, also called "itemset-evolving" data streams or data streams with evolving tuples hereafter, refer to streams in which data elements (i.e. tuples or transactions) can evolve/change over time. Esther Ryvkina at al. pointed out in [15] that, "......Many data stream sources (e.g., commercial ticker feeds) issue "revision tuples" (revisions) that amend previously issued tuples......".

We investigate the problem of online frequent itemset mining from sliding windows over tuple-evolving data streams. Since tuples can evolve inside this kind of data streams, the mining task is more difficult than in the case of traditional streaming data. The following example will illustrate these challenges. Prior to this, we define a pane (slide) based sliding window [12, 14] as a sliding window W which is partitioned into M ($M \geq 1$) consequent non-overlapping panes (slides), with each pane containing one or more tuples. Let p be a pattern in W, the support ratio of p is defined as the frequency of p in W divided by $|W|$ which is the total number of data elements in W. It should be noted that if a tuple has been revised one or more times in the same window, when calculating p we should only count these tuples (the tuple and the corresponding revision tuples) once in $|W|$. The minimum support ratio is a support ratio threshold for p to become frequent in W.

EXAMPLE 1. *Consider the watch/bid lists of users in an online auction site*[1]*, where customers watch or place bids on items that they are interested in. Users at any time can add to or remove items from their lists. In addition, items automatically expire from their watch lists when the auctions are over. Figure 1 shows an example in which* $users = \{u_1, u_2, ..., u_m\}$ *(m is the total number of users),* $items = \{A, B, C, D, E, F\}$ *and each user is associated with a tuple that contains the items in her/his list. The goal is to find frequent itemsets with windows of size* 12 *and slides of size* 4*, and a minimum support ratio of* 0.4.

Figure 1 shows two different approaches to managing the "tuple-evolving" data streams. The lower row in this figure considers the data streams with evolving tuples. If tuple T_1 *arrived before* T_2 *but they share the same user* u_i*, then* T_2 *is regarded as the new watch list of* u_i*. Meanwhile, since* T_1

[1]http://www.ebid.net/

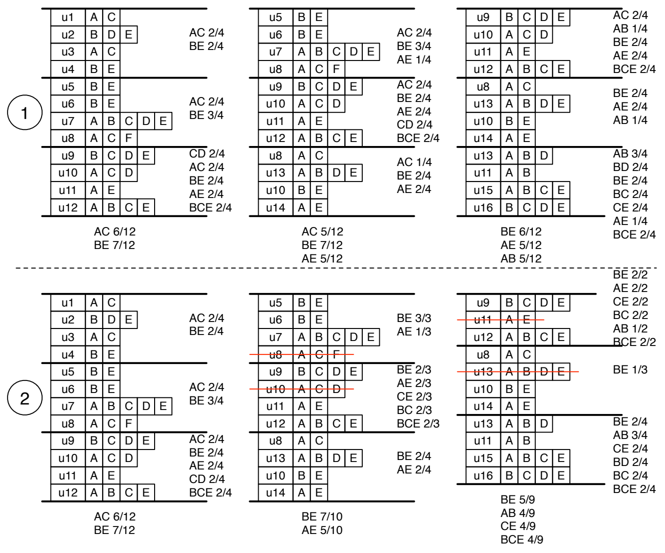

Figure 1: Motivating Example

no longer represents the current watch list of u_i, we delete it and decrease the counts of the itemsets that are related to it. The first sliding window in the figure contains all distinct users. We can extract AC and BE, which are frequent in this window. The first window then moves to the second window with the arrival of a new slide, which contains two tuples whose users u_8 and u_{10} already exist. Thus, we delete the two old tuples, u_8 and u_{10}, then re-compute the frequencies of itemsets in the new window. We find that itemset AC is no longer frequent, while AE has become frequent. Similarly, when a new slide containing existing users u_{11} and u_{13} arrives in the third window, we delete their old tuples. We discover that BE remains frequent, while AB and CE become frequent.

Now let us consider the upper row in Figure 1, where a traditional sliding window approach is used that considers every tuple as a new one (tuple revisions are ignored). In the second window, where u_8 and u_{10} have two tuples each, one is obsolete and the other is current. Itemset AC is incorrectly reported as frequent because all tuples were inappropriately considered as being independent. Similarly, in the third window, CE is not reported as frequent because the old tuples of u_{11} and u_{13} were considered when computing the counts of the itemsets.

As shown in Example 1, the method in the lower row of Figure 1 always allows correct discovery of frequent itemsets while the method in the upper row does not. However, the method in the lower row requires re-computation of the counts of itemsets over the whole window. This is very costly hence an efficient method is needed.

There are two models for dealing with revised tuples when mining frequent itemsets from sliding windows over tuple-evolving streams. One model is to update the content of the tuple inside the same slide that previously contained the tuple. The other model is to first remove the old tuple from the slide where it was previously, then add the revised tuple to the new slide of the streaming data. Both models are reasonable depending on the needs of the applications. In this paper, we adopt the latter model, hereafter referred to as *revision model*, simply because we consider the revised tuple as a new tuple in the stream.

It is worth noting that a lot of the data (e.g. Web usage data) for mining and management are similar to tuple-evolving data streams. But existing methods often assume that the raw data has been preprocessed before pattern discovery. For instance, in Web usage mining, identifying users and user sessions, and defining transactions are both needed before association rule mining [6]. However, these first data preparation then pattern discovery methods are offline approaches thus can not be utilized to online extract frequent itemsets from tuple-evolving data streams.

1.1 Contributions

We discuss the idea of mining frequent itemsets from tuple-evolving data streams. The novelty lies with the "evolution" of tuples that may get amended over time. We prove that in streams without evolving tuples slide-infrequent itemsets cannot be window-frequent. Then we show through examples that in the more general case of itemset-evolving streams, slide-infrequent itemsets actually can be window-frequent, even if we consider frequentness separately among the new and updated partitions of an updated slide. We next define conditions in which it can be guaranteed that an itemset is window-infrequent, based on previous slide-infrequentness and low enough support in the new slide.

We suggest two tree structures to maintain the evolving tuples and candidate frequent itemsets, respectively, of the sliding window, and describe the update procedures during data evolution. We develop a strategy to help prune the unpromising itemsets, we also present a theorem to prove the correctness of this strategy. We next design an efficient verifier which scans the two tree structures and produces the frequent itemsets. We prove that frequent itemsets found by this verifier are complete and exact.

Our experiments on two real datasets, one with tuple revisions and one without tuple revisions, show that our method outperforms a well-known algorithm *DTV-DFV* [14] and a naive method without using more memory.

The rest of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we give the theoretical evidence that supports our chosen model for dealing with the revised tuples. We introduce our data structures in Section 4. In Section 5, we present our approach to mining frequent itemsets in tuple-streaming streams. We evaluate the proposed method in Section 6 and draw the conclusions in Section 7.

2. RELATED WORK

Streaming Frequent Itemset Mining. Landmark window model and sliding window model are two basic models usually adopted in the literature to manage streaming data. Here we focus on related works using sliding window models.

estWin is an approximate method that maintains significant (candidate) itemsets in a common prefix lattice structure, where each node represents an itemset while an edge between nodes denotes a subset relationship [4]. Frequent itemsets are selected from the lattice only when mining results are requested.

Moment [5] is an algorithm for mining closed frequent itemsets. It stores all itemsets (frequent or not) in a FP-tree-like structure (*CET*) and updates them when a new transaction arrives or an old one expires. The method suffers when the number of transactions in the slide is large.

Mozafari *et. al.* [14] proposed a method for exact frequent itemset mining over large sliding windows, where each window is divided into similar-sized slides. In their approach, however, the reporting of frequent itemsets may be delayed by up to one window.

Tuple updates has been discussed extensively in the context of streams. The main problem is how to keep the patterns or query answers up-to-date when new tuples come and/or old tuples expire [8].

Tuple revisions are not the same as updates, tuple revisions are corrections as they invalidate previously processed inputs and all query results that were produced from them, while tuple updates do not invalidate any previously output query results [15]. To minimize the staleness of query results over streams with revision tuples, Alexandru Moga et al. proposed an efficient storage-centric framework for load management over the streams [13].

Parisa Haghani et al. studied the problem of continuous top-k query processing over multiple non-synchronized streams, where the attributes of an object arrive separately in different streams [9].

In [17], we present *ABS*-a user-centric data stream model. This model is suited to itemset-evolving data streams since it handles tuple revisions effectively. Moreover, it is good at preserving the long usage patterns and measures the bounce rate of a usage stream more authentically. Based on *ABS*, in [18] we investigated how to cluster itemset-evolving usage streams.

In summary, the related issue about mining frequent itemsets from itemset-evolving data streams, to the best of our knowledge, has not been investigated in the literature.

3. PROBLEM ANALYSIS

Using the *revision model*, we now investigate the problem of mining frequent itemsets from sliding windows over itemset-evolving streaming data, where the windows are cut into smaller slides (panes). However, when tuples in the past slides get revised, we should make sure that the frequent itemsets derived are correct and complete. But this may require re-scanning of the past slides to recompute the counts of all the itemsets. To solve this problem, we present here the theoretical support that simultaneously eliminates the need for re-scanning the slides having updated tuples and guarantees that we can accurately identify all the frequent itemsets. The notations used in the paper are shown in Table 1.

Table 1: Notation

Symbol	Meaning
W, W_o, W_c	window, old window, current window
$\|W\|$	number of transactions in W
$S_1, \ldots, S_n$	slides
$S'_1, \ldots, S'_n$	updated slides
S_n	set of new transactions in S_{n+1}
S_u	set of updated transactions in S_{n+1}
$l_1, \ldots, l_n$	number of transactions in $S_1, \ldots, S_n$
$l'_1, \ldots, l'_n$	number of transactions in $S'_1, \ldots, S'_n$
I	itemset
$Count(I\|W)$	count of I in W
$\sigma, 0 < \sigma \leq 1$	support ratio threshold

Given a sliding window W, an itemset I has to be frequent in at least one slide in order to be frequent in the whole window, regardless of the size of the slides. That is:

THEOREM 1. *Given a sliding window $W = \{S_1, S_2, ..., S_n\}$, an itemset I and a support threshold σ. I is infrequent in W, if I is infrequent in every slide.*

PROOF. Let l_i be the number of transactions in the slide S_i (see Table 1). If I is infrequent in every slide then: $Count(I|S_i) < \sigma \times l_i, \forall S_i, 1 \leq i \leq n$. Hence: $Count(I|W) = \sum_1^n Count(I|S_i) < \sum_1^n \sigma \times l_i = \sigma \sum_1^n l_i$. Therefore, I is infrequent in W. $\square$

From Theorem 1, we can see that an itemset I should be frequent in at least one slide before it becomes frequent in the whole window, irrespective of the size of the slides. However, in the case of itemset-evolving streams, when the window moves from W_o to W_c, we have a different result. Recall that in itemset-evolving streams, tuples in past slides can be revised, that is, some tuples may be removed from the existing slides. Indeed, if I is infrequent in every slide of $W_o = \{S_1, S_2, ..., S_n\}$ and remains infrequent in the new slide S_{n+1}, I may be frequent in W_c. We give two counter examples in the following.

Let $W_o = \{S_1, S_2\}$, $W_c = \{S_2, S_3\}$, $\sigma = 0.4$. In W_o, $l_1 = 55$, $l_2 = 45$; for itemset I, $Count(I|S_1) = 21$, $Count(I|S_2) = 17$. Then:

(i) If the total number of tuples in the sliding window remains the same, then $|W_o| = |W_c| = 100$. After some tuples of S_2 were amended in S_3, in W_c: $l_2 = 40$, $l_3 = 60$. Suppose that in W_c, $Count(I|S_2) = 17$, $Count(I|S_3) = 23$. We can see that in W_o, I is frequent in neither slide; in the new slide S_3, it is still infrequent. However, $Count(I|W) = 40$ thus I becomes frequent in W_c.

(ii) Else, in W_c, let $l_2 = 40$, $l_3 = 55$, $Count(I|S_2) = 17$ and $Count(I|S_3) = 21$. Then $|W_c| = 95$, and $Count(I|W) = 38$, although I remains infrequent in S_3, it becomes frequent in W_c.

When some slides have been updated, we need to guarantee the completeness of the frequent itemsets. Therefore, we may need to check the frequency of all the itemsets in all the previous slides as well as in the new one. To avoid such disadvantages, we need to correctly figure out which itemsets have become frequent and which have not. To this end. we first divide the new slide S_{n+1} into two parts, with one block S_n containing only the new transactions and the other S_u consisting of revised transactions, then propose the following theorem.

THEOREM 2. *Let p_i, be the count of transactions in slide S_i in W_o that have been updated in the new slide S_{n+1} of W_c, where $W_o = \{S_1, S_2, ..., S_n\}$, $W_c = \{S_2, ..., S_n, S_{n+1}\}$.*

For all I, if it satisfies,

(1) not exists S_i in W_o, $1 \leq i \leq n$, such that I is frequent in S_i.

(2) $Count(I|S_{n+1}) < \sigma(l_{n+1} - \sum_2^n p_i)$.

Then I is infrequent in W_c.

PROOF. For all I in W_o, because of Theorem 1 and condition (1) we have:

$$\sum_2^n Count(I|S_i) < \sigma \sum_2^n l_i, 2 \leq i \leq n$$

Tuples in slides S_i $(1 \leq i \leq n)$ in W_o could be updated in slide S_{n+1} in W_c. Thus, the new count of I in S'_i $(2 \leq i \leq n)$ in W_c must not be greater than the old count in W_o. That is:

$$\sum_{2}^{n} Count(I|S'_i) \leq \sum_{2}^{n} Count(I|S_i), 2 \leq i \leq n$$

Moreover, for W_c, given condition (2), we can estimate $Count(I|W_c)$ which is the count of I in W_c:

$$Count(I|W_c) = \sum_{2}^{n} Count(I|S'_i) + Count(I|S_{n+1}) =$$

$$\leq \sum_{2}^{n} Count(I|S_i) + Count(I|S_{n+1})$$

$$< \sigma \sum_{2}^{n} l_i + Count(I|S_{n+1})$$

$$< \sigma \sum_{2}^{n} l_i + \sigma(l_{n+1} - \sum_{2}^{n} p_i)$$

$$= \sigma\{l_{n+1} + \sum_{2}^{n}(l_i - p_i)\} = \sigma\{l_{n+1} + \sum_{2}^{n} l'_i\} = \sigma|W_c|$$

Thus $Count(I|W_c) < \sigma|W_c|$ and according to Theorem 1, I must be infrequent in W_c. □

Therefore, we can rely on Theorem 2 to guarantee the completeness of frequent itemsets discovered from the sliding window and in the meantime avoid scanning all past slides to rediscover frequent itemsets. Based on Theorem 2, we introduce Theorem 3 which allows us to easily separate the possible frequent itemsets from the unpromising ones.

THEOREM 3. *Let S_{n+1} be divided into S_n and S_u and l_n and l_u their respective sizes. We introduce a new parameter σ',*

$$\sigma' = \sigma \times l\_n/(l\_n + l\_u) \qquad (1)$$

For all I, if I satisfies,
(1) not exists S_i in W_o, $1 \leq i \leq n$, such that I is frequent in S_i.
(2) its support ratio is less than σ' in the new slide S_{n+1}. Then, I is infrequent in W_c.

PROOF. The number of transactions in slide S_{n+1} is l_{n+1}, thus $l_{n+1} = l\_n + l\_u$. According to condition (2) and equation 1 we have:
$Count(I|S_{n+1}) < \sigma' l_{n+1} = \sigma l\_n = \sigma(l_{n+1} - \sum_{2}^{n} p_i)$
Therefore, the conditions of this theorem are exactly the same as those in Theorem 3. Thus this theorem holds. □

As a consequence of Theorem 3, we do not have to scan old slides to make sure that the frequent itemsets are completely discovered; instead, all we have to consider are those itemsets whose support ratios in the new slide are no less than σ'.
It should be noted that when we calculate the revised tuples in Theorem 3, we do not include the ones from the expired slides. In other words, tuples updated from the expired slides are considered to be new tuples in the new slide.

4. DATA STRUCTURES

We propose two novel tree-based data structures that allow us to efficiently extract frequent patterns from itemset-evolving streams. *swTree* stores all tuples in the sliding window, while *cfTree* keeps candidate-frequent itemsets.

4.1 swTree: Sliding Window Tree

swTree always holds all transactions in the current sliding window. Items in tuples are ordered by canonical order. When a tuple is inserted or amended, we keep a reference pointer to the according node in *swTree*.

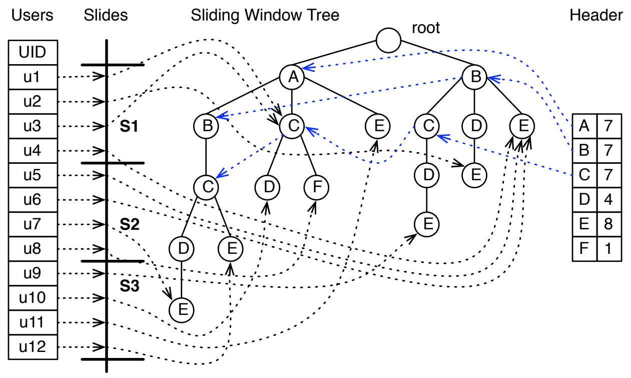

Figure 2: *swTree* structure

Figure 2 shows the *swTree* structure for window 1 in Figure 1. We can see that our structure contains four parts: Users, Slides, *swTree* and a Header table. Each user is uniquely identified and points to the node in *swTree*, which is the last item of his/her tuple after it has been inserted in the structure. For example, user u1 with itemset AC points to the according node C that follows node A (A/C) in *swTree*. Keeping such pointers allows us to update the tuples efficiently. The slides structure is the same as in a classic sliding window. For *swTree*, we keep a header table that is the same as that in the *FP-tree* [10]. This header table stores all distinct items with their frequencies and a list of pointers to the nodes in *swTree* having the same item.

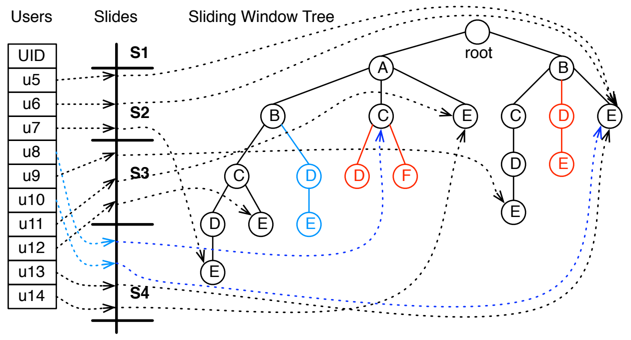

Figure 3: Updated *swTree* structure

Consider again the lower row in Figure 1. After slide S1 expires, window 1 moves to window 2. Figure 3 shows the new *swTree* representing all the tuples in window 2 (for the sake of clarity, the header table has been omitted from the diagram). To update *swTree* from window 1 to window 2, we take the following steps:

- for expired slide(s), we simply remove all the pointers related to that slide and decrease the counts of all ancestor nodes. Therefore, all pointers related to s1 are deleted.
- for amended tuples, we decrease the counts of old ancestor nodes and update the related pointers. For example, when u8 is revised, we delete the old pointer from S2, which is A/C/F, and add a new pointer in S4 to the last node of the new tuple, which is A/C.
- for the new slide that arrives, we insert the new tuple to the *swTree*, then draw pointers from the users to the corresponding tuples in *swTree*.

When updating tuples in the classic sliding window, old tuples first have to be retrieved in the window, then new tuples can be added. Using *swTree* structure, however, tuple revisions and slide expirations are quickly handled.

4.2 cfTree: Tree Keeping Candidate Frequent Itemsets

cfTree keeps only candidate-frequent itemsets and itemsets discovered from the new slide are added to it. Meanwhile, we update the frequencies of the related itemsets in *cfTree*.

The structure of *cfTree* is similar to FP-tree [10], with the only difference being that we associate a slide id to each node in *cfTree*. The slide id registers the id of the most recent slide where the itemset was found to be frequent. So, when an existing itemset is found frequent in the current slide, the slide id of the corresponding node is updated to this current slide id. The reason that we add such a slide id is that *cfTree* can grow very large. Therefore, we need a strategy to help prune the unpromising itemsets. We develop such a strategy according to the following theorem.

THEOREM 4. *Let* csn *be the current slide id and* n *the number of slides in W. For an itemset I, let* nd *be the according node in* cfTree *and* nd.sn *the id of the most recent slide where I was found frequent. We have*
$\forall$ itemset I, if csn - nd.sn≥n*, then I must be infrequent in W thus can be pruned from* cfTree*.*

PROOF. Let S_1 denotes the most recent slide where I was found frequent. $csn - nd.sn \geq n$, thus I must not be frequent in any of the m followed slides S_j, $2 \leq i \leq m+1$, $m \geq n$. Given the number of slides in a sliding window is n, according to Theorem 1, I must be infrequent in the current sliding window. Therefore, I can be safely pruned from *cfTree*. □

According to Theorem 4, we can make a breadth-first traversal over *cfTree* and prune all nodes that satisfy its condition.

But does *cfTree* contain all itemsets that are actually frequent in the current sliding window? The following theorem answers to this question.

THEOREM 5. *$\forall I \notin$ in* cfTree *pruned using Theorem 4, I must be infrequent within the current sliding window.*

PROOF. $\forall I \notin$ *cfTree*, if I is frequent in the current sliding window then, according to Theorem 1, I should have been frequent in at least one of the n recent slides in the window. Then I should have been added to *cfTree* and not pruned from it as it must not satisfy the condition of Theorem 4. So I must exist in *cfTree*, which is contradictory to the conditions that $I \notin$ *cfTree*. Thus, the assumption does not hold. Therefore, $\forall I \notin$ in cfTree, I must be infrequent in the current sliding window. □

5. MVERIFIER: VERIFICATION METHOD

We introduce a novel verifier with the goal of getting the exact counts for all itemsets of *cfTree* from *swTree*. That is, $\forall itemset\ I \in cfTree$, what is the support of I in *swTree*? Our general idea is one to one sub-tree mapping and verification. We adopt a top down mapping, starting from the roots of both trees, traversing their descendants and verifying the itemsets successively. Most importantly, this mapping verifier method can avoid the excessive building of conditional trees used by a conventional *FP-growth* algorithm [10].

Algorithm 1: The Mapping Verifier

```
Input: $cfTree,\ swTree$
Output: $the\ count\ for\ each\ itemset\ of\ cfTree\ in\ swTree$
1 begin
2   foreach $cfchild \in root\ of\ cfTree$ do
3     $supp_{cfchild} = 0$;
4     $swchildren \leftarrow$
      $access\ header\ of\ swTree\ and\ find\ each\ swchild$
      $such\ that\ cfchild = swchild$;
5     foreach $swchild \in swchildren$ do
6       $supp_{cfchild} += supp_{swchild}$;
7     if $supp_{cfchild} \geq supp\_threshold$ then
8       $mverify(cfchild,\ swchildren)$;
9 end
```

Algorithm 1 explains how the approach works. The inputs are $cfTree$ and $swTree$. For each child of $cfTree$, we access the header of *swTree* looking for subtrees that match such child (line 4). After updating the counts (line 6), if its support is above the support threshold, we will recursively verify its descendants (line 8).

The calculation of the exact counts of itemsets is given in the `mverify` function, shown in Algorithm 2. For each node of the *cfTree*, we look for the corresponding children in *swTree* (line 4). After updating the count of the considered node, if its support is above the support threshold, we will recursively verify the its descendants (line 8).

THEOREM 6. *The frequent itemsets found by the mapping verifier are complete and exact.*

PROOF. $\forall$ itemset $I \in cfTree$ that begins with an item s and ends with t, $s \prec t$, we should guarantee that all paths in *swTree* containing I should be found, and each path should be calculated one and only one time for I.

(1)The completeness of transactions/paths containing I. If I is a true frequent itemset within the sliding window *swTree*, then it must have been frequent in one or more past slides of the window. I must have been registered in *cfTree* as we keep all the frequent itemsets in each slide. Since both

Algorithm 2: mverify

```
    Input: $cfroot,\ swroots$
    Output: $the\ count\ for\ each\ itemset\ of\ cfroot\ in\ swroot$
1 begin
2     foreach $cfchild \in children\ of\ cfroot$ do
3         $supp_{cfchild} = 0$;
4         $swchildren \leftarrow find\ each\ swchild \in swroots,\ such\ that\ cfchild = swchild$;
5         foreach $swchild \in swchildren$ do
6             $supp_{cfchild} += supp_{swchild}$;
7         if $supp_{cfchild} \geq supp\_threshold$ then
8             $mverify(cfchild,\ swchildren)$;
9 end
```

cfTree and *swTree* are ordered in the same way, the mapping verifier is able to map I from *cfTree* to all related transactions/paths in *swTree* that contain I. Therefore, when the verifier traverses all the sub-trees, it is able to find all distinct transactions containing I. Hence the verifier can find in *swTree* all the expected transactions containing I.

It should be noted that the mapping of I is stopped at an item immediately whenever the support of this item of I in *swTree* is below the minimum support, or when the verifier could not find such an item following the path of I in *swTree*. Will we miss the true frequent itemsets that are composed of the same prefix except for the stopping item, and items of I that are after the stopping item? If these itemsets are candidate patterns in *cfTree*, they must follow paths that are different from I. Since *cfTree* keeps all possible frequent patterns, each itemset should have already been registered in another path of *cfTree*. Therefore, we will also verify these itemsets and not miss the true frequent itemsets when applying this stop condition.

(2)The exactness of the counts for I. Given items in I and *swTree* are ordered identically, the verifier matches items in I sequentially, from s until t, each time searching for the same item in *swTree* for paths that have satisfied previous matchings of items. Therefore, the count for I must be calculated exactly. □

Discussion. *FP-growth* [10] and *SWIM* [14] are well-known algorithms for mining frequent itemsets, but they suffer from the exhaustive conditionalization and sub-trees merging for every itemset in the *FP-tree*. *SWIM* differs from *FP-growth* mainly in that it refers to the candidate frequent itemset tree when conditionlizing the sub-trees. Our method, however, matches *cfTree* directly to *swTree* without repeated conditionlization and sub-tree mergence. Since all we have to verify are the itemsets in *cfTree*, our verifier goes top-down in *swTree* referring to *cfTree*, and verifies the sub-trees based on the results from their ancestors. As such, our method greatly reduces the verification cost.

6. EXPERIMENTS

We conducted experiments on a real-world dataset: the mobile browsing data of smart phone subscribers on the portal of a large telecom. The telecom mobile browsing data is 21.8 GB, and there are more than 8000 Web pages divided into 23 different categories (sports, weather, technology, etc.). To test the efficiency of our counting algorithm, we also use a public dataset *kosarak* that has no tuple revisions. For simplicity, the naive method, which mines frequent itemsets from scratch whenever the window shifts, is called *naiveMethod*. Our framework is named *Fideo* (**F**requent **i**temset mining from **d**ata streams with **evo**lving tuples), while *Fideo.D* represents our method combined with the delayed verification strategy used in [14].

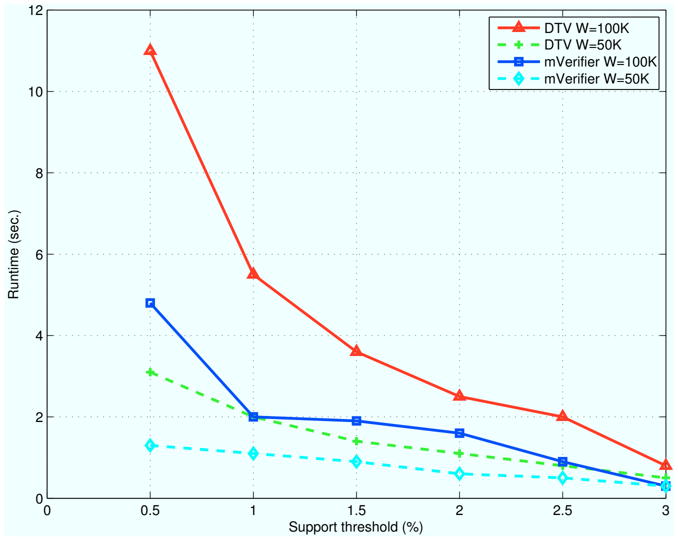

Figure 4: Scalability on Kosarak

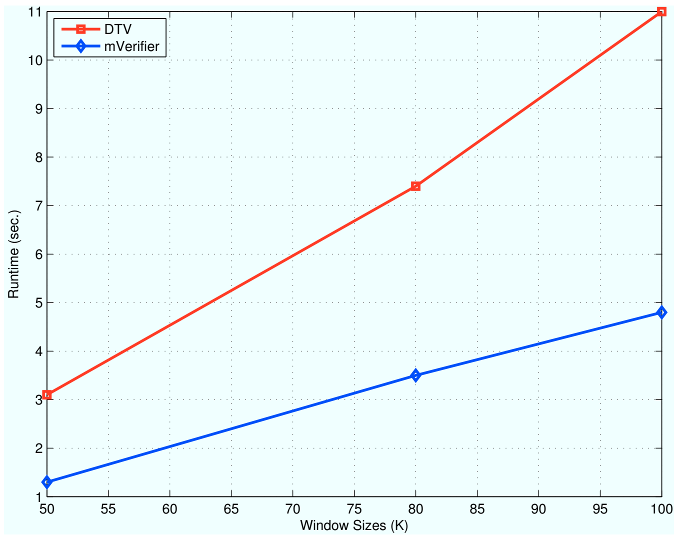

Figure 5: Scalability with Window Sizes

6.1 Efficiency of Counting Algorithm

First, we compare the performance of *mVerifier* and *DTV-DFV* [14] which has shown to be an order of magnitude faster than *Apriori* [1] and *FP-growth* [10]. *DTV-DFV*, however, needs to recursively build conditional trees to get the counts for the candidate itemsets, while *mVerifier* eliminates building such conditional trees, it counts the supports

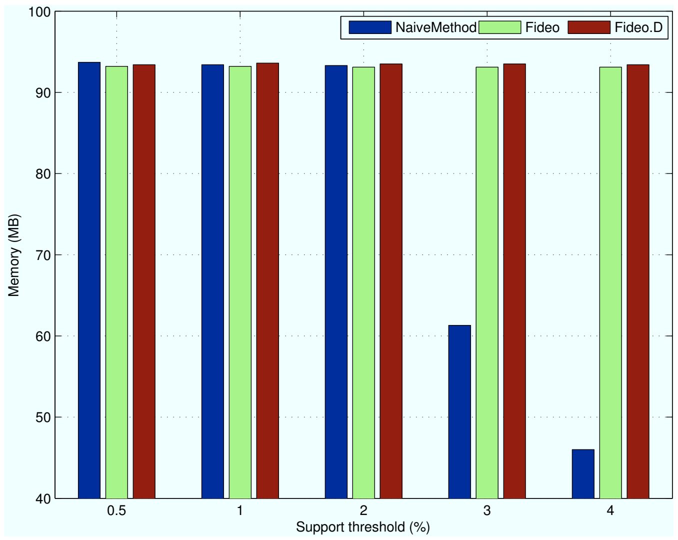

Figure 6: Memory Consumption

of the candidate itemsets top-down and stops the verification of a subtree immediately once the count of its prefix itemset is verified below the minimum support. During the comparisons, we input the same window of data and candidate frequent itemset tree to both methods, and ask them to return the frequent itemsets in the window and their respective supports. The dataset used is *kosarak* which is a public real dataset[1].

We test the efficiency of *mVerifier* with different support thresholds from 0.5% to 3%. As shown in Figure 4, *mVerifier* is indeed faster than *DTV-DFV* verifier; twice as fast as *DTV-DFV* when the support threshold is low, i.e. when there are many qualifying patterns, for the pattern matching with a lot of candidate itemsets by recursively building the conditional trees becomes very expensive.

We also evaluate the scalability with the window sizes. Here we use *kosarak* dataset and the support threshold is fixed to 0.5%. We test three window sizes: 50K, 80K and 100K. Depicted in Figure 5, both *mVerifier* and *DTV-DFV* show linear scalability with the window sizes. However, *mVerifier* is much more scalable than *DTV-DFV*. Overall, as the window size grows up or the minimum support threshold reduces, the difference between the two counting methods becomes larger and larger.

During the experiments with varying window sizes, we also notice that the maximum memory occupied by these two counting algorithms is almost the same, although *mVerifier* takes slightly less memory. This observation illustrates that *mVerifier* runs much faster but consumes no more memory than *DTV-DFV*.

We conduct experiments to verify the memory requirements for naive method, *Fideo*, and *Fideo.D* on the same dataset with varying window and slide sizes, by monitoring the maximum memory occupied by these methods.

The memory cost for *Fideo* consists of three parts: (i) *swtree* which compacts all the tuples of a sliding window in a tree, (ii) *cftree* which keeps the candidate frequent itemsets in a tree, and (iii) memory overhead when verifying the candidate itemsets. *Fideo.D*, in addition to the above three parts, needs additional memory for the auxiliary arrays attached to the nodes of *cftree*. The memory consumption of *swTree* and *cfTree* should be the same for *Fideo* and *Fideo.D*. *Fideo.D*, however, needs less memory during verification of candidates compared to *Fideo*, for it only verifies *cfTree* in the expiring slides and revision tuple trees. While for *Fideo*, it also needs to verify new candidates in *swTree* to get their counts. The naive method still needs *swTree* to hold the transactions in the sliding window (i.e. (i)), and also memory space to discover frequent itemsets from the whole window (i.e. (iii)). But when the window is large, (iii) for the naive method should be larger than that of *Fideo* (and *Fideo.D*), since it needs to build lots of conditional trees recursively. We would like to note that (iii) memory cost is much larger than (i) and (ii).

Figure 6 shows the results of the experiments in terms of physical memory requirements with a variation of support thresholds, with the window size set to 40,000 and slide size being 2,000. Our first observation from Figure 6 is that when the support threshold is low, the memory requirements for these three methods are very close. As discussed in the above passage, this is due to the (iii) memory overhead for the *naiveMethod* when the window is large. But *naiveMethod* does require less memory when the support threshold is large (e.g. 4%). Our second observation is that the memory consumptions for *Fideo* and *Fideo.D* are stable, irrespective of the support thresholds. We also vary the slide and window sizes, but the maximum memory usages for these methods are similar to those in Figure 6. This is because the memory cost of all the methods are more influenced by the (iii) cost, while (iii) cost is influenced by both the window size and the number of distinct items [11, 3].

6.2 Runtime Efficiency

Finally, we investigate the runtime efficiency of all the methods. Before presenting the results, we would like to note the total time elapsed in mining each pane/slide in the window is larger than directly discovering frequent patterns from the sliding window. Thus although pane-based sliding windows enables the incremental mining and maintenance of frequent itemsets, there is indeed no free lunch for *Fideo* and *Fideo.D*, since they need to spend more time extracting frequent itemsets in each pane, than the *naiveMethod* that only extracts frequent patterns from the window. In fact, this is also the reason that the speedup over the *naiveMethod* is not very huge: *Fideo* and *Fideo.D* save up to 50% of the time needed by the *naiveMethod*.

In Figure 7, we fix the window size as 50K and the slide size as 2k. We observe that when the support threshold is large (4%), the *naiveMethod* uses even less time than the improved methods, for the number of frequent itemset candidates are not large. However, the lower the support threshold, i.e. there are lots of candidate itemsets, the faster *Fideo* and *Fideo.D*. When support threshold is 0.5%, *Fideo* and *Fideo.D* are more than 50% faster than the *naiveMethod*.

In the next series of experiments, we set the support threshold to 1% and the number of slides to 20, then vary the sizes of the slides. Shown in Figure 8, when we increase the slide sizes from 2000 to 3500, we see that the *naiveMethod* is more influenced by the slide sizes (window sizes), while *Fideo* and *Fideo.D* scale better with respect to the slide sizes.

[1]http://fimi.ua.ac.be/data/

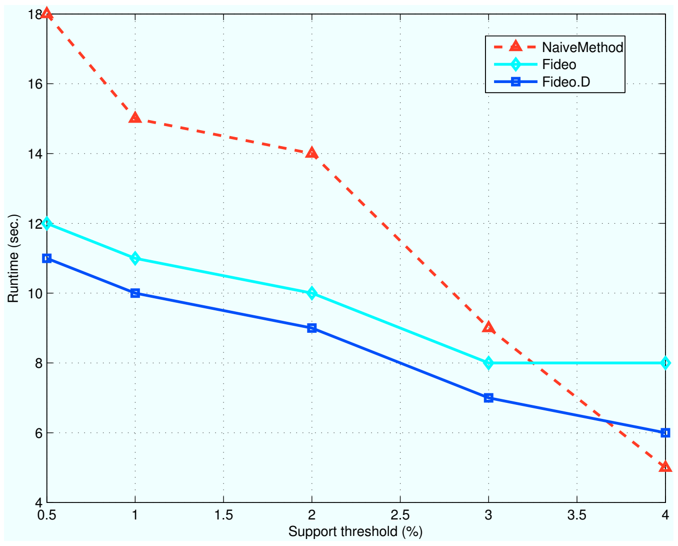

Figure 7: Runtime with Varying Thresholds

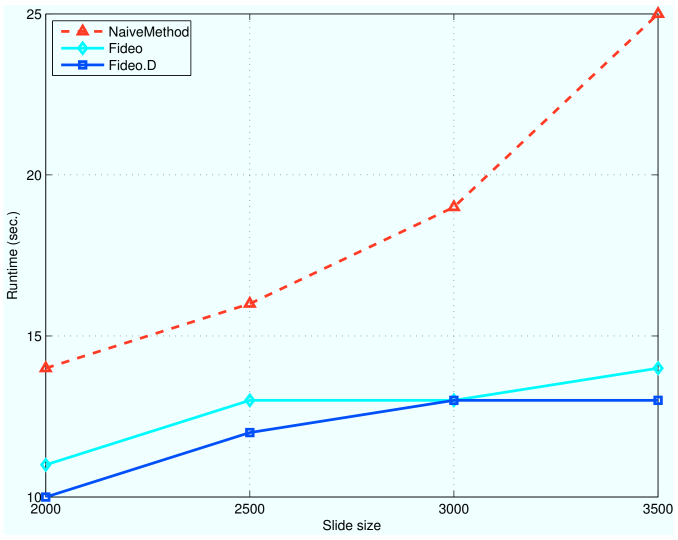

Figure 8: Runtime with Varying Slide Sizes

7. CONCLUSION

In this paper we investigate how to extract frequent itemsets from tuple-evolving streams. We define conditions to determine whether a slide-infrequent itemset can be window-infrequent. We design data structures that manage the evolving tuples effectively. We design methods that adapt to tuple-evolving streams and discover frequent itemsets efficiently. Experiments have demonstrated the efficiency and effectiveness of our proposal. In the future work we will study how to do pattern matching over tuple-evolving streams.

8. ACKNOWLEDGEMENTS

This work, conducted at UCLA under Prof. Zaniolo, was funded by INRIA and its *Explorer* program.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[3] A. Ceglar and J. F. Roddick. Association mining. *ACM Comput. Surv.*, 38, July 2006.

[4] J. H. Chang and W. S. Lee. *stWin*: adaptively monitoring the recent change of frequent itemsets over online data streams. In *ACM CIKM*, pages 536–539, 2003.

[5] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *IEEE ICDM*, pages 59–66, 2004.

[6] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowl. Inf. Syst.*, 1:5–32, 1999.

[7] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[8] L. Golab and M. T. Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD Conference*, pages 658–669, 2005.

[9] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.

[11] W. Kosters, W. Pijls, and V. Popova. Complexity analysis of depth first and fp-growth implementations of apriori. In *Machine Learning and Data Mining in Pattern Recognition*, volume 2734 of *Lecture Notes in Computer Science*, pages 77–119. Springer, 2003.

[12] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34:39–44, March 2005.

[13] A. Moga, I. Botan, and N. Tatbul. Upstream: storage-centric load management for streaming applications with update semantics. *VLDB Journal*, 20(6):867–892, 2011.

[14] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, pages 179–188, 2008.

[15] E. Ryvkina, A. Maskey, M. Cherniack, and S. B. Zdonik. Revision processing in a stream processing engine: A high-level design. In *ICDE'06*, pages 141–144, 2006.

[16] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo. Smm: A data stream management system for knowledge discovery. In *ICDE*, pages 757–768, 2011.

[17] C. Zhang, F. Masseglia, and Y. Lechevallier. Abs: The anti bouncing model for usage data streams. In *IEEE ICDM*, pages 1169–1174, 2010.

[18] C. Zhang, F. Masseglia, and X. Zhang. Modeling and clustering users with evolving profiles in usage streams. In *TIME*, pages 133–140, 2012.