Discovering Frequent Tree Patterns over Data Streams

Mark Cheng-Enn Hsieh
BenQ Corporation
Hsinchu, Taiwan 30078, R.O.C.
Mark.Hsieh@BenQ.com

Yi-Hung Wu
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 30013, R.O.C.
dr824349@cs.nthu.edu.tw

Arbee L.P. Chen[*]
Department of Computer Science
National Chengchi University
Taipei, Taiwan 11605, R.O.C.
alpchen@cs.nccu.edu.tw

ABSTRACT

Since tree-structured data such as XML files are widely used for data representation and exchange on the Internet, discovering frequent tree patterns over tree-structured data streams becomes an interesting issue. In this paper, we propose an online algorithm to continuously discover the current set of frequent tree patterns from the data stream. A novel and efficient technique is introduced to incrementally generate all candidate tree patterns without duplicates. Moreover, a framework for counting the approximate frequencies of the candidate tree patterns is presented. Combining these techniques, the proposed approach is able to compute frequent tree patterns with guarantees of completeness and accuracy.

Keywords: Data Mining, Data streams, Tree patterns

1. INTRODUCTION

Depending on various application domains, the frequent tree patterns discovered from tree-structured data streams can be of extensive use. Consider an online shopping website, which is browsed by thousands of people every few minutes. The web access log is produced in the form of trees to record the browsing behavior of each user. The large-volume and fast-accumulation of data constitute a tree-structured data stream. Frequent tree patterns discovered from this data stream can facilitate decision making for the website management. For the reason, Asai et al. [2] propose an online algorithm, *StreamT*, to continuously discover frequent subtrees on data streams by using a subtree generation scheme like the *rightmost expansion* techniques in [1][6]. It adopts a method similar to the one for online association rule mining in [3] to store all candidate subtrees. However, it neither provides any accuracy guarantee on the result set nor the error bound on the estimated frequency count of a subtree.

Yang et al. [5] present another result on frequency counting for tree-structured data streams. By grouping the query patterns that are in the form of trees into batches, an Apriori-based method for candidate generation is designed. Since it needs a lot of containment tests to compute the frequency count of each candidate subtree, the proposed algorithm, *XQSMiner*, limits the search space of candidates and allows that only some of these candidates should be examined. These strategies highly improve the performance and the experimental results also show its scalability. However, XQSMiner aims at discovering only the rooted subtrees due to its purpose for particular applications. As a result, it cannot be directly applied to the problem considered in this paper.

In this paper, we investigate the problem of mining all frequent *labeled ordered subtrees* over a tree-structured data stream. Instead of mining frequent subtrees in a static dataset, we propose an online algorithm named *STMer (Stream-Tree-Miner)* for the streaming environment. The result set can be derived without storing or performing multiple scans over the data stream.

The remainder of this paper is organized as follows. In Section 2, the basic terminologies and the problem formulation are presented. Section 3 describes the main components of STMer. Section 4 shows the experiment result and then we conclude the paper in Section 5.

2. PRELIMINARIES

2.1 Basic Terminologies

A labeled ordered tree T stands for a tree that is identified by not only the labels on its nodes but also the order among the siblings. As a result, switching any two siblings in T can result in a different tree.

In STMer, we represent such tree as a string to enable efficient subtree generation. Initially, an empty string is given. During the preorder traversal of a tree, whenever a node is visited, the label and the level associated with it are combined into a pair and appended to the string. In this way, the string representing a given tree can be uniquely determined and vice versa.

Similarly, since a tree can be represented as a string

[*]Contact author

consisting of its nodes by a preorder traversal, the data stream composed of trees can be viewed as an unbounded node sequence. For the sake of generality, STMer works in a node-by-node manner, although it is easy to adapt STMer to work in a tree-by-tree manner.

2.2 Problem Formulation

Let the set of trees recovered from the node sequence N_u be denoted as D_i, where the variable i indicates the number of nodes received so far. It is obvious that the volume of D_i increases as the data stream in. Figure 1 shows an example of an unbounded node sequence N_u and the set D_6, corresponding to the first six nodes (two trees) of N_u.

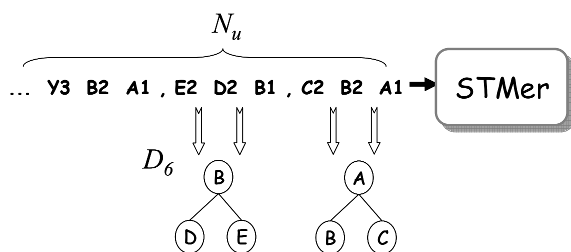

Figure 1: An unbounded node sequence

Definition 2.1 ***Frequency count***

Let *MatTree(S,T)* be the number of matches of tree S in tree T. The frequency count of S on the node sequence N_i is defined as follows:

$$freq(S,N_i) = \sum_{T \in D_i} ind(S,T), \text{ where } ind(S,T) = \begin{cases} 1, \text{if } MatTree(S,T) > 0 \\ 0, \text{otherwise} \end{cases}$$

In other words, the frequency count of S on N_i is equal to the number of trees in D_i in which S occurs.

Definition 2.2 ***Support***

The *support* of S on N_i is defined as $freq(S,N_i)/|D_i|$, where $|D_i|$ is the number of trees in D_i.

A subtree S is *frequent* if its support satisfies the user-specified threshold (*minimum support*) θ, where $0 \leq \theta \leq 1$. Based on these definitions, the problem we are solving in this paper can be defined as follow:

Given θ and an unbounded node sequence N_u, how can we discover all the frequent subtrees on N_i when the first i nodes are received, for any i?

Consider the data stream in Figure 1 and let θ be set as *0.6*. Since the first three nodes form a single tree encoded as *A1B2C2*, the result set for N_3 contains all the subtrees of the first tree. The result set then turns to be {*B1*} when the first six nodes are received. Note that the supports of all the subtrees except the subtree *B1* are reduced to *0.5* at this moment.

3. STREAM TREE MINER

3.1 An Overview

The framework of STMer is shown in Figure 2. The subtree inventor is in charge of the candidate subtree generation according to the nodes received so far. More precisely, when a node v is received from the data stream, it will generate all the subtrees in which node v is involved. All these subtrees are then used to update the candidate pool, named *GPT* (*Global Prefix Tree*), from which the result set can be derived.

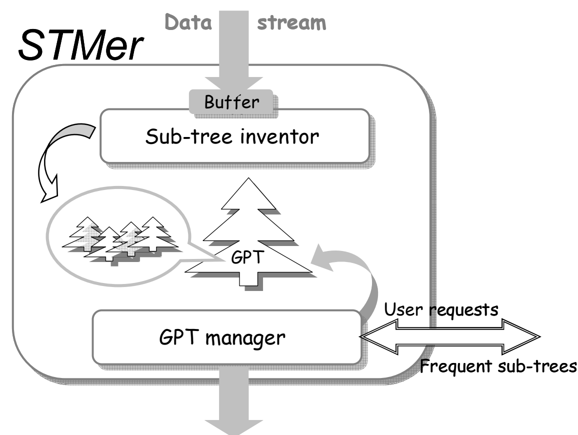

Figure 2: The framework of STMer

The GPT is a prefix tree constructed by sharing the common prefix of the strings converted from candidate subtrees. Each node in the GPT stands for a candidate subtree whose string corresponds to the path from the root of the GPT to the node. In this way, a node at the *i*th level of GPT refers to a candidate subtree with i nodes. Moreover, each of the candidate subtrees, is associated with its frequency count. Note that the root of the GPT, denoted as Φ, is a virtual node and its level is set as *0*.

The GPT manager is responsible for keeping the GPT from excessive expansion due to the huge number of distinct subtrees. In order to limit the memory usage, the GPT manager has to keep pruning the infrequent subtrees. However, this method may lead to the underestimation of frequency counts. Consider the candidate subtree T in the GPT with frequency count f as an example. If T is removed from the GPT, STMer will lose its frequency count accumulated in the GPT. If T is generated as a candidate again, the frequency count of T in the GPT will be underestimated by f. To address this issue, we adopt the concept of *Lossy Counting* [4] while designing the pruning policy of the GPT. As a result, the estimation error of a frequency count can be limited to a range specified by a user-given error parameter.

Finally, while STMer keeps performing the loop to manipulate the subtrees generated from the incoming data, the user can acquire the result set on demand in the meantime. The GPT is examined to find the results, i.e. the subtrees with frequency counts that are not smaller than $(\theta-\delta)N$. The value $(\theta-\delta)N$ is derived from the theoretical basis of Lossy Counting where N is the total number of trees currently received.

3.2 Subtree Generation

The process of subtree generation is started whenever a new node is received. The subtrees generated are added to the *APT* (*Augmented Prefix Tree*), which is used to temporarily store all the subtrees of the tree currently received. The structure of the APT is similar to that of the GPT, except that it merely stores the subtrees of a single tree.

The basic idea is to generate all the subtrees having node v from tree T_D at the moment node v is received, where T_D is the current tree recovered from the node sequence ending at node v. Every node triggers off the above operation exactly once when it arrives and at that moment it is called the *tail*. The parent of the tail in T_D is called the *branching point*. The subtrees that are previously generated and contain the branching point are called the *extensible subtrees*. The strategy named *tail-expansion* appends the tail to each of the extensible subtrees to generate new subtrees.

STMer carries out the above procedure by storing the subtrees in the APT as a well-organized structure. First, the subtrees stored in the APT are in the form of strings so that the common prefix among strings can be shared. Figure 4 shows the APT after the node (*B,2*) in Figure 3 has been processed.

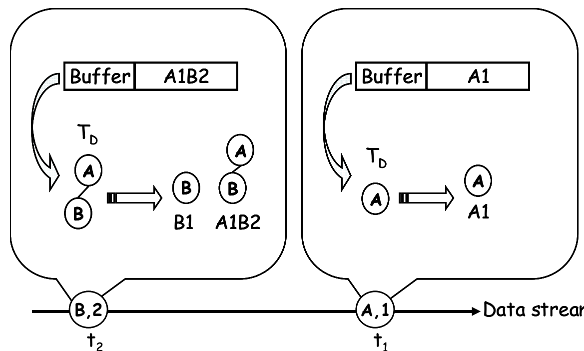

Figure 3: An illustration of subtree generation

The three nodes in the APT mean the subtrees *A1*, *A1B2*, and *B1*, respectively. Note that the node *A1* in the APT is created when the node (*A,1*) is received.

Second, to enforce the tail-expansion strategy, searching the APT for the extensible subtrees can be time-consuming. Therefore, in the APT the nodes with the same label are linked together to facilitate not only the searching but also the insertion of new subtrees. As a result, each label is associated with a sequence of horizontal links as Figure 4 depicts.

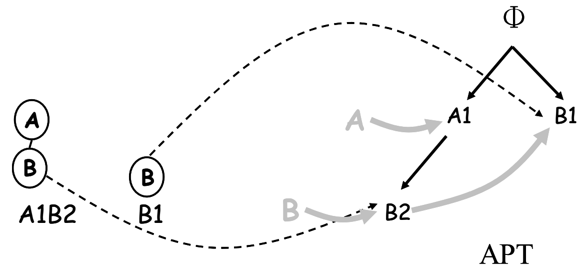

Figure 4: The APT with horizontal links

Figure 5 continues the process in Figure 3. For the node (*C,2*) received at t_3, STMer finds the branching point *A1* in the buffer to locate the corresponding node on the APT by using the horizontal links of label *A*. The nodes in the subtree rooted at node *A1* correspond to exactly all the extensible subtrees. To append the tail to each of these subtrees, STMer just adds a node *C2* to be a descendant of each of the corresponding nodes. As indicated in Figure 5, the new nodes stand for the two subtrees *A1C2* and *A1B2C2* respectively. After the tail-expansion, the single-node tree *C1* is also inserted to produce the APT with six nodes.

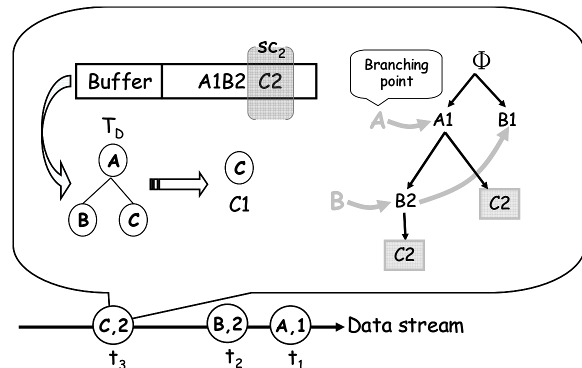

Figure 5: The use of horizontal links in the APT

Based on the concepts, we design an advanced procedure of subtree generation, outlined in Algorithm 1, which allows the entire *scope*, i.e. the string segment in which the node levels are increasing, to be the tail. First of all, the subtree generation is postponed until all the nodes in the same scope are gathered in the buffer (getscope). Once a scope is obtained, the scope is regarded as the tail to generate the subtrees with respect to the entire scope (in line 1). After that, all the suffix strings of the scope are enumerated (in line 2) and then inserted into the APT (in line 3). In this way, the subtrees can be generated in a scope-by-scope manner. Note that for the first scope the tail-expansion is skipped since at that moment the APT is empty.

```
Algorithm 1  Advanced sub-tree-generation
Input:
      A data node v_t = (label, level) and the local candidate pool APT.
Output:
      A set of candidate sub-trees maintained in APT.
Variable:
      A set S for storing the suffix paths.
Method:
      /* append incoming node to the buffer and obtain the scope */
      B[v_t] = (v_t.lable, v_t.level);
      v_t.scope = getscope(B, v_t);
      /* scope expansion */
      If not first scope do
1.        APT = tail-expansion(v_t.scope, APT);
      End
      /* suffix path enumeration */
2.    S = suffix(v_t.scope);
      /* insert suffix paths into APT */
3.    APT = insertion(S, APT);
      return APT;
```

3.3 Subtree Maintenance

The management of candidate subtrees in the GPT can be conceptually divided into two phases—the *merging* phase for inserting the subtrees in the APT into the GPT and the *maintenance* phase for preventing the GPT from excessive expansion. The policy for pruning the candidates in the GPT follows the theoretical basis of Lossy Counting. For this reason, we first introduce its underlying concept and then describe how STMer uses it in the mechanisms for the two phases.

The goal of Lossy Counting proposed in [4] is to keep monitoring the data streams composed of items to report the ones with frequency counts exceeding the user-specified threshold. A data stream is conceptually divided into buckets of $w=\lceil 1/\delta \rceil$ items, where δ stands for the error parameter. Each time the last item of a bucket is received, all the counters of the items in the candidate pool are decreased by one. If there are N items received, the frequency counts of the items in the candidate pool are underestimated by at most $\lfloor N/w \rfloor = \lfloor \delta N \rfloor$. To ensure the completeness of the result set, all the items with the estimated frequency counts exceeding $(\theta-\delta)N$ are outputted, where θ is the minimum support specified by the user.

The maintenance of the candidate pool in the Lossy Counting is described as follows. When an item d is received, the candidate pool C is looked up to find and increase the corresponding counter by one. If it does not exist, a new counter in the form of $(d, d.f, d.\Delta)$ is added to C, where $d.f$ is the accumulated frequency count ($=1$) of d and $d.\Delta$ holds the maximum possible error ($=\lfloor \delta N \rfloor$) in $d.f$ due to the underestimation. Note that $d.\Delta$ remains unchanged once it is assigned, while $d.f$ is increased as more copies of d are received from the data stream. In order to prevent the candidate pool from excessive expansion, whenever a bucket boundary is reached (N mod $w = 0$), all the items in C are examined once and an item d is pruned if $(d.f + d.\Delta) \leqq \delta N$.

The above policy is adopted by the GPT manager in STMer. Consequently, the GPT is controlled in a reasonable size and can give the following guarantees on the result set:

1. Every subtree with support exceeding the minimum support θ is included in the result set.
2. The estimated frequency count of each subtree is less than the true count by at most δN, where δ is the error parameter and N is the number of trees received so far (rather than the number of nodes).

In the following, we describe how STMer performs like the Lossy Counting in the two phases respectively. For the merging phase, the entire APT is regarded as single item as that in the scenario of Lossy Counting and the GPT can be viewed as the candidate pool C. The steps of merging phase are outlined in Algorithm 2. When a subtree T in the APT is inserted into the GPT, the GPT is looked up to find the corresponding node. If the node exists, it is increased by one. Otherwise, a new node associated with the counter in the form of $(T, T.f, T.\Delta)$ is added to the GPT, where $T.f$ ($=1$) keeps the accumulated frequency count of T and $T.\Delta$ holds the maximum possible error ($=\lfloor \delta N \rfloor$) in $T.f$. This process is repeated until all the subtrees in the APT are processed. At last, the GPT size, i.e. the number of trees received so far, is increased by one.

```
Algorithm 2  Sub-tree-storage
Input:
      The local candidate pool APT.
Output:
      The global candidate pool GPT.
Method:
      /* merging phase */
      For each sub-tree S in APT do
         /* insertion */
         If  T in GPT && T = S do
            T.frequency ++;
         Else
            GPT = insert-sub-tree(S, 1, [δ*GPT.size]);
         End
      End
      /* update data trees received over data stream */
      GPT.size ++;
      return GPT;
```

The maintenance phase is also based on the pruning policy of Lossy Counting. Whenever the GPT size can be divided by $1/\delta$, the GPT manager traverses the GPT once and prunes the node T if $(T.f + T.\Delta) \leqq \delta N$. Consider the node T that is pruned due to the inequality $(T.f + T.\Delta) \leqq \delta N$. The estimated frequency count $T.f$ merely accumulates the counts after the creation of this counter. Therefore, the true frequency count $T.f_{true}$ must be smaller than or equal to $(T.f + T.\Delta)$,

where $T.\Delta$ is the maximum possible error due to the underestimation. By combining the above two inequalities, the fact $T.f_{true} \leqq \delta N \leqq \theta N$ reveals that if the node T is pruned, it must be infrequent. According to the Apriori property, saying that all the supersets of an infrequent set are infrequent, it can be proved that all the super-trees of an infrequent tree must be infrequent as well. We incorporate this property into the pruning policy so that all the descendants (super-trees) of the node that is pruned can also be pruned.

4. EXPERIMENT

In this section, we evaluate the processing time of STMer on a synthetic data set produced from a data generator provided by Zaki [6]. The generator first constructs a mother tree based on the given parameters including the maximum fan-out of a node (*=10*), the maximum depth of a tree (*=10*), and the number of nodes in the mother tree (*=100*). After that, *1,000,000* subtrees of the mother tree are extracted to constitute the dataset for experiments. The average size of a tree, i.e. the number of nodes, in the dataset is about *7.3*.

Base on the experiment setting, we compare STMer with the previously proposed method, StreamT. The main idea of StreamT is to generate candidates using a technique of sweeping the right-most branch in a virtual tree, which is constructed according to the nodes previously received. This idea is similar to the tail-expansion presented in this paper. However, since STMer adopts the advanced tail-expansion to deal with the subtree generation scope by scope, its performance is expected to be superior to that of StreamT. Figure 6 demonstrates the results, indicating that STMer merely requires about *70%* of the processing time compared with StreamT.

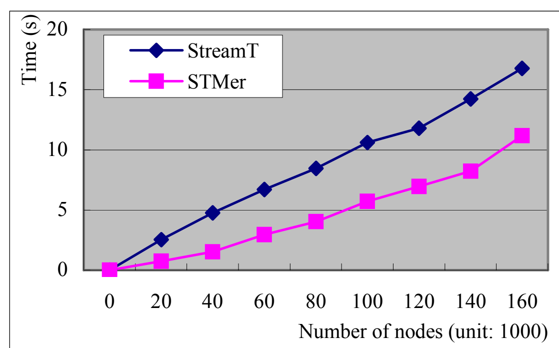

Figure 6: The comparison with StreamT

On the accuracy, StreamT tends to prune away all the candidates that are infrequent in order to keep the number of candidates in the pool as small as possible. In this way, the result set derived from the candidate pool cannot provide guarantees of an error bound on the estimated frequency counts and the completeness of the valid results. All these issues are addressed in STMer. Therefore, compared with StreamT, STMer is a more efficient and effective approach.

5. CONCLUSION

The main contributions of this paper are as follows:

1. Novel techniques for candidate subtree generation are proposed to incrementally generate the subtrees without duplicates in an efficient way.
2. A compact structure is designed to store all the candidate subtrees. It enables not only an efficient access of candidate subtrees but also an efficient candidate generation.
3. A framework providing estimated frequency counts of candidate subtrees is constructed. The result set returned from STMer is provided with a bound on the estimation error. Advanced pruning techniques are also provided to reduce the storage costs.

As a next step, it is interesting to extend STMer to discover frequent query patterns on a query engine for more intelligent caching policy. Besides, the generation of closed frequent subtrees is another challenging topic.

REFERENCES

[1] T. Asai, K. Abe, S. Kawasoe, et al., “Efficient Substructure Discovery form Large Semi-structure Data,” *Proc. of SIAM Intl. Conf. on Data Mining*, 2002.

[2] T. Asai, K. Abe, S. Kawasoe, et al., “Online Algorithms for Mining Semi-structured Data Streams,” *Proc. of IEEE Intl. Conf. on Data Mining*, 2002.

[3] C. Hidber, “Online Association Rule Mining,” *Proc. of ACM Intl. Conf. on Management of Data*, 1999.

[4] G.S. Manku and R. Motwani, “Approximate Frequency Counts over Data Streams,” *Proc. of Intl. Conf. on Very Large Data Bases*, 2002.

[5] L.H. Yang, M.L. Lee, and W. Hsu, “Finding Hot Query Patterns over an XQuery Stream,” *VLDB Journal—Special Issue on Data Streams*, 2004.

[6] M.J. Zaki, “Efficiently Mining Frequent Trees in a Forest,” *Proc. of ACM Intl. Conf. on Knowledge Discovery and Data Mining*, 2002.