Stream Mining of Frequent Sets with Limited Memory

Juan J. Cameron
University of Manitoba
Winnipeg, MB, Canada
umcame33@cs.umanitoba.ca

Alfredo Cuzzocrea
ICAR-CNR & Uni. Calabria
Rende, CS, Italy
cuzzocrea@icar.cnr.it

Carson K. Leung*
University of Manitoba
Winnipeg, MB, Canada
kleung@cs.umanitoba.ca

ABSTRACT

With advances in technology, streams of data are produced in many applications. Efficient techniques for extracting implicit, previously unknown, and potentially useful information (e.g., in the form *frequent sets*) from data streams are in demand. Many existing stream mining algorithms capture important streaming data and assume that the captured data can fit into main memory. However, problem arose when the available memory is so limited that such an assumption does not hold. In this paper, we propose a novel data structure called *DSTable* to capture important data from the streams onto the disk. The DSTable can be easily maintained; it can be applicable for mining frequent sets from datasets, especially in limited memory environments.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms; Design; Experimentation; Management; Performance; Theory

Keywords

Data mining, data streams, data structure, frequent patterns, matrix structure

1. INTRODUCTION & RELATED WORK

Frequent set mining searches for implicit, previously unknown, and potentially useful sets of frequently co-occurring items (i.e., frequent itemsets). As the mined frequent sets often serve as building blocks to other mining tasks (e.g., constrained association rule or pattern mining [14, 17]), numerous frequent set mining algorithms have been proposed since the introduction of the classical Apriori algorithm [1]. An example is FP-growth [8], which builds a *Frequent Pattern tree* (*FP-tree*) to capture the information of static databases (DBs) of precise data so that frequent sets can be mined from the FP-tree. Other examples include UV-Eclat [2], UF-growth [15] and U-VIPER [16], which find frequent sets from static DBs of uncertain data.

*Corresponding author: C.K. Leung.

With the automation of measurements and data collection, tremendously big volume of data has been produced in many application areas. The increasing development and use of large numbers of sensors has added to this situation. These advances in technology have led to streams of data [18, 19]. In order to be able to make sense of these data, stream mining algorithms are needed [5, 9, 20].

When compared with the mining from *static* databases, mining from *dynamic* data streams is more challenging due to the following properties of data streams:

Property 1: Data streams are continuous and unbounded. To find frequent sets from streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Hence, we need some data structures to capture the important contents (e.g., recent data than older data because users are usually more interested in the former) of the data streams.

Property 2: Data in the streams are not necessarily uniformly distributed; their distributions are usually changing with time. A currently infrequent set may become frequent in the future, and vice versa. So, we have to be careful not to prune infrequent sets too early; otherwise, we may not be able to get complete information such as frequencies of certain sets (as it is impossible to retract those pruned sets).

To mine frequent sets from data streams, several approximate and exact algorithms have been proposed. *Approximate* algorithms (e.g., FP-streaming [6], UF-streaming [11], DUF-streaming [12]) focus mostly on efficiency. Due to approximate procedures, these algorithms may find some infrequent sets or miss frequency information of some frequent sets (i.e., some false positives or negatives).

Alternatively, an *exact* algorithm mines truly frequent sets (i.e., no false positives and no false negatives) by (i) constructing a *Data Stream Tree* (*DSTree*) [13] to capture contents of the streaming data and then (ii) recursively building FP-trees for projected DBs based on the information extracted from the DSTree.

The above global DSTree and subsequent local FP-trees (for α-projected DBs where α are itemsets such as $\{a\}$, $\{a, b\}$ & $\{a, b, c\}$) are assumed to fit into main memory. While this assumption holds in many situations, there are situations where *not all* local FP-trees can fit into memory. To handle these situations, *Data Stream Projected trees* (*DSP-trees*) [10] can be built for x-projected DBs (where x is an item) as alternatives to FP-trees for α-projected DBs.

Along this direction, what if the global DSTree does not fit into main memory? Although there are some works [3, 4, 7] that use disk-based structure for mining, they mostly mine frequent sets from *static* DBs. Our **key contribution** of this paper is our proposal of a simple yet powerful on-disk data structure called *Data Stream Table* (*DSTable*), which captures relevant contents of the dynamic data streams so that frequent sets can be mined when using the *sliding window model.*

2. OUR DSTable STRUCTURE

Given (i) a stream of uncertain data and (ii) a limited memory environment, our proposed **Data Stream Table (DSTable)** structure captures the important contents of the streaming data onto the disk. Specifically, the DSTable is a two-dimensional table that captures the contents of transactions in all batches in the current sliding window.

Each row of the DSTable represents a domain item. Due to the dynamic nature and Property 2 of data streams, frequencies of items are continuously affected by the insertion of new batches (and removal of old batches) of streaming data. Arranging the items (i.e., rows of the DSTable) in frequency-dependent order may require frequent rearrangement. Hence, in the DSTable, items are arranged according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the construction of the DSTable. Consequently, the DSTable can be constructed using only a single scan of the data stream.

Each table entry (i) represents an item within a transaction in a batch of streaming data and (ii) points to the next item in transaction (with "EOT" representing the end of transaction).

When dealing with sliding window, we also need to be able to identify transactions in one batch from another. A naive solution is to capture transaction IDs. Observing that *all* transactions in an older batch are removed when the window slides, an alternative solution is to capture batch IDs. Observing that all transactions in the same batch bare the same batch ID, we avoid the redundancy of capturing repeated batch IDs. Specifically, our solution is to keep track of the boundaries between batches for each row in the DSTable. See Example 1.

Example 1. Consider the following streaming transactions:

Batch	Transactions	Contents
first	t_1	$\{a, c, d, e\}$
	t_2	$\{a, d\}$
	t_3	$\{a, b\}$
second	t_4	$\{a, e\}$
	t_5	$\{a, c, d, e\}$
	t_6	$\{c\}$
third	t_7	$\{a\}$
	t_8	$\{a, b, e\}$
	t_9	$\{a, c, d\}$

Let the window size w be 2 batches (indicating that only two batches of transactions are kept in the current window). Then, when the first two batches of transactions in the stream flow in (at time T_2), we capture contents of these transactions into the following DSTable:

T_2	Boundaries		1st	2nd	3rd	4th	5th
Row a	3rd	5th	(c,1st),	(d,2nd),	(b,1st);	(e,2nd),	(c,2nd);
Row b	1st	1st	EOT;				
Row c	1st	3rd	(d,1st);	(d,3rd),	EOT;		
Row d	2nd	3rd	(e,1st),	EOT;	(e,3rd);		
Row e	1st	3rd	EOT;	EOT,	EOT;		

The first entry in Row a points to "(c,1st)", i.e., the 1st entry of Row c. Its content "(d,1st)" refers to the 1st entry of Row d. Its content "(e,1st)" then refers to the 1st entry of Row e. Its content "EOT" indicates the end of transaction. This sequence of DSTable entries represents $t_1 = \{a, c, d, e\}$. Similarly, $t_2 = \{a, d\}$ is represented by the sequence with the second entry of Row a pointing to the 2nd entry of Row d. Moreover, we also record the last DSTable entries of item a (i.e., "3rd" & "5th" for Batches 1 & 2) to indicate the boundaries between batches. In other words, the 1st–3rd entries of Row a are for transactions in Batch 1, and the 4th–5th entries of Row a are for transactions in Batch 2.

When the window slides (at time T_3), the DSTable can be maintained by (i) removing those transactions for older batches (e.g., Batch 1), (ii) inserting transactions for new batches (e.g., Batch 3), and (iii) updating the boundary information. The updated DSTable becomes the following:

T_3	Boundaries		1st	2nd	3rd	4th	5th
Row a	2nd	5th	(e,1st),	(c,1st);	EOT,	(b,1st),	(c,3rd);
Row b	0th	1st	;(e,3rd);				
Row c	2nd	3rd	(d,1st),	EOT;	(d,2nd);		
Row d	1st	2nd	(e,1st);	EOT;			
Row e	2nd	3rd	EOT,	EOT;	EOT;		

This DSTable captures the information for Batches 2 & 3. □

3. MINING WITH DSTable AND DSP-trees

The DSTable is kept up-to-update when the window slides. As such, the mining can be "delayed" until it is needed. So, when the user requests for frequent sets, we extract from this global DSTable and build a local DSP-tree for each domain item. Each node in the DSP-tree contains (i) an item x and (ii) a counter. The value of this counter is initially set to the frequency of x on that tree path and is decremented during the mining process until it reaches 0 (cf. value remains unchanged in the FP-tree).

To mine frequent sets from the DSP-tree built for $\{x\}$-projected DB, we locate a node containing x and follow the tree path from x to the root. This tree path gives us a set $(\{x\} \cup \alpha)$, where α is an itemset comprised of items on such a tree path. The frequency of the set $(\{x\} \cup \alpha)$, or each of its subsets, equals the value of the counter of x. If the set (or its subset) is newly generated, we keep track of its frequency; otherwise, we increment the frequency of the set. Then, we subtract the value of the counter of x from the counter for each of the nodes along this tree path. By so doing, we eliminate the need for recursive constructions of FP-trees for projected DBs. See Example 2.

Example 2. Let us continue with Example 1 and set the user-specified threshold *minsup*=2. At time T_2, we first extract transactions starting from each entry in Row a. We obtain $\langle a, c, d, e\rangle$, $\langle a, d\rangle$, $\langle a, b\rangle$, $\langle a, e\rangle$ & $\langle a, c, d, e\rangle$, which form a DSP-tree shown in Figure 1. Note that item b is infrequent and thus pruned. From the path $\langle a$:5, c:2, d:2, e:2$\rangle$, we generate $\{a, c, d, e\}$:2 and all its seven subsets containing a (and all with frequencies 2). We then decrement counter values of the four nodes on this path by 2. Similarly, from the path $\langle a$:3, e:1$\rangle$, we increment the accumulative frequencies for $\{a\}$ & $\{a, e\}$ both to 2+1=3 and decrement the counter values of a from 3 to 2 and of e from 1 to 0. From the path $\langle a$:2, d:1$\rangle$, we increment the accumulative frequency for $\{a\}$ to 3+1=4 & that for $\{a, d\}$ to 2+1=3 and decrement the counter values of a from 2 to 1 and of d from 1 to 0. Then, from the path $\langle a$:1$\rangle$, we increment the accumulative frequency for $\{a\}$ to 4+1=5 and decrement the counter value of a from 1 to 0. Finally, we find frequent sets $\{a, c, d, e\}$:2, $\{a, c, d\}$:2, $\{a, c, e\}$:2, $\{a, d, e\}$:2, $\{a, c\}$:2, $\{a, d\}$:3, $\{a, e\}$:3 and $\{a\}$:5. Afterwards, we build DSP-trees for b, c & d in a similar fashion in order to find all other frequent sets from the current sliding window of the streaming data. □

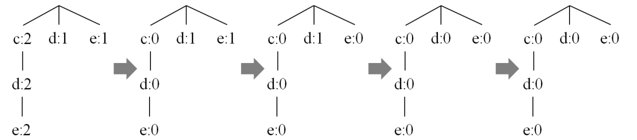
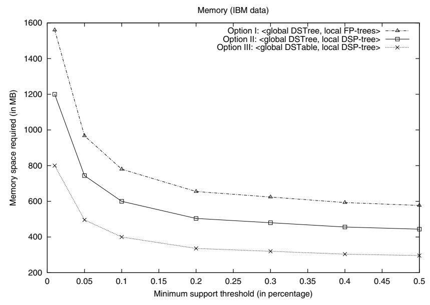


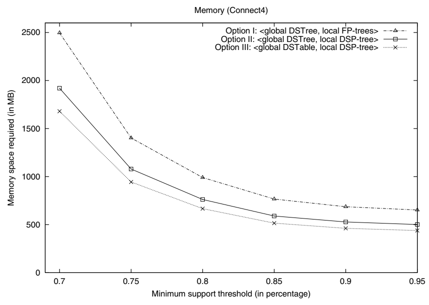
Figure 1: DSP-tree for the $\{a\}$-projected DB.

4. EVALUATION RESULTS

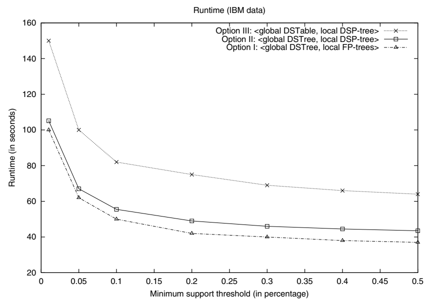
For evaluation, we compared three mining options: Option I uses a global DSTree and local FP-trees for α-projected DBs [13], Option II uses a global DSTree and a local FP-tree



(a) Main memory consumption for IBM data (sparse data)



(b) Main memory consumption for connect4 (dense data)



(c) Runtime for IBM data (sparse data)

Figure 2: Experimental results of our proposed DSTable.

for each x-projected DB [10], and Option III uses a global DSTable and a local FP-tree for each x-projected DB.

We used different DBs including IBM synthetic data, real-life DBs from the UC Irvine Machine Learning Depository (e.g., connect4 data), and those from the Frequent Itemset Mining Implementation (FIMI) Dataset Repository. IBM synthetic data are generated by the program developed at IBM Almaden Research Centre [1]. The data contain 1M records with an average transaction length of 10 items, and a domain of 1,000 items. We set each batch to be 0.1M transactions and the window size to be $w = 5$ batches. All experiments were run in a time-sharing environment in a 1 GHz machine. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os; it includes the time for DSTable construction and frequent set mining steps.

We summarized our observations as follows: In terms of accuracy, all three options give the same mining results.

The three options required various amounts of main memory space. For instance, Option I required the largest main memory space as it stores one global DSTree and multiple local FP-trees in main memory. Option II requires less memory as it stores at most one global DSTree and one local DSP-tree at any time during the mining process. Option III requires the *smallest* main memory space because our proposed DSTable is a disk-based structure. See Figures 2(a) & (b).

Runtime performance of the three options also varied. Both Options I and II took almost the same amount of time, whereas Option III took slightly longer. This is because the latter need to read from disk whereas the former two just read from main memory. See Figure 2(c). It is important to note that reading from disk would be a *logical choice* in a limited main memory environment.

Results on additional experiments showed that (i) the runtime decreased for all three mining options when *minsup* increased (as indicated in Figure 2(c)) and (ii) the DSTable was scalable with respect to the number of transactions in the streaming data.

5. CONCLUSIONS

Our key contribution of this paper is to provide users with a simple yet powerful alternative structure for efficient tree-based frequent set mining from data streams. Specifically, our proposed DSTable structure (i) captures the transactions in a sliding window and (ii) arranges transaction items according to some canonical order that is unaffected by changes in item frequency. Projected trees can then be built to mine frequent sets. As ongoing work, we are designing structures optimized for capturing dense/sparse streaming data so that frequent sets can be mined.

Acknowledgments. This project is partially supported by NSERC (Canada) in the form of research grants.

6. REFERENCES

[1] R. Agrawal & R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB 1994*, pp 487–499.
[2] B.P. Budhia, A. Cuzzocrea & C.K. Leung. Vertical frequent pattern mining from uncertain data. In *Proc. KES 2012*, pp. 1273–1282.
[3] G. Buehrer et al. Out-of-core frequent pattern mining on a commodity. In *Proc. ACM KDD 2006*, pp. 86–95.
[4] M. El-Hajj & O.R. Zaïane. Inverted matrix: efficient discovery of frequent items in large datasets in the context of interactive mining. In *Proc. ACM KDD 2003*, pp. 109–118.
[5] M.M. Gaber et al. Data stream mining. *Data Mining and Knowledge Discovery Handbook 2010*, pp. 759–787.
[6] C. Giannella et al. Mining frequent patterns in data streams at multiple time granularities. *Data Mining: Next Generation Challenges and Future Directions*, AAAI/MIT Press (2004), ch. 6.
[7] G. Grahne & J. Zhu. Mining frequent itemsets from secondary memory. In *Proc. IEEE ICDM 2004*, pp. 91–98.
[8] J. Han et al. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD 2000*, pp. 1–12.
[9] R. Jin & G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. In *Proc. IEEE ICDM 2005*, pp. 210–217
[10] C.K. Leung & D.A. Brajczuk. Efficient mining of frequent itemsets from data streams. In *Proc. BNCOD 2008*, pp. 2–14.
[11] C.K. Leung & B. Hao. Mining of frequent itemsets from streams of uncertain data. In *Proc. IEEE ICDE 2009*, pp. 1663–1670.
[12] C.K. Leung & F. Jiang. Frequent itemset mining of uncertain data streams using the damped window model. In *Proc. ACM SAC 2011*, pp. 950–955.
[13] C.K. Leung & Q.I. Khan. DSTree: a tree structure for the mining of frequent sets from data streams. In *Proc. IEEE ICDM 2006*, pp. 928–932.
[14] C.K. Leung & L. Sun. A new class of constraints for constrained frequent pattern mining. In *Proc. ACM SAC 2012*, pp. 199–204.
[15] C.K. Leung et al. A tree-based approach for frequent pattern mining from uncertain data. In *Proc. PAKDD 2008*, pp. 653–661.
[16] C.K. Leung et al. Mining probabilistic datasets vertically. In *Proc. IDEAS 2012*, pp. 199–204.
[17] C.K. Leung et al. Mining uncertain data for frequent itemsets that satisfy aggregate constraints. In *Proc. ACM SAC 2010*, pp. 1034–1038.
[18] O. Papapetrou et al. Sketch-based querying of distributed sliding-window data streams. In *Proc. VLDB 2012*, pp. 992–1003.
[19] S. Tirthapura & D.P. Woodruff. A general method for estimating correlated aggregates over a data stream. In *Proc. IEEE ICDE 2012*, pp. 162–173.
[20] K. Yu et al. Mining emerging patterns by streaming feature selection. In *Proc. ACM KDD 2012*, pp. 60–68.