An Approach for Mining Frequent Item sets from Tuple-evolving Data Streams

Bhargavi Peddireddy, Ch. Anuradha, P.S.R. Chandra Murthy

Abstract: Today, data streaming applications consider every incoming transaction as a new tuple.Most of the applications allows the tuples revision inside the streams over the time. This kind of revision in data streaming application gives new and hidden knowledge, also brings new challenges to the tasks. One of the issue is, frequent itemsets may become to infrequent and viceversa . To address this issue,We design efficient data structures to maintain stream data, information and candidate information of evolving tuples. We propose an algorithm that combines effective data structures that derives frequent itemsets over the tuple evolving data streams.

Keywords :data streams, SlideTree, HashTable, tuple-evolving data streams.

I. INTRODUCTION

Due to the usage of technology and data, processing data streams has been used by many applications greatly in the past few years. In this environment, transactions are continuously added to the data streams and these transactions are to be processed fast to give answer to the user query [3]. For the faster processing data streams, several models have been proposed in the related work. Some of them are with the consideration that is unbounded sequence of streaming and other are popular sliding windows [14]. In data streams, sliding window property has been playing an important role and proved to be efficient and complete for frequent pattern discovery[4] and query processing[12].

Frequent pattern discovery has been one of the important and prominent research topic in data mining, because of various applications such are Market Basket analysis, gene analysis, and fraud detection. The integration of FIM and data streams has been attracted [4, 9, 15, 19].

However, it has not been investigated extracting frequent patterns from the data streams where tuples are allowed to get revised, named as tuple-evolving data streams [7].

A. Motivation example

Consider auction site as an example, where customers can watch or bid the list of itemsets that they are interested from the list of items in the auction site and also user can update their bid list at any time. In addition to that, the auction site allows many bids across the various items, also allows to resubmit bid and items expire automatically from the bid list over the time.

Revised Manuscript Received on May 06, 2019

Bhargavi Peddireddy, Department of Computer Science and Engineering, ANUCET, Acharya Nagarjuna University, Guntur, A.P, India.

Ch. Anuradha, Assistant Professor, Department of Computer Science and Engineering, V.R. Siddhartha Engineering College, Vijayawada, A.P, India.

P.S.R.Chandra Murthy, Department of Computer Science and Engineering, ANUCET, Acharya Nagarjuna University, Guntur, A.P, India

Table 1 shows the users and their interested items as bid in which users = {UT_1, UT_2, … ,UT_M}, items {*a, b, c, d, e and f*} are the item names that associated with each user transaction. Figure contains 12 transactions hence the window size |M| is 12. The goal is to find the frequent itemsets with in a window and slide size is 4, and a minimum support is 40% (count 4). Since each item has its expiry time, the database is divided into slides. From the example, it can be seen that slide size is 4, and the minimum support is 40 % (means support count is 5).

UT1	a	c			
UT2	b	d	e		
UT3	a	c			
UT4	b	e			
UT5	b	e			
UT6	b	e			
UT7	a	b	c	d	e
UT8	a	c	f		
UT9	b	c	d	e	
UT10	a	c	d		
UT11	a	e			
UT12	a	b	c	e	

Table 1: Sample Auction Database

Figure 1 shows the various the assumptions and approaches that are considered for incoming transactions to derive frequent itemsets from the data streams. Basic approaches are projected in the figure 1. The very first one describes that it consider every incoming transaction as a new transaction. In the second window, itemset <ac>:5 become frequent due to the repeated transaction of UT8 which is supposed to be obsolete at one place. The reason for that is tuple evolution / update is not considered. Similarly, *ce* was not reported as frequent in 3rd slide because of the consideration of the older tuples UT8 and UT10, which causesthe support count changes from 4 to 5. From this example, it is observed that ignorance of tuple evolving can causes for such invalid knowledge.

The second one consider, if the tuple T1 arrives before T2 of the same user, then T2 is the new bid transaction, and discards T1 from the database if they are in the same slide. If they are in different slide, then we delete the transaction and decreases the support count of the itemsets that are in the deleted transaction. In the second sliding window, window moves because of the new slide 4, which contains 2 older transactions UT8 and UT10 out of four.

Thus we delete UT8 and UT10 and also update the knowledge of the window, then we recomputed the support count of the new slide. It is observed that itemsets a, c and ac become infrequent whereas ***ae*** remains frequent.

Figure 1 (c) consider if the tuple T1 arrives before T2 of the same user, then T2 is the new bid transaction, update T1 with the items of T2 (remove and insert same transaction) and discard the transaction T2 when they are in the different slides. In the first sliding window, all the recorded transactions are made by unique users. And few of them are found as frequent {*a:6, b:7,c:7,e:8,c;6,be:7*}. Second column shows that window moves from slide 2 to slide 4 with arrival of new slide 4, which contains UT8 and UT10 are already exist in slide 2 and slide 3. Thus, we update two tuples and re-compute these slides the support count of the itemsets. There was a change in frequency of itemsets in second slide, but not in frequent list. In third slide of second column, itemset d is found to be infrequent, hence there was a change in frequent list because of UT10 tuple evolving.

The possible solutions are visualized for the tuple evolving data streams bellow.

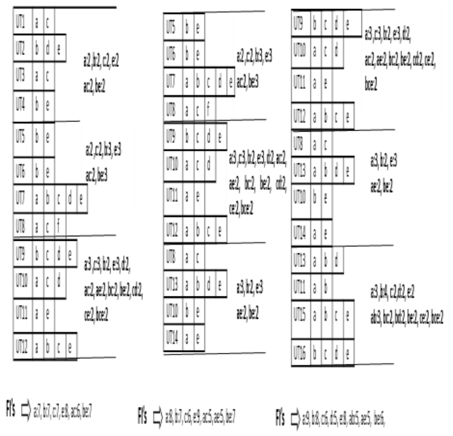

Figure 1.a: Naïve solution

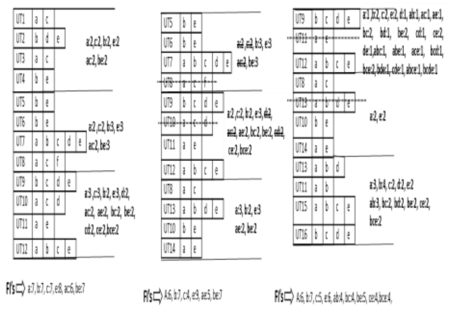

Figure 1.b: Revision Model-Remove and Update

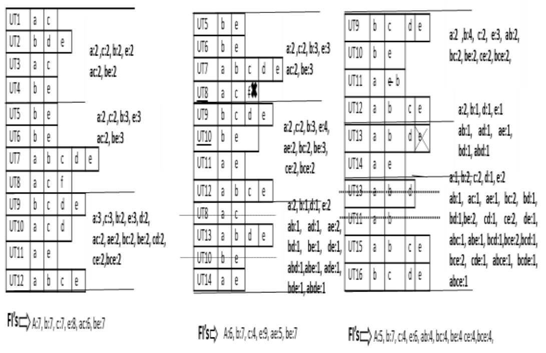

Figure 1.c: Revision Model- Update

As shown in figure 1, first method which consider every incoming transaction is a new transaction doesn't guarantee that the derived frequent itemsets are complete. There are the two models handling tuple-evolving (revised/ modified tuples), are presented in the figure. The first one removes the first occurrence transaction and decreases the count for the related revised tuples, whereas second model update the first occurrence transaction by replacing with the new transaction data and ignore the new transaction. In both cases, we recomputed the slides which got involved with the revised tuples. Hence it is guarantee that the output of these model is complete.

In this paper, we consider the first model as revision model (figure 1 b) which consider the new transaction is the revised tuple.

B. Contributions

In this section, we discuss the idea of frequent itemset mining from the data streams where tuples get evolved.In addition the idea, we also discusses the conditions that are able to identify the itemsets that are going to be frequent and infrequent when the tuples are get updated in new slide. We propose SlideTree, FIHashTable and use Hash Table data structures to maintain all the transaction information of each slide including evolving tuples, and frequent information of each itemset, and for handling candidate frequent itemsets of the new slide. In addition to that we propose verify and Frequent Itemsets methods to update SLTree for completeness and deriving all the frequent itemsets. We propose an efficient approach Update which scans database once and scans data structures and produces the frequent itemsets that are complete.

We show the performance of the proposed method though the experiments on two real data sets, one without tuple revisions and one with tuple revisions, the proposed method in this paper outperforms Naïve method, DTV-DFV [4], and Fedeo [7].

The remaining paper is presented as follows, wefirst discuss the related work of data streams in very next section that is section 2. We present the formal notations and conditions that supports chosen model for dealing with tuple-evolving in section 3. We introduce the data structures in section4. We discuss the proposed approach for handling tuple-evaluation in section 5.

We perform the experiment evaluation in section 6. And finally conclude this paper with a conclusion in section 7.

II. RELATED WORK

FIM [1, 11, 16, 18] has been attracted in mining frequent patterns from data streams. To exhibit continuous flow of the data, several extensions have been proposed to FIM to extend it to the data streams. But all approaches had challenges.

1. The current frequent items sets may become infrequent in future
2. The current itemsets which are not interested may become to interested patterns or frequent in future.

In order to mine itemsets from data streams, new concept is introduced named as window model, whose size was fixed or variable with the transactions from n^{th} to m^{th}. It is observed from the literature that, various models have been proposed those are:

1. Land mark window model
2. Sliding window model,
3. Damped window model
4. Title Time Window Model

Land mark and sliding model are the two basic models for handling or finding the interested itemsets from the data streams.*estWin*[10] is an approximate method for addressing the old itemset domination. It uses prefix lattice structure for maintaining candidate itemsets, where each node is an itemset and edge shows subset relationship. The advantage of this method is frequent itemsets are derived from lattice only when mining is requested.

An extension of FIM is, Moment algorithm [19] is proposed for extracting closed frequent itemsets. It maintains all the itemsets in a FP Tree kind of tree CET tree. The CET tree is to be updated for each update that is either new transaction arrival or old transaction removal. This method shows the difficulty when the slide size is large. To address this issue of large slide, Mozafari et al. [4] proposed a method, where each slide is divided into equal small slides and finds all the itemsets. Tuple updates has been main theme of this paper and discussed in the introduction. It has been initiated in [8,13], where tuple updates are considered as corrections as they invalidate the past results which are considered for the next slide. To minimize the overhead, Alexandru et al. [2] proposed storage-centric framework for managing the data.

Parisa Haghani et al. [8]investigated top-k query processing over data streams, where the streams are multiple non-synchronized and the arrival time of objects attributes are different.*ABS*-a user-centric data stream model is presented in [6] to handle tuple revisions effectively. It is good atpreserving the long usage patterns and measures the bouncerate of a usage stream more authentically. The investigation of clustering itemset-evolving usagestreams is presented in [5].

C Zhang et al. [7] proposed Fideo framework to handle itemset-evolving tuples efficiently. It maintains all the tuples information including tuple update information in *swTree* and candidate itemset information of a new slide in *cfTree*.In addition to the two data structures, MVerifier method to ensure that the output is complete. However, for each move, two tree data structures need to be updated, for finding frequent itemset too, that leads to a huge storage and more computation.

In summary, to the best of my knowledge, from the literature, the issues of mining frequent itemsets from itemset-evolving data streams has not been investigated extensively.

III. PROBLEM ANALYSIS

We now investigate the problem of mining frequent itemsets fromsliding windows of data streams using the *revision model*(figure 1), where the tuples in the past slides get updated and the windows are cutinto smaller slides. Usually it requires re-scanning of the past slides to re-compute thecounts of all the itemsets in updated transactions. To solve this problem, we presenthere the conditions that eliminatesthe need for re-scanning the slides that are having updated tuplesand guarantees for all the frequent itemsets. The notations used in this paper are shown in Table 1.

Symbol	Meaning
W, W_{old}, W_{cur}	Window, Old Window, Current Window
$\|W\|$	Length of window (total number of transactions)
$Sl_1, .., Sl_n$	Slide
$Sl'_1, \ldots, Sl'_n$	Updated Slides
Sl_n	New transactions in new slide Sl_{n+1}
Sl_u	Updated transactions in new slide Sl_{n+1}
$L_1, \ldots., L_n$	Number of transactions in ith slide
$L'_1, \ldots, L'_n$	Number of transactions in updated slides
I	Itemset
I^K	K^{th} item of an itemset I, where K from 0 to |K|
Support(I/W)	Occurrence of an itemset I in window W
	User Minimum threshold

Table 2: Terminology

A. Theorem 1 [7]:

For a given sliding window $W=\{Sl_1, Sl_2, .. , Sl_n\}$, Sl_i is a slide number that contains transactions, an itemset I and minimum threshold. If I is infrequent in every slide, then I is infrequent in W.Proof: proof for the theorem is presented in [7]. The counter example for the statement is presented below. For example consider itemset $I= \{ad\}$. An Itemset I is not frequent in any of the slide, hence it is not frequent in window W. Another example is $I=\{f\}$, I is infrequent since it is infrequent in every slide.In tuple revision model, when slide moves from W_{old} to W_{cur}, the result is different. When tuples get updated, means that some of the tuples may be deleted. Indeed, an itemset I may become frequent though it is infrequent every slide in W_{old} and new slide [7].

When some slides have been updated, we need to guarantee that all the frequent items are derived. Therefore, it may require to check the count of all the itemsets in the past slides as well as in new slide. To avoid such repetition, it is essential to identify the itemsets that are become frequent and not. We divide new slide into two parts, one part containing the new transactions and other containing the past slide transactions. We figure out itemsets that are frequent or infrequent in a new slide using theorem 2.

B. Theorem 2 [7]:

TC_i be the number of the transactions in slide *Sl_i* in *W_{old}*, that have been updated in new slide *S_{ln+1}* of *W_{cur}*, where *$W=\{Sl_1, Sl_2, .. , Sl_n\}$*, *$W_{cur}=\{ Sl_2, Sl_3, \dots , Sl_{n+1}\}$*.For all I

(1) *$SupCount(I, Sl_{n+1}) < (\delta \times (L_N/(L_N + L_U)))$* then *I* is infrequent in *Sl_{n+1}*.

(2) *$SupCount(I, Sl_{n+1}) \geq (\delta \times (L_N/(L_N + L_U)))$* then *I* is frequent in *Sl_{n+1}*.

From the theorem 2, we do not need to scan the past slides when itemsets are infrequent in new slide. If an itemset is frequent in new slide, there is a chance that it may become frequent in *W_{cur}*. But if it is not frequent in the past slides of *W_{cur}*, then such kind of itemsets are to be checked in the past slides. To avoid re-computation, we propose *SlideTree* data structure, Update and verification method to efficiently find all the itemsets whose support count is greater than minimum threshold. *SlideTree* maintains all the tuple information of sliding window. Update method remove the past slide from *SlideTree* and updates new slide information of *W_{cur}*. Verify method checks the support of itemsets which are frequent in new slide but not in the past slides.

IV. SLIDETREE: SLIDING WINDOW TREE

SlideTree maintains all the transactions of current sliding window. When a tuple is inserted, we keep a reference pointer, and remove a reference pointer when a tuple is removed. For each item in the current sliding window, reference pointer is created in the header. We maintain item and its total count in the header. Figure 2 shows the node structure of *SlideTree* that holds the information about node includes label name *info*, count at that path as *sup*, parent node address *PID*, and address of the same item*adj*.

info	*sup*	*PID*	*adj*

Figure 2 : Node structure

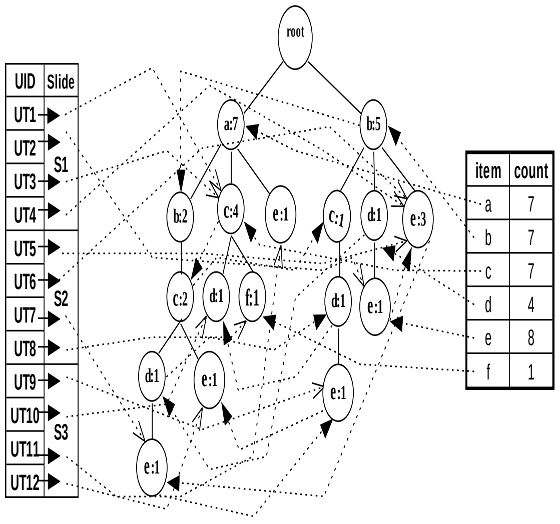

Figure3: SlideTree of W(Sl1, ..., Sl12)

Figure 3 shows the SlideTree for the window *$W(Sl_1, .. ,Sl_{12})$*. We can see that it contains three parts:User Transaction id (UTi) with slide information, SlideTree and Header. Each user transaction is uniquely identified by its UTi, each UTi associated with the slide number and points to the last items of transaction (branch) in the tree. Storing such kind of pointers helps to update the tuple easily whenever it is updated in next window. We keep a header table to maintain item label and its support count, where each item is associated with its one label in SlideTree. Each node holds the address of the same label in the tree. Keeping such pointer helps to find the support count of each itemset easily.

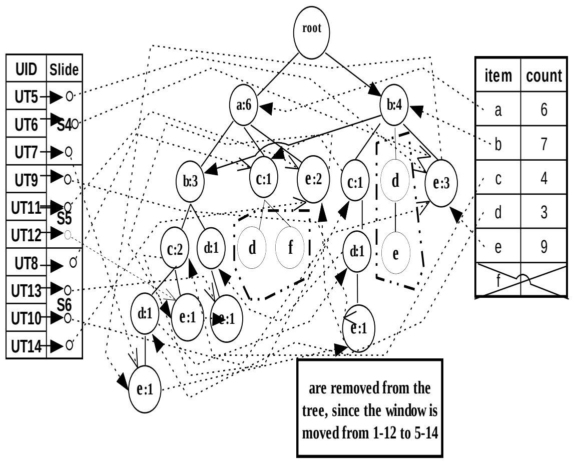

Figure 4: SlideTree of $W_{cur}(Sl_5, ..., Sl_{14})$

Figure shows the *SlideTree* structure for the example in figure 1.b. After window moves from *W_{old}* to W_{cur}, slide 1 expires. We can see that information of Sl_1 is removed from the tree that is visualized with dotted lines. For example, item f is presented in slide Sl_1, which is not there in W_{cur}. We can see that item f is removed from header table. The following steps are considered to update *SlideTree* from *W_{old}* to *W_{cur}*.

- For obsolete or expired slides, we remove the pointers of the items (nodes) from *SlideTree* and decrease the count.
- For updated tuples, we remove and decrease the support of related pointers and items count. For example, UT8 {*a,c,f*}, we remove item a,c and f and decrease its count. After removal, {*a,c*} is inserted into the tree and update the support of its pointers and count.
- When a new transaction arrives, we insert new transaction id, tuple information to the *SlideTree*, update the Header table, and update the pointers.

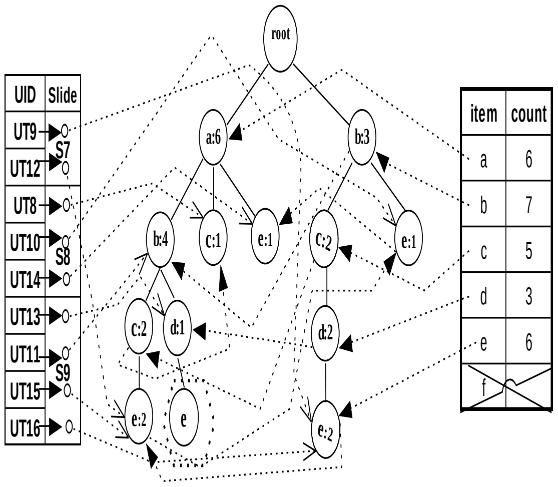

Figure 5: SlideTree of $W_{cur}(Sl_9, ..., Sl_{16})$

FIHashTable: Hash Table keeping Frequent Itemsets
FIHashTable keeps only frequent itemsets that are frequent in at least one slide. The structure of the hash table is visualized in figure 6. It contains three parts: Itemset name, support count and the list of slides, where it is found to be frequent.

Itemset Name	Support Count	The list of slides (found frequent)

Figure 6: The FIHashTable

FIHashTable maintains only the frequent information. For example, an itemset *a* is appeared in the with the support count 6 and frequent in all the slides of Window with the slide numbers from 1 to 12. This kind of structure allows to insert, retrieve and update easily in faster manner.
FIHashTable for the window 1-12 is visualized in figure 7 below.

Itemset	Support Count	The list of slides (found frequent)
a	6	1, 2, 3
b	7	1, 2, 3
c	7	3
e	8	1, 2, 3
ac	6	1,2,3
be	7	1,2,3
ae	3	3
bc	3	3
cd	3	3
ce	3	3
bce	3	3
d	4	3
ae	3	3
bc	3	3

Figure 7: FIHashTable for the window 1-12

V. UPDATE

Update (Algorithm 1) method handles the mining of frequent itemsets from the data streams when window moves from old to current with or without tuples update. Firstly, it initiates the procedure for removing the past slide information from the SlideTree and FIHashTable. For handling new slide, we inset new tuples into the tree and find local frequent items in a new slide.For handling the itemsets which are local frequent but infrequent in all other slides, we visit only the respected branches of *SlideTree*.Thus, we extracts all the frequent itemsets and said to be complete.

1. If a new candidate is frequent in local slide, and if it is not frequent in any of the past slide, then we traverse *SlideTree* to get the updated support count and keep it in the FIHashTable.
2. Frequent in local slide, also frequent in the past slide, then update the support in FIHashTable.
3. Infrequent in local slide, but frequent in other slides, then update the support in FIHashTable.
4. Infrequent in local slide, but infrequent in all other slides, then discard the itemset.

```
Algorithm 1: Update
Input: SLTree, FIHashTable, $W_{old}$, $W_{cur}$, δ- minimum threshold from (0 to 1)
Output: SLTree, FIHashTable, FI
// For handling obsolete window
$Sl_i \leftarrow W_{old}-W_{cur}$
For each $T_i$ from SLi
        Brach b from SLTree, Cand←PossComb(b)
        For each C ∈ Cand
                Decrease the support count of C in FIHashTable
        End for
Remove branch bfrom SLTree
For each Ti ∈ $Sl_{n+1}$  // For handling $Sl_{n+1}$
Update (SLTree) // insert each transaction into SLTree.
```

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

```
  Cand ← PossComb(Ti)
//Finding local frequent itemsets in a new slide
  For each C ∈ Cand
    if it is not hashed already
      then Hash(h) ← C, if it is not hashed already
    else increment count of h by 1
  For each h ∈ Hash
    If Support(h) ≥ (δ × (L_N/(L_N + L_U))) || δ
      Then LF ← LF + h
    Else LIF ← LIF + h
  For each l ∈ LF
    if it is not created already
    FIHashTree ← hash (l)
    Count ← Verify(I, SLTree)
      Add Count value to the support count of I in FIHashTree
    Else Increment support by 1    // if it is already created.
  Frequent Itemsets (SLTree, δ)
```

```
Algorithm 2: Frequent Itemsets
Input: SLTree, δ- minimum threshold
Output: FI- Frequent Itemsets

For each h ∈ HashTree
  If Support (h) ≥ δ
    FI ← FI + h
```

```
Algorithm 3: Verify
Input: SLTree, I- Itemset
Output: item-count :- support count of itemset in the SLTree
item ← I^K
item-ptr ← Hashtable (item, SLTree)
item-next ← item-ptr -> adj
While( item-next = item -> ptr -> adj)
  Count ← item-ptr. sup
  While (item-ptr -> pred == I^(K-1) of item && K-1>0)
    Item-ptr ← item-ptr -> pred
      K-K-1
    End while
  If (K==1)
    Item-count ← item-count+count
  Item-ptr ← item→next
End while
Return item-count
```

Verify

It is a verification method to find the support of itemsets that are frequent in new slide and not in any other slides. We visit the last item of itemset I and continue till its predecessors match with the itemset, and cumulate the support count of itemset if its predecessor matches with itemset. We use adjacent pointer to visit all occurrences of the itemset I. Thus, we find the support of itemset I from the *SlideTree*.

VI. EXPERIMENTS

We consider the real world dataset that contains mobile browsing data [17] of smart phone for conducting experiments.It contains more than 7000 pages with size of 21.8 GB, which is divided into 23 different categories. To test the efficiency of update and verify method, we use a public dataset *kosarak* which has no tuple revisions. For comparison, we use naive method, which consider mining from scratch for each update and Fideo [7] which uses revision model with two tree data structures. Our framework is named *FIDE*(**F**requent **I**temsets from **D**ata Streams with **Ev**olving tuples),

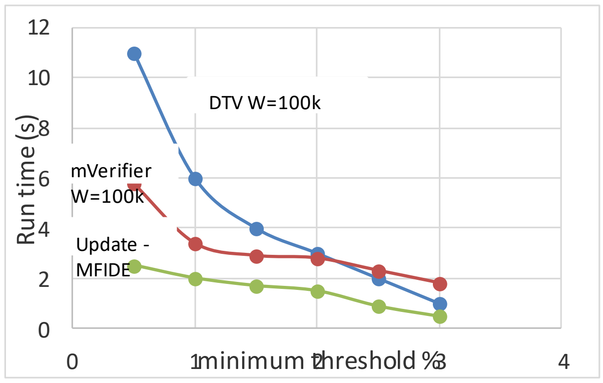

Figure 8: Execution time w.r.t threshold

Efficiency

We compare the performance of update with *DTV-DFV* [14]. *DTV-DFV* needs to re-build conditional trees for finding frequent itemsets. Thus it leads to many conditional trees. *Mverifier* [7] of *Fideo* uses two tree data structures instead of conditional trees. But the proposed FIDE framework uses one tree structure to maintain sliding window information and hash table to maintain frequent information which takes O(1) for insertion or any other activity. Thus it is proved that *FIDE* is efficient than others.

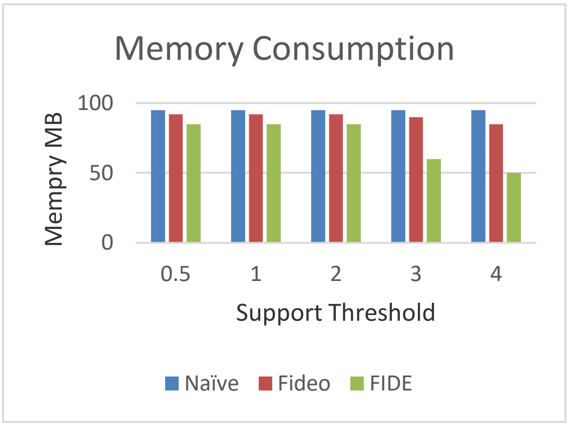

Figure 9: Memory Consumption in MB

VII. CONCLUSION

In this paper, we investigate how to extract frequent itemsets from the data streams when some of the tuples gets updated. To reduce unnecessary computation, we define conditions to determine whether itemset of a new slide can become frequent or not. We propose verify approach to find the support of local frequent itemset of new slide which is infrequent in all other slides. We design tree and hash table data structures to manage sliding window information. We design efficient update method that derives complete frequent itemsets. Experiments are conducted on standard dataset and it shows the efficiency and effectiveness of our proposal. In the future work we will do investigation on compact representation of itemsets over tuple –evolving streams.

REFERENCES

1. A. Ceglar and J. F. Roddick. Association mining. *ACM Comput. Surv.*, 38, July 2006.
2. 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.
3. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
4. B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, pages 179–188, 2008.
5. C. Zhang, F. Masseglia, and X. Zhang. Modeling and clustering users with evolving profiles in usage streams. In *TIME*, pages 133–140, 2012.
6. C. Zhang, F. Masseglia, and Y. Lechevallier. Abs: The anti-bouncing model for usage data streams. In *IEEE ICDM*, pages 1169–1174, 2010.
7. Chongsheng Zhang, Yuan Hao, MirjanaMazuran, Carlo Zaniolo, Hamid Mousavi, and FlorentMasseglia. Mining frequent itemsets over tuple-evolving data streams. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, pages 267–274. ACM, 2013
8. 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.
9. 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.
10. 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.
11. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.
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. L. Golab and M. T. Ozsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD Conference*, pages 658–669, 2005.
14. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.
15. P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.
16. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.
17. R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowl. Inf. Syst.*, 1:5–32, 1999.
18. 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.
19. 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.