Approximate Counting of Frequent Query Patterns over XQuery Stream

Liang Huai Yang, Mong Li Lee, and Wynne Hsu

School of Computing, National University of Singapore
{yanglh,leeml,whsu}@comp.nus.edu.sg

Abstract. One efficient approach to improve the performance of XML management systems is to cache the frequently retrieved results. This entails the discovery of frequent query patterns that are issued by users. In this paper, we model user queries as a stream of XML query pattern trees and mine for frequent query patterns in a batch-wise manner. We design a novel data structure called *D-GQPT* to merge the pattern trees of the batches seen so far, and to dynamically mark the active portion of the current batch. With the *D-GQPT*, we are able to limit the enumeration of candidate trees to only the currently active pattern trees. We also design a summary data structure called *ECTree* to incrementally compute the frequent tree patterns over the query stream. Based on the above two constructs, we present the frequent query pattern mining algorithm called *AppXQSMiner* over the XML query stream. Experiment results show that the proposed approach is both efficient and scalable.

Keywords: Stream Mining, XML Query Pattern, Tree Mining

1 Introduction

An efficient approach for improving the performance of query evaluation is to discover frequent query patterns and cache their results in anticipation of future retrievals. The problem of finding frequent query patterns arises in the context of search engines and XML query systems. Frequent query patterns are the ideal caching objects in XML caching system [5,13].

While there have been success in improving the query performance of XML management system for sets of XML queries, real-life applications typically involve XML queries which arrive in streams. Given the sheer volume of the query stream, it would be impractical to scan the queries twice. Instead, new summary data structures and optimization techniques have to be devised.

In this paper, we design a novel data structure called *D-GQPT* to merge all the query patterns seen so far. Mining proceeds in a batch-wise manner. By marking the active portion of the *D-GQPT* that is related to the current batch, we are able to enumerate only the rooted subtrees that are related to the query pattern trees involved in the current batch. In addition, we also propose a summary structure, the *ECTree*, to keep track of the mining result, and to incrementally compute the frequent tree patterns over the XQuery stream. Based on the *D-GQPT* and the *ECTree*, we develop an algorithm called *AppXQSMiner* to mine frequent query patterns over XML query stream. Experiment results show that the proposed approach is both efficient and scalable.

The rest of the paper is organized as follows. Section 2 briefly reviews some basic concepts and gives the problem statement. Section 3 presents the two structures that form the basis for the candidate generation and mining processes. Section 4 addresses the issues of candidate generation and frequent pattern mining over XQuery stream. Experiment results are given in section 5. We discuss related work in Section 6 and draw our conclusion in Section 7.

2 Preliminaries

In this section, we define the basic concepts and the problem we address.

2.1 Query Pattern Tree

XML queries can be modelled as query pattern trees(QPT). In addition to element tag names, a query pattern tree may also consist of wildcards "*" and relative paths "//". The wildcard "*" indicates the ANY label (or tag), while the relative path "//" indicates zero or more labels (descendant-or-self). We assume the query pattern trees do not contain sibling repetitions, that is, the siblings in a query pattern trees have distinct labels. Formally we define:

Query Pattern Tree: A query pattern tree is a rooted tree QPT=<V,E>, where V is the vertex set, E is the edge set. The root of a QPT is denoted by root(QPT). For each edge e=(v_1, v_2), node v_1 is the parent of node v_2. Each vertex v has a label, denoted by *v.label*, whose value is in {"//","*"}∪*tagSet*, where the *tagSet* is the set of all element and attribute names in the schema.

Rooted Subtree: Given a query pattern tree QPT=<V,E>, a rooted subtree RST=<V',E'> is a subtree of QPT if it satisfies the following conditions:
(1) *root*(RST)=*root*(QPT),and
(2) $V' \subseteq V$, $E' \subseteq E$.

We call a RST with k edges a k-edge rooted subtree and denote it as RST^k.

2.2 Mining XQuery Stream

As XML queries stream into the system, a sequence of query pattern trees is formed. Let S = QPT_1, QPT_2,..., QPT_N where N is the length of the current stream, that is, the number of query pattern trees seen so far. Mining the frequent query patterns in S implies discovering the frequent rooted subtrees in the current sequence. A rooted subtree RST matches a query pattern tree QPT in S, or we say, RST occurs in S, if there exists a QPT that includes the RST. The total occurrence of an RST in S is denoted by $freq(RST)$, and the support level *supp*(RST) is given by $freq(RST)/N$. We say that RST is σ-frequent in S if *supp*(RST) $\geq\sigma$ for some positive number σ.

Consider the query pattern trees and a 3-edge rooted subtree *RST* in Fig. 1. The rooted subtree *RST* occurs in QPT_1 and QPT_2 with a frequency and support of $freq(RST) = 2$ and $supp(RST) = 2/3$ respectively.

Transaction IDs,(TID) are often used to expedite the mining process. Here we associate each query pattern tree *QPT* with a unique *TID*, denoted as *QPT.TID*. This will be used in our mining algorithm to reduce the number of expensive tree inclusion tests.

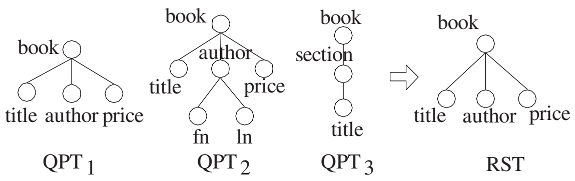


Fig. 1. Example of a Frequent Query Pattern Tree.

2.3 Counting Frequent Query Patterns

With *QPTs* streaming in, the set of frequent *RSTs* will evolve. Some of them may become infrequent, while other new frequent *RSTs* may emerge. The challenges faced in this streaming scenario are twofold:

(1) The mining process is incremental;

(2) Only one pass is allowed over the query stream.

A naïve method to count the frequencies of the pattern trees is to maintain the count of every possible rooted subtrees. However, this method is not practical. The reason is that the number of *RSTs* increases exponentially with the pattern size. A more feasible approach is to use the theoretical framework for approximate frequency count proposed in [9]. We adapt this framework to the tree mining scenario as follows:

Problem Statement: Given a support threshold σ $\in$(0,1) and an error parameter ϵ $\in$(0,1) such that $\epsilon \ll \sigma$, construct an algorithm to produce a set of frequent *RSTs* F along with their estimated frequencies on request.

For any *RST rst*, let $rst.count_{true}$ denote the true frequency of *rst*, and $rst.count_{app}$ denote the estimated frequency of *rst*. The *RSTs* produced have the following guarantees:

1. For any *rst* in the search space, if $rst.count_{true} \geq \sigma$N, then we have $rst \in F$. There are no false negatives.
2. For any *rst* that is considered to be frequent, we have rst.$count_{true} \geq (\sigma - \epsilon)$N.
3. For any *rst*, its estimated frequency $rst.count_{app} \geq rst.count_{true} - \epsilon$N.

Solution (adapted from [9]:) The incoming XQuery stream is conceptually divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$ *QPTs* each. Buckets are labelled with bucket *ids* starting from 1. The current bucket id is denoted as $b_{current}$, whose value is $\lceil \frac{N}{w} \rceil$. For a *rst*, we denote its true frequency in the stream seen so far by *rst.count*$_{true}$. Note that the values of N, $b_{current}$ and *rst.count*$_{true}$ will change as the stream progresses.

A summary data structure D contains a set of entries of the form (*rst*, *rst.count*$_{app}$,Δ), where *rst.count*$_{app}$ is the estimated frequency of *rst*, and Δ is the maximum possible error in *rst.count*$_{app}$.

However, it is not feasible to handle the query pattern trees one at a time. The reason is that this entails the enumeration of all candidate RST that the QPT contains, and the number of $RSTs$ may be exponential to the size of the QPT. Hence, a batch processing approach is taken.

Let β be the number of buckets in main memory in the current batch $\boldsymbol{B}$. Initially, the summary data structure D is empty. D is updated as follows:

- **Insert a new *RST***: For a RST rst, which is not found in D, its frequency count in the current batch is *rst.count*$_{app}$. If *rst.count*$_{app}$ $\geq \beta$, then add a new entry (*rst*, *rst.count*$_{app}$, $b_{current}$ - β) to D.
- **Update and prune an old *RST***: For each entry (*rst*, *rst.count*$_{app}$, Δ)$\in D$, the *rst.count*$_{app}$ is increased by the occurrence of *rst* in the current batch. If *rst.count*$_{app}$$+\Delta \leq b_{current}$, then this entry is deleted from D.

When a user requests the frequent $RSTs$ with threshold σ, then entries in S where *rst.count*$_{app}$ $\geq (\sigma - \epsilon)$N are output.$\diamond$

3 Data Structures

In this section, we describe two data structures that are designed to facilitate the mining of frequent $QPTs$ in the XQuery stream. The first data structure called *D-GQPT* merges all the query patterns seen so far. An active *D-GQPT* is obtained by marking the portion of the *D-GQPT* that corresponds to the current batch.

The second data structure called *ECTree* keeps track of the mining result. *ECTree* is used as an infrastructure to generate candidate $RSTs$, and to incrementally compute the frequent tree patterns over the XQuery stream.

3.1 *D-GQPT*

An important step in mining tree patterns is to enumerate all the frequent $RSTs$ in the stream. One method is to generate new $RSTs$ by expanding the rightmost branch with all possible labels. This method is shown to be inefficient in [13] because it tends to produce a large number of unnecessary candidates. A more efficient way is to construct a global query pattern tree $GQPT$ by merging the query pattern trees in the stream.

Fig. 2(a) shows the global query pattern tree ($GQPT$) obtained from the query pattern trees in Fig. 1. The nodes in the $GQPT$ are numbered using a pre-order traversal. Each node in a rooted subtree RST of the $GQPT$ has the same number as the corresponding node in $GQPT$ (see Fig. 2). A hash table is provided for the lookup of the mapping of each node and its label. This numbering scheme not only reduces the amount of memory used during the mining process, but it also simplifies the representation of the query pattern trees.

For example, RST^3 can now be represented as <1><2></2><3></3><8></8></1>. By removing the brackets and replacing each end tag with -1, the above representation can be further compacted to “1,2,-1,3,-1,8,-1”. Note that the last end tag can be omitted.

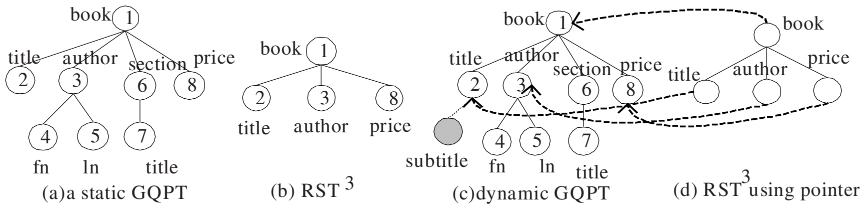


Fig. 2. Node Labelling Scheme.

We call the above method *static-GQPT*. It is only suitable for a single batch of queries. If a new batch of queries arrives, a new *GQPT* will need to be constructed. The numbering of nodes in the new *GQPT* may be different from the numbering in the previous *GQPT*, which invalidates all the node numbering in the previous *RSTs*. Therefore, instead of directly using the node numbering in the *GQPT* for the nodes in the *RST*, we make use of “pointers”. All the nodes in the *RSTs* use pointers to point to the *GQPT* nodes. Consequently, as the *GQPT* evolves, the original node numbers may change, but this will not affect the numbering of the *RSTs*. We call this dynamic method *D-GQPT*.

Example. Assume that for the first batch of queries, we obtain a *D-GQPT* as shown in Fig. 2(c) except that it does not have the node “subtitle”. A 3-edge RST^3 “1,2,-1,3,-1,8,-1” will be represented as shown in Fig. 2(d). Suppose a second batch of queries arrives. Without loss of generality, let the node “subtitle” be added as the child of “title”. If the static numbering scheme is used, then RST^3 will need to be re-labelled. However, by utilizing a pointer scheme, RST^3 is not affected. RST^3 is now represented as “1,2,-1,4,-1,9,-1”.

As the XML queries stream in and batches are formed, the size of the *D-GQPT* will increase. For the current batch, not all the nodes in the *D-GQPT* will be relevant. Making use of this *D-GQPT* to enumerate the RSTs will be inefficient since many RSTs which are not relevant to the current batch will be generated. We solve this problem by marking the nodes for the current batch when the *D-GQPT* is formed. However, we observe that the marked portion may not be complete.

For example, suppose “book/section//title” is marked as active, and the portion {section//title} in “book / section / section // title” is not active. However, the latter is included in the former path, and should be considered as relevant to the current batch. As a result, {section//title} should be marked as active.

Following this reasoning, we check each "inactive" node in the *D-GQPT* to see if the path from the root to this node is contained in some active path. The marked portion of the *D-GQPT* is called the active *D-GQPT*. When enumerating the RSTs, only the active *D-GQPT* is used.

3.2 *ECTree*

After obtaining the global query pattern tree, the problem of enumerating *RSTs* is now reduced to the problem of enumerating the *RSTs* in the active *D-GQPT*. We employ the *GQPT*-guided *RST* enumeration method proposed in [13]. Starting with all the possible 1-edge *RSTs*, denoted as candidate set C_1, we use the active *D-GQPT* to systematically guide the generation of 2-edge *RST* set C_2 level-wise by expanding the rightmost branch, from which 3-edge *RST* set C_3 are obtained, etc.

The above enumeration forms a search space. On closer examination, we find that the search space can be reorganized to yield a more efficient candidate generation scheme. Each candidate set can be divided into equivalence classes (EC), and each EC in turn, can be partitioned into two groups: G_{rmlne}- formed by right-most leaf node expansion; and G_{join} - formed by the join of two *RSTs*. For details see [13].

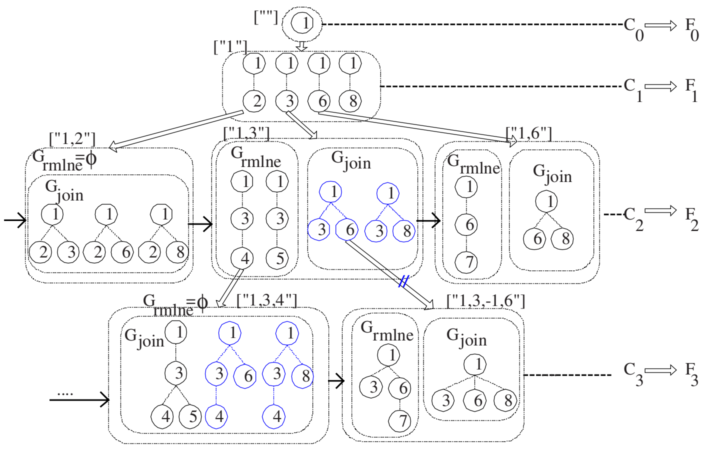


Fig. 3. The *ECTree* Summary Structure.

Based on the these concepts, we present the *ECTree* summary structure (see Fig. 3). Each node in the *ECTree* corresponds to an equivalence class, and we call it an *EC* node. An *EC* node is organized as a linked list of *RSTs* in ascending order. Each *RST* in the *ECTree* is associated with the following attributes:

1. *RST.current* - indicates whether the(existing) *RST* is part of current batch or not;

2. *RST.IsNew* - indicates that this *RST* does not appear in the *ECTree* and may be inserted in;
3. *RST.tidlist* - contains the information of which *QPTs* include this *RST* in the current batch;
4. $RST.count_{app}$ - the exact frequency count of the *RST* since it appears in the *ECTree*;
5. $RST.\Delta$ - the maximum possible error in $RST.count_{app}$. This is set when it is inserted into the *ECTree* for the first time, i.e., $b_{current}$-β.

4 Algorithms

The candidate generation process for stream data is different from the traditional setting. In the standard candidate generation step, when generating C_{k+1} from F_k, all the elements in F_k will participate and C_{k+1} does not exist. However, in the stream setting, since a batch-wise mining approach is taken, not all elements in F_k are related to the current batch. Further, C_{k+1} may exist. Such differences lead to a new candidate generation and pruning scheme.

We develop an efficient candidate generation algorithm *XQSRSTGen* that is based on *ECTree* and *D-GQPT*. New pruning strategies are applied to efficiently maintain the *ECTree*. Finally, we develop the mining algorithm *AppXQSMiner* for discovering the frequent query patterns over XQuery stream.

4.1 Candidate Generation

Recall that the *ECTree* keeps the frequent *RSTs* of the query pattern stream seen so far. When a new batch of queries is formed, the candidate *RSTs* that are related to the current batch have to be generated. We use the following result to generate the relevant *RSTs* for the current batch.

Theorem 1 *Given a RST of D-GQPT, RST is related to current batch, i.e., RST.current=true iff all the leaf nodes of this RST are active.* ◇

Corollary 1 (Matching Scheme) *: Only RSTs with RST.current=true need to be matched against the QPTs in the current batch.*◇

When the new *RSTs* that are relevant to the current batch of queries are generated, their *current* values will be set to true. If they do not exist in the *ECTree*, then they are inserted into the respective *EC* node. On the other hand, if they exist in the *ECTree*, then we set the corresponding *RST.current* to true. On generating C_{k+1} from F_k, we have the following result:

Theorem 2 (RST Generation Scheme) *: To generate* C_{k+1} *from* F_k *in ECTree, for each EC=* $\{RST_1^k, RST_2^k, \ldots, RST_N^k\} \in F_k$*, where EC is sorted in ascending order, and for each k-edge* $RST_i^k \in EC$ *, if* $RST_i^k.current$ *= true, then the following operations obtain the new EC(*RST_i^k*) induced by* RST_i^k *:*

(a) $G_{rmlne}(RST_i^k) = \{RST_{ir}^{k+1} | RST_{ir}^{k+1}$ *is the rightmost active leaf node expansion of* $RST_i^k\}$.
(b) $G_{join}(RST_i^k) = \{RST_{ij}^{k+1} \mid RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, $j = i+1, \ldots, N$, *and* $RST_i^k.current$ *true* $\wedge$ $RST_j^k.current = true\}$.

Consequently, $EC(RST_i^k) = G_{rmlne}(RST_i^k) \cup G_{join}(RST_i^k)$ *holds.* $\diamond$

By incorporating the *TIDs* into the mining process, we can reduce a large number of tree inclusion tests. We associate a list of *TIDs* with each *RST* that is contained in those *QPTs* in the current batch. Under our assumption that *QPTs* contains no repeated siblings, for a *RST* that is a join of other two *RSTs*, $RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, we have $RST_{ij}^{k+1}.tidlist = RST_i^k.tidlist \cap RST_j^k.tidlist$. Hence, only those *RSTs* in G_{rmlne} needs to undergo tree inclusion test against *QPTs*.

Fig. 4 gives a sketch of candidate generation algorithm *XQSRSTGen*. The procedure $rmlne(RST^k)$ performs the right most leaf node expansion while $join(RST_i^k, RST_j^k)$ carries out the join operation. Note that the candidate generation phase not only involves the modification of *RST*'s attribute values, but it also requires the maintenance of the *ECTree*. This is covered in the next section.

```
Algorithm: XQSRSTGen(ec,i,ECTree, D-GQPT)
Input: ec—k-edge EC; ECTree, D-GQPT—two data structure;
Output: k+1-edge EC of $rst_i^k$;
1. $rst_i^k$ =ec[i]; /*ec is an EC with RSTs in ascending order*/
2. if $rst_i^k$.current=true then /*only expand RSTs of current batch*/
3.     $G_{rmlne}$=rmlne($rst_i^k$);/*only expand it with active node in D-GQPT */
4.     for j=i+1 to |ec| do/*perform join*/
5.         $rst_j^k$ =ec[j];
6.         if($rst_j^k$.current=true) then/*only current RSTs need join*/
7.             $G_{join} \leftarrow join(rst_i^k, rst_j^k)$;
8.     newEC= $G_{rmlne} \cup G_{join}$;  /*form a new equivalence class */
9. return newEC;
```

Fig. 4. Algorithm *XQSRSTGen*.

4.2 Pruning Strategies and Maintenance of ECTree

In this section, we deal with the modification of an *RST*'s attribute values and maintenance of the *ECTree*. When a new batch of queries is formed, we need to update the information of the *RSTs* in the *ECTree*. New *RSTs* needs to be inserted and existing *RSTs* need to update their frequency counts. Infrequent *RSTs* need to be deleted from *ECTree*.

Before mining the new batch, all the existing *RSTs*' current values in *ECTree* are set to false. When a new batch is formed, the active portion of *D-GQPT* is also created which corresponds to the current batch. Let *D-GQPT*' denote the active portion of *D-GQPT*. By calling the candidate generation algorithm

XQSRSTGen, the *ECTree* will be updated as follows. For each RST^{k+1}, we check its corresponding *EC* node. There are two cases:

(1) RST^{k+1} does not exist in the ECTree.

If it is a new *RST*, then we set RST.IsNew = true, $RST^{k+1}.\Delta = b_{current}$ - β. If it is a *RST* of G_{join}, then we set $RST^{k+1}.count_{app} = |RST^{k+1}.tidlist|$. If $|RST^{k+1}.tidlist| < \beta$, then RST^{k+1} should be pruned away. If it is not pruned, then we insert it into the corresponding *EC* node. In the case where no such *EC* node exists, a new *EC* node will be created.

(2) RST^{k+1} exists in the *EC* node.

RST^{k+1}.current is set to true. If RST^{k+1} is a *RST* of G_{join}, then $RST^{k+1}.tidlist$ is updated, and early pruning is applied. We set $RST^{k+1}.count_{app} = RST^{k+1}.count_{app} + |RST^{k+1}.tidlist|$.

If $RST^{k+1}.count_{app} + RST^{k+1}.\Delta \leq b_{current}$, then we know that RST^{k+1} is not frequent. Based on the Apriori property, all the *RSTs* (including RST^{k+1}), which is the "superset" of RST^{k+1} will not be frequent and hence can be pruned.

Two types of descendants can be properly removed:

(a) Cut off the subtree induced by RST^{k+1} from the *ECTree*;

(b) Prune away those *RSTs* which are the join result with RST^{k+1}.

These *RSTs* appear in the subtrees induced by the *RSTs* before RST^{k+1} in the same *EC* node. We call these subtrees the affected subtrees. This pruning process is applied recursively down the subtrees.

Instead of searching the affected subtrees each time we find an infrequent *RST*, a quick optimization is to first find all the infrequent *RSTs* in the same *EC* node and then perform the pruning in a batch mode. We denote this procedure as *pruneECTree*(RST^{k+1}).

4.3 AppXQSMiner

The framework of *AppXQSMiner* is shown in Fig. 5. It reads in *QPTs* in the XML query stream and forms a batch of given BatchSize. During this period, *QPTs* are merged into *D-GQPT*, and an active *D-GQPT* is obtained when the batch is finalized. It initiates the mining process by calling *DFSMiner*.

The *DFSMiner* adopts a depth-first approach and has three main steps. For each $RST^k \in ec$, Line 2 generates or updates its EC node *newEC* in the *ECTree* by calling *XQSRSTGen*; Lines 3-5 match *RSTs* in G_{rmlne} of *newEC* against *QPTs* of the current batch and obtain their *tidlist*, where *Contain* fulfils the tree inclusion test; Lines 6-12 prune *RSTs* of G_{rmlne} and non-current *RSTs*. Note the *RSTs* for the current batch except *RSTs* in G_{rmlne} are pruned by *XQSRSTGen*.

5 Experimental Study

In this section, we evaluate the performance of *AppXQSMiner*. The mining algorithms were implemented in C++. We carried out experiments on a Pentium IV 2.4 GHz with 1 GB RAM, running under Windows XP.

```
Algorithm: AppXQSMiner(QPTStream,$\sigma$,$\varepsilon$,BatchSize)
Input: QPTStream—the stream of QPTs; $\sigma$--the minimum support;
         $\varepsilon$--allowed error tolerance;  BatchSize--#of QPTs in a batch;
Output:frequent patterns seen so far;
Initialise data structures: D-GQPT and ECTree;
Repeat:
   read from QPTStream and form a batch B; $b_{current}$ is the largest bucket ID of  B;
   merge QPTs into dynamic D-GQPT and form the active D-GQPT;
   call DFSMiner($ec_0$), where $ec_0$ =root node of ECTree;
function DFSMiner(ec)/*ec-the k-edge EC*/
1. for i=1 to |ec| do/*ec is an EC with RSTs in ascending order*/
2.       newEC=XQSRST-Gen(ec,i,D-GQPT,ECTree); /*ECTree is updated*/
3.       for each $qpt \in B$ do
4.            for each $RST^{k+1} \in G_{rmlne}$ and $RST^{k+1}$.current=true do
5.                 if Contains(qpt, $RST^{k+1}$) then $RST^{k+1}$.tidlist$\leftarrow$t.TID;
6.       for m=1 to |newEC| do/*only checking rmlne() and non-current RSTs*/
7.           $RST_m^{k+1}$ = newEC[m];
8.           if ($RST_m^{k+1} \in G_{rmlne}$ and $RST_m^{k+1}$.IsNew=true) then
9.               if (|$RST_m^{k+1}$.tidlist|<$\beta$) then  pruneECTree($RST_m^{k+1}$);
11.          if($RST_m^{k+1}$.current=false) then /* non-current RSTs */
12.              if ($RST_m^{k+1}.count_{app}$+$RST_m^{k+1}.\Delta \leq b_{current}$) then pruneECTree($RST_m^{k+1}$);
13.      DFSMiner(newEC);
```

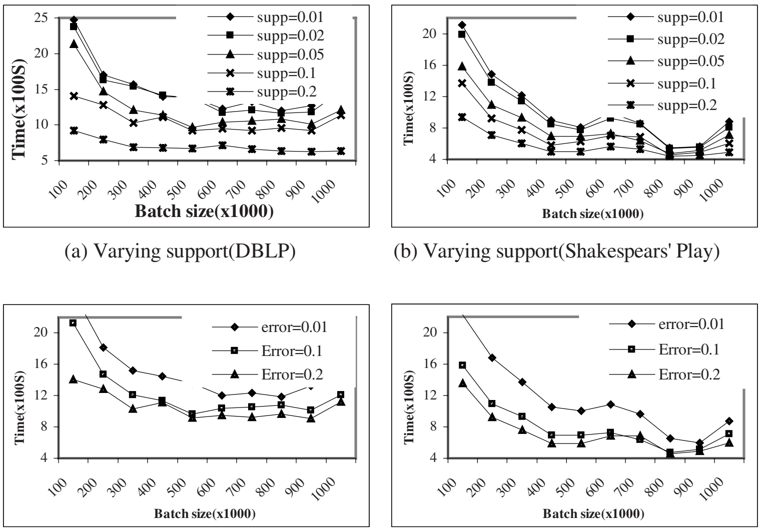
Fig. 5. Framework Of AppXQSMiner.

We examine how varying batch size, error rate(ϵ), stream length and support(σ) will affect *AppXQSMiner*. The default values of σ and ϵ are 0.05 and 0.1σ respectively. Two streams of 1 million XML query patterns are used. The first stream is generated from the DBLP DTD with a total of 98 nodes. The second stream is obtained from the Shakespears' Play DTD, or *SSPlay* for short, with a total of 23 nodes. Both query streams conform to the Zipfian distribution ($\theta = 0.5$).

5.1 Effect of Batch Size

The first set of experiments examines the effect of varying batch size on the performance on the algorithm. We first fix the error rate at 0.1 and vary the minimum support. The results for the DBLP and the *SSPlay* dataset are shown in Fig. 6(a) and (b). We observe that when the batch size is too small (less than 200,000), the performance deteriorates significantly because *RSTs* need to be enumerated and tested for a larger number of batches.

However, an increase in the batch size does not necessarily lead to a corresponding decrease in response time. The reason is that we are dealing with tree patterns instead of itemsets. If a large tree pattern appears in a batch, then a large number of *RSTs* need to be matched against the *QPTs* in this batch, regardless of the batch size. If the tree pattern includes "//", the cost of the inclusion test increases.



(a) Varying support(DBLP)

(b) Varying support(Shakespears' Play)

(c) Varying error rate(DBLP)

(d) Varying error rate(Shakespears' Play)

Fig. 6. Effect Of Varying Batch Size.

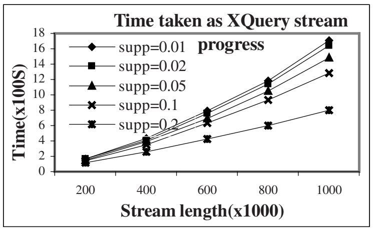
Both the DBLP and *SSPlay* query streams show similar trends. Even though we buffered 1000K *QPTs* in the main memory, the response times cannot be minimized. For the DBLP stream, more time is needed at 1000K *QPTs* because the DBLP has 98 nodes compared to the *SSPlay* . This also indicates that frequent tree pattern mining is CPU-intensive.

Fig. 6(c) and (d) show the results when we set the support at 0.05 and vary the error rate. We see that a smaller error rate leads to a larger response time. This is because smaller error rate results in larger bucket width, and thus fewer buckets(β) in a batch. The smaller β may require more *RSTs* to be maintained and matched. In other words, it takes time to compute a more accurate set of results.

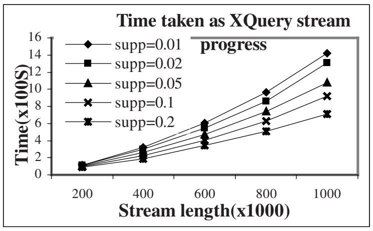
5.2 Effect of Stream Length

The final set of experiments demonstrates how the algorithm scales with the size of the query stream at different minimum support. The batch size is fixed at 200K and error rate at 0.05. Fig. 7 reveals that the time taken by *AppXQSMiner* is almost linear to the length of XQuery stream.

Overall, mining over the DBLP query stream incurs more time compared to mining over the *SSPlay* query stream. This is because the DBLP DTD is larger than the *SSPlay* DTD, and larger tree patterns tend to generate more rooted subtrees as candidates and consequently require more tree inclusion tests.



(a) DBLP(Batch Size=200K)



(b) SSPlay(Batch Size=200K)

Fig. 7. Effect of Varying Stream Length.

6 Related Work

There has been much research on tree mining [2,10,11,14] to discover the frequent substructures from semistructured data. [2,14] develop the rightmost expansion method to enumerate the subtrees that do not contain wildcards. All these techniques are not appropriate for mining XML query patterns since these patterns contains special characters such as wildcard "*" and relative path "//".

[13] develop an efficient algorithm called FastXMiner to discover frequent XML query patterns. FastXMiner uses a *GQPT*-guided method to enumerate the rooted subtrees. Effective optimization techniques allow FastXMiner to require only a small number of expensive tree containment tests.

At the same time, research in *data stream* has been gaining momentum [9,4,6, 7, etc.]. The focus ranges from clustering, frequency estimation, norm estimation, synopsis structures and order statistics to signal reconstruction. The data items in the stream are typically of simple data types, such as numbers, symbols or sets of such data types. For complex data types like tree-structured data, the related work is rare.

[1] proposes an online mining algorithm called StreamT to discover frequent tree patterns from semi-structured data streams. However, the basic mining model is based on the online association rule mining method [8]. The latter involves two database scans. Hence, StreamT is not a true one-pass algorithm.

7 Conclusion and Future Work

In this paper, we have presented a frequent query pattern mining algorithm called *AppXQSMiner* over the XML query stream. Experiment results show that the proposed approach is both efficient and scalable. To the best of our knowledge, this is the first work in mining tree patterns over query streams.

Future work includes monitoring the frequent query patterns for a XML search engine or query system for emerging patterns, and giving more weight to the newly emerged patterns. Such considerations are important for caching currently hot queries.

References

1. T. Asai, H. Arimura, et.al. Online Algorithms for Mining Semi-structured Data Stream. IEEE ICDM, pp: 27–34, 2002.
2. T. Asai, K. Abe, S. Kawasoe, et. al. Efficient Substructure Discovery from Large Semi-structured Data. 2nd SIAM Int. Conference on Data Mining, 2002.
3. M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. 29th Int. Colloquium on Automata, Languages and Programming, 2002.
4. M. Charikar, S. Chaudhuri, R. Motwani, V. R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. ACM PODS, pp: 268–279, 2000.
5. L. Chen, E. A. Rundensteiner, S. Wang. XCache-A Semantic Caching System for XML Queries. ACM SIGMOD, pp:618, 2002.
6. P. B. Gibbons, Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. ACM SIGMOD, pp:331–342, 1998.
7. S. Guha, A. Meyerson, N. Mishra, R. Motwani, L. O'Callaghan. Clustering Data Streams: Theory and Practice. IEEE Transactions on Knowledge and Data Engineering, pp:515–528, 2003.
8. C. Hidber. Online Association Rule Mining. ACM SIGMOD, pp:145–156,1999.
9. G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. VLDB, pp:346–357, 2002.
10. A. Termier, M. C. Rousset, M. Sebag. TreeFinder: a First Step towards XML Data Mining. IEEE ICDM, 2002.
11. K. Wang, H. Liu, Discovering Structural Association of Semistructured data, IEEE TKDE, 12(3):353–371, 2000.
12. L. H. Yang, M. L. Lee, W. Hsu. Mining Frequent Query Patterns in XML. DASFAA, pp:355–362, 2003.
13. L.H. Yang, M.L. Lee, W. Hsu. Efficient Mining of Frequent Query Patterns for Caching. VLDB, 2003.
14. M. Zaki. Efficiently Mining Frequent Trees in a Forest. ACM SIGKDD, 2002.