Mining Frequent Query Patterns from XML Queries

Liang Huai Yang Mong Li Lee Wynne Hsu Sumit Acharya
School of Computing, National University of Singapore
{yanglh, leeml, whsu}@comp.nus.edu.sg, sumita@iiitb.ac.in

Abstract

As XML prevails over the Internet, the efficient retrieval of XML data becomes important. Research to improve query response times has been largely concentrated on indexing XML documents and processing regular path expressions. Another approach is to discover frequent query patterns since the answers to these queries can be stored and indexed. Mining frequent query patterns requires more than simple tree matching since the XML queries involves special characters such as "" or "//". In addition, the matching process can be expensive since the search space is exponential to the size of XML schema. In this paper, we present two mining algorithms, XQPMiner and XQPMinerTID, to discover frequent query pattern trees from a large collection of XML queries efficiently. Both algorithms exploit schema information to guide the enumeration of candidate subtrees, thus eliminating unnecessary node expansions. Experiments results show that the proposed methods are efficient and have good scalability.*

1. Introduction

With the rapid increase in XML applications such as e-business transactions, XML middleware systems, efficient delivery of XML data has become an important issue. Regular path expression (RPE) is a common feature of XML query languages[3,4]. Processing such RPE can be expensive since it involves navigation through the hierarchical structure of XML, which can be deeply nested. Much research efforts have been focused on the storage and indexing of XML documents, and the efficient evaluation of regular path expressions [8, 9,10].

Another approach to improve query response time is to discover frequent query patterns since the answers to these queries can be stored and indexed. Given an XML data source and the history of XML queries $\{q_1,\ldots,q_N\}$ issued against it, we can transform them into a corresponding history of query pattern trees $\mathbf{D} = \{QPT_1,\ldots,QPT_N\}$. This gives us a database of query pattern trees. Each transaction is then a query pattern tree QPT_i while an itemset is a rooted subtree of QPT_i. Mining frequent query patterns is equivalent to finding the rooted subtrees that occur frequently over the set of pattern trees **D**. This requires more than simple tree matching since the XML queries involves special characters such as "*" or "//". In addition, the matching process can be expensive since the search space is exponential to the size of XML schema. Since the candidate patterns in this mining problem are rooted subtrees, an efficient enumeration technique is critical to reduce unnecessary node expansion and comparisons.

In this paper, we describe two efficient mining algorithms, XQPMiner and XQPMinerTID, to discover the frequent query pattern trees from a large collection of XML queries. Both algorithms exploit schema information to guide the enumeration of candidate subtrees, thus eliminating unnecessary node expansions. To speed up the matching process required in determining the frequency counts of the enumerated subtrees, we develop an algorithm to quickly determine if a rooted subtree is contained in the query pattern trees. Experiments results on real datasets show that the proposed methods are efficient and have good scalability.

The rest of the paper is organized as follows. Section 2 first defines some basic concepts. Section 3 describes the XQPMiner algorithm. Section 4 presents the optimizations for XQPMiner. Section 5 gives the results of the experiments. We discuss related work in Section 6, before concluding in Section 7.

2. Preliminaries

We first define the concepts of query pattern trees and rooted subtrees that form the basis of the algorithms XQPMiner and XQPMinerTID. We also illustrate how an XML query can be transformed into a query pattern tree. In order to determine the frequent query patterns, we need to compute the frequency counts of rooted subtrees efficiently. This entails a tree pattern matching technique that takes into consideration the wildcards and relative paths that may occur in the query patterns.

2.1 Query Pattern Tree

Figure 1 shows an example Book DTD and the corresponding XML tree. The following query Q_1,written in XQuery syntax [4], retrieves the title, author and price of books written by "Buneman".

```
Q1: for $b in document(book.xml) /book
    where some $a in $b/author
       satisfies $a/lastname/data()="Buneman"
    return <result>
              <book>{$b/title, $b/author, $b/price}<book>
           </result>
```

We can extract the following information from Q_1

Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)
0-7695-1895/03 $17.00 © 2003 IEEE

Q_1 {*resultPattern* = {/book/author, /book/title, /book/price}, *predicates* = {/book/author/lastname/data() = "Buneman"}, *documents* = {book.xml}}

where *resultPattern* is the schema pattern of the result, *predicates* are the filtering conditions used in the query, and *documents* are the XML data files involved.

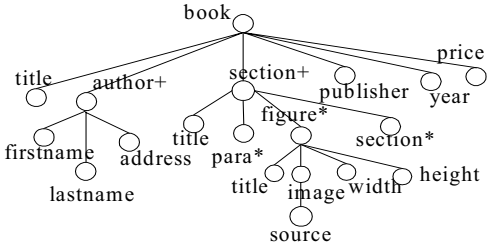

Figure 1. Book DTD Tree.

Next, we extract the paths from the set *predicates* by ignoring the selection conditions. For example, we can extract the path "/book/author/lastname/" from the predicate /book/author/lastname/data() = "Buneman" in Q_1. We combine these extracted path expressions with the paths in the set *resultPattern* to generate the query pattern tree. Figure 2(a) shows the query pattern tree obtained for Q_1. A query pattern tree may not only consist of element tag names, but also wildcard "*" and relative path "//". Wildcards indicate the ANY label in DTD, while relative paths indicate zero or more labels (descendant-or-self). Formally, a query pattern tree can be defined as follows.

Definition 1 (Query Pattern Tree): A *query pattern tree* is a rooted tree *QPT* = <V, E>, where V is the vertex set, and E is the edge set. The root is denoted as *Root*(QPT). We denote each edge e by (v_1, v_2) where node v_1 is the parent of node v_2. Each vertex v has a label with its value in {"*", "//", tagSet}, where tagSet is the set of all element and attribute names in the underlying DTD. We denote the label of a vertex v as *v.label*.

(a) Query Pattern Tree for Q_1 (b) A Rooted Subtree

Figure 2. Query Pattern Tree for Q_1.

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

Figure 2(b) shows a rooted subtree of the query pattern tree in Figure 2(a).

2.2 Frequent Query Pattern Trees

Having transformed a set of queries into query pattern trees, we obtain a database of query pattern trees D = {QPT_1,….,QPT_N}. The query pattern mining problem is to find all the frequent rooted subtrees that occur in D with some minimum support level. The total occurrence of a rooted subtree RST in D is denoted as *freq(*RST*),* and its support is given by *supp(*RST*)* = *freq*(RST)/|D|. For some positive number σ, we say that an RST is σ-frequent in D if *supp*(RST) ≥ σ. Figure 3 shows a database of three query pattern trees and a frequent root subtree. RST occurs in QPT_1 and QPT_2. Hence, its frequency is *freq*(RST) = 2, with a support of *supp*(RST) = 2/3.

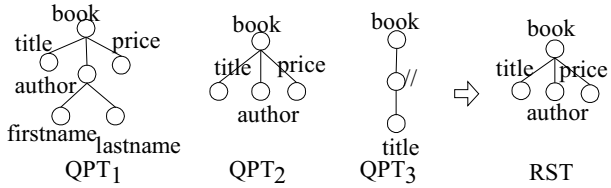

Figure 3. Database of Query Pattern Trees and a Frequent Rooted Subtree.

2.3 Tree Pattern Matching

In general, a tree T = <V, E> *matches* another tree T' = <V', E'> if there exists a mapping φ such that

1. *Root*(T') = φ (*Root*(T)) and ∀v ∈ V, ∃v'∈ V's.t. v' = φ (v) where v.label = v'.label
2. φ preserves the parent-child relation: if (v_1,v_2) ∈ E, then (φ(v_1), φ(v_2)) ∈ E'

then we say that T is a subtree of T' or T is contained in T'.

This naïve definition is not applicable to our tree matching problem due to the presence of wildcards and relative paths in the query pattern trees. Figure 4 shows two query trees T_1 and T_2. Intuitively, the path "book/section/figure/title" in T_2 matches the path "book//title" in T_1 since "//" in "book//title" indicates zero or more labels between the nodes book and title. Since the wildcard "*" can be substituted by ANY single node, then "book/section/figure/image" matches "book/section/*/image". We say that T_2 matches T_1. In other words, T_2 is contained in T_1, written as $T_2 \subseteq T_1$.

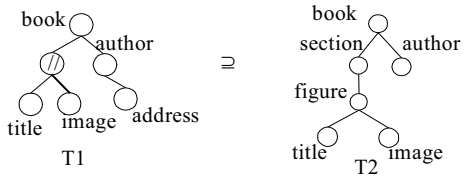

Figure 4. Example of Pattern Tree Containment.

While one could try to expand the non-deterministic paths such as "book//title" to become a deterministic path "book/section/figure/title", this is only feasible when the XML DTD (or schema) is a directed acyclic graph. The expansion method will fail if the DTD contains cycles. Nonetheless, expanding "//" remains crucial. This is because without the context information, one cannot tell whether a path is contained in "//" or not.

Figure 5 shows a query pattern containing two relative path expressions and three rooted subtrees. RST_1 and

RST_2 are contained in QPT since the paths "/book/section/figure/title" and "/book/section/figure/image" are contained in "/book//title" and "/book//image" respectively. The matching paths have common leaf nodes as their context. But it is not clear if RST_3 is contained in QPT since we cannot merge the two child nodes "//" of "book" because nodes "title" and "image" may not share the same parent node. Hence, the expansion of "//" is necessary.

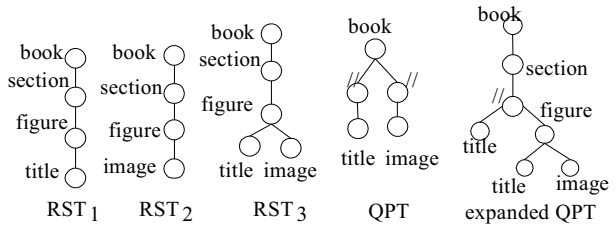

Figure 5. Matching Root Subtrees with a Complex Query Pattern.

We expand the relative paths "//" in a query pattern tree as follows. Suppose the node "//" has a child *n*. If no cycles exist in the underlying XML schema, then the expansion is straightforward. However, if a cycle exists and one of the expanded paths is "root/…/p/n" where the parent of n (node p) has a child that points back to p's ancestor, then we will need to introduce a node "//" between p and n.

Consider the XML schema in Figure 1 and the query pattern tree QPT in Figure 5. As the schema contains cycles, the path "book//title" is expanded into an infinite number of paths including book/title, book/section/title, or book/section*/title, etc. The path book//image can be expanded similarly. We can concisely represent the set of possible expansions by the *expandedQPT* in Figure 5. In this way, we augment the initial query pattern tree with context information, and remove cycles in query pattern trees. It is clear that RST_3 is contained in QPT.

We also observe that the labels in two pattern trees satisfy the partial order relationship ≤. Given a label *x*, we have x≤x, or two nodes with the same label matches. Similarly, we have * ≤ * and //≤//. In order to match nodes involving wildcards and relative paths, we have x≤*≤//, that is, a node with label x matches a wildcard, which in turn matches a node with label //.

Definition 4 (Query Pattern Tree Matching): Given a query pattern tree QPT and a rooted subtree RST, we say that RST is contained in a QPT if the following hold:

1. The root nodes in RST and QPT have the same label.
2. If a node $w \in$ RST is matched with node $v \in$ QPT, then it satisfies (a) *w*.label ≤ *v*.label, and (b) each subtree of *w* is contained in some subtree of QPT.

3. Discovering Frequent Rooted Subtrees

In the context of mining frequent query pattern trees, the database of transactions *D* is a set of pattern trees. Each transaction $t \in D$ is a labeled directed pattern tree extracted from an XML query. Given a minimum support *minSupp*, we would like to find the frequently occuring rooted subtrees in at least *minSup**|D| transactions. In this section, we describe the algorithm XQPMiner for discovering frequent query patterns.

The main framework of XQPMiner is shown in Figure 6. XQPMiner employs the Apriori [2] style to generate frequent rooted subtrees. The notation RST^{k+1} denotes a k+1-edge rooted subtree; F_{k+1} is a set of frequent k+1-edge rooted subtrees; and C_{k+1} is a set of k+1-edge candidate RSTs. The edges correspond to items in traditional frequent itemset discovery. XQPMiner increases the size of frequent RSTs by adding an edge one at a time.

XQPMiner initially enumerates all the frequent 1-edge RSTs by scanning *D* once. The subsequent pass k consists of two phases. In the first phase, the algorithm **RST-Gen** first generates the candidate set C_{k+1} by using the previously found frequent set F_k and pruning those unqualified candidates. Then it counts the frequency for each of these candidates, and removes those RSTs that do not satisfy the minimal support requirement. In the second phase, the algorithm **Contains** determines if RST_{k+1} is contained in the pattern tree *t*.

```
Algorithm XQPMiner (D, minSupp)
Input: D—pattern tree transaction database
       minSupp—the minimum support
Output: Set of all frequent RST sets
1. F1={all frequent 1-edge rooted subtrees in D};
2. for (k=1; Fk≠ϕ;k++) do
       /*generate frequent rooted subtrees*/
3.     Ck+1=RST-Gen (Fk);
4.     for each transaction t∈D do
5.       for each candidate RSTk+1∈Ck+1 do
6.         if Contains (t,RSTk+1) then/* prune */
7.           RSTk+1.count++;
8.     Fk+1={RSTk+1∈Ck+1|RSTk+1.count ≥minSupp*|D|};
9. return {Fi|i=1,…,k-1};
```

Figure 6. Algorithm to Find Frequent RSTs.

3.1 Generation of Candidate RSTs

This step enumerates all the frequent rooted subtrees in *D*. We examine two important issues, namely, how to generate candidate RSTs without repetition, and how to prune the candidates RST early and quickly.

The first issue is also known as the RST enumeration problem. In order to reduce the number of candidate RSTs generated, we propose to use a *schema-guided right most expansion* enumeration method. We construct a global query pattern tree G-QPT by merging the query pattern trees in the database. Figure 7 shows the global query pattern obtained from QPT_1, QPT_2, and QPT_3 in Figure 3.

The schema-guided enumeration method works as follows. Starting with all the possible 1-edge RSTs, we use the G-QPT to systematically guide the generation of 2-edge RSTs level-wise, from which 3-edge RSTs are obtained, and so on. Figure 8(a) shows a 2-edge RST *R* and the set of corresponding 3-edge RSTs obtained based on the G-QPT in Figure 7.

The method in [13] do not use schema information to guide the generation of candidate subtrees. As a result, a large number of candidate trees will be produced, including unnecessary and invalid trees. Figure 8(b) shows all the possible 3-edge RSTs that shares the same prefix as *R* produced. Note the vast difference in the number of RSTs produced when schema information is not utilized.

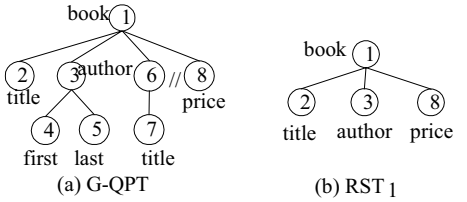

Figure 7. Global Query Pattern Tree for the Query Patterns in Figure 3.

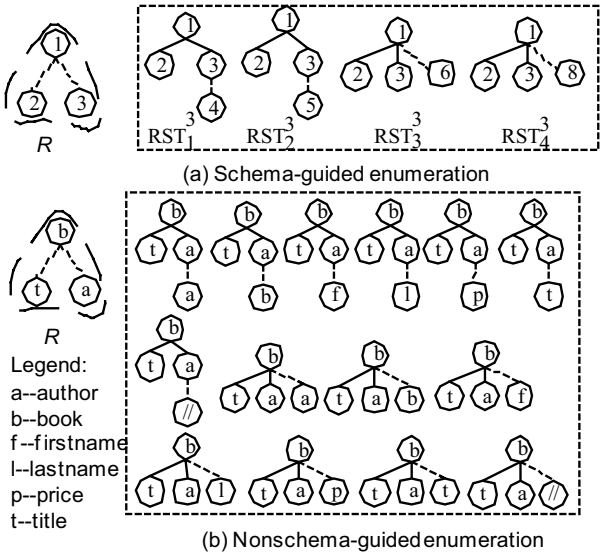

Figure 8. Schema versus Non-schema Guided Enumeration of RSTs.

The second issue is to prune the candidates RST early. A key idea used in the efficient mining of association rules is the Apriori property [2]: If one subset of an itemset is not frequent, then the itemset itself cannot be frequent. This allows one to use the frequent itemsets of size k-1 as filters for candidate itemsets of size k. This property is also true for frequent query pattern mining. If a k-edge RST is frequent, then all its (k-1)-edge RSTs must be frequent. We use this property to prune candidate RSTs.

The candidate generation algorithm **RST-Gen** is shown in Figure 9. Its input is a set of frequent k-edge RST F_k from which a set of candidate (k+1)-edge rooted subtrees C_{k+1} is generated. For each frequent k-edge $RST_i^k \in F_k$, all the possible (k+1)-edge RSTs are enumerated based on the schema (Line 3). Lines 4-7 prune infrequent rooted subtrees by checking each k-edge rooted subtree in the (k+1)-edge RST^{k+1}. If any of the k-edge RST does not exist in the set of frequent k-edge RSTs, then RST^{k+1} is not frequent and will not be added to the candidate set C_{k+1}. Finally, Line 8 returns the (k+1)-edge candidate set C_{k+1}, where the k+1-edge RSTs will be subsequently matched against the query pattern tree database to count their frequency.

Algorithm RST-Gen (F_k)
Input: F_k – Set of frequent k-edge RSTs
Output: Candidate set C_{k+1}
1. $C_{k+1}=\phi$;
2. for each $RST_i^k \in F_k$ do
3. S = enumerate (RST_i^k);
4. for each k+1-edge $RST^{k+1} \in$ S do
5. if $\exists$ leaf node $l_0 \in$ {n | n is a leaf node in RST^{k+1}} and RST^k is a k-edge RSTs obtained by removing l_0 from RST^{k+1} s.t. $RST^k \notin F_k$
6. then S = S - RST^k;
7. $C_{k+1} = C_{k+1} \cup S$;
8. return C_{k+1};

Figure 9. Algorithm to Generate Candidate RSTs.

3.2 Containment of RST in a Pattern Tree

After generating the set of candidate rooted subtrees, the next step is to count their support in the database of QPTs. Traditionally, it is relatively straightforward to test if a candidate itemset is contained in a transaction, since an itemset is a set. Here, a candidate is a rooted subtree. This requires tree matching which is expensive. Further, the matching process is complicated by the presence of "*" and "//" in XML queries.

Figure 10 shows the Contains algorithm that we have designed to determine whether a rooted subtree is contained in a query pattern tree. The input trees are compared recursively from the root to the leaf nodes. The underlying matching criterion is based on the partial order definition given in Section 2. That is, given two nodes $w \in$ RST and $v \in$ QPT, if they satisfy the partial order *w*.label $\leq$ *v*.label and each subtree of *w* is contained in some subtree of QPT, we say the node *w* is matched with the node *v*. We will now elaborate on all the possible matching scenarios:

Case 1: *w* is a leaf node.
(a) We check whether *w* has the more specific label, or its label is not contained in *v.label*. If *v*.label is not "//", then nodes *w* and *v* can be matched directly using

w.label ≤ *v*.label. The following sub-cases requires special attention:

i. *w*.label = "//" or "*". Then the RST is in a transitional state, and we need to examine edges further down the tree before deciding if the RST is actually contained in the QPT. For the moment, the algorithm will return the result of the comparison *w*.label ≤ *v*.label.

ii. If *w*.label appears in the set of labels of node *v*'s ancestors (that is, recursion exists in the schema), then the containment is also determined by *w*.label ≤ *v*.label. This decision is related to Case 3, which we will explain and illustrate with an example later.

(b) If *v*.label = "//", then *w*.label ≤ *v*.label should hold according to the partial order relationship. However, without looking into the context information of the two nodes, we cannot claim that *w* is contained in *v*.

Consider "book//[*v*]price" and "book/section[*w*]" where the nodes enclosed in square brackets are currently being compared. We cannot conclude that *w* is contained in *v* without looking ahead at node *v*'s children. If any of *v*'s child node *n* satisfies the partial order *w*.label ≤ *n*.label, then *w* is contained in *v*. If we have "book/section/title[*w*]" and "book/section//[*v*]title", then *w* is contained in *v*.

Case 2: *w* is not a leaf node, and *v* is a leaf node.
Since *w* is more specific than *v,* it is not possible for *w* to be contained in *v*. An example of this case is *v* = "section" and *w* = "section/title".

Case 3: Both *w* and *v* are not leaf nodes.
If *w*.label ≤ *v*.label does not hold, then *w* is not contained in *v*. Otherwise, if *w*.label ≤ *v*.label holds, then Line 10 tries to compute whether all the subtrees of *w* is contained in those of *v*. If yes, then the algorithm will return *true*. Otherwise, if *v*.label = "//", then by treating it as a zero length path, Lines 11-12 will check whether *w* is contained in one of *v*'s children. If it is not, then Lines 13-14 will consider *v*.label="//" as having multiple nodes and test if the subtrees of *w* are contained in *v*.

We will now illustrate the algorithm with an example. Suppose we want to match "book/section/section[*w*]/ section[*w*']" with "book/section//[*v*]title[*v*']". Both *w* and *v* are not leaf nodes, and we have *w.label* = "section" ≤ *v.label*="//". The algorithm will first determine if the subtrees of *w* are contained in some subtrees of *v*, that is, it will try to match *w*' with *v*' (Case 3). When this fails, the algorithm will examine if any children of *v* contains *w*, that is, treat "//" as 0 length. Otherwise, the algorithm tries the multi-label choice for "//". Lines 13-14 accomplishes this by considering *w* as a node that is contained in *v* and decides if *w*' is contained in *v*. Since *w*' is a leaf node, and *w*'.label appears in the label set of *v*'s ancestors (Case 1), the algorithm returns *true*.

4. Optimizations for XQPMiner

By analyzing the mining framework in Figure 6, optimization techniques can be applied to the candidate generation phase (Line 3) and the frequency counting phase (Lines 4-7). We employ a prefix tree to index the frequent RSTs to facilitate pruning during the generation of candidate RSTs. We also use transaction IDs (TIDs) to reduce the number of tree matching needed during the counting step.

Algorithm Contains (QPT, RST)
Input: QPT— Query pattern tree
RST— Candidate RST
Output: if RST⊆ QPT return *true*; else return *false*
return *SubtreeMatching* (QPT.root, RST.root);

function: *SubtreeMatching* (*v, w*)
Input: *v,w* are nodes in tree pattern T_1, T_2 respectively;
Output: if *w* ⊆ *v* return *true*; else return *false*.
/* Case 1: *w* is a leaf node */
1. if (IsLeaf(*w*))
2. if (*v*.label="//" and (*w*.label∉ {n.label | n ∈ ancestors(*v*)}) and (*w*.label≠"//" or "*") then //will be pruned later
3. if ∃n∈ child(*v*),s.t. *w*.label ≤n.label then
4. return true;
5. else return false;
6. else return (*w*.label≤*v*.label);
/* Case 2: *w* is not a leaf node, and *v* is a leaf node */
7. if (IsLeaf (*v*)) or (¬ (*w*.label ≤ *v*.label)) then
/* v is a leaf while w is not or (¬(*w*.label≤*v*.label) */
8. return false;
/* Case 3: Both *w* and *v* are not leaf nodes, and *w*.label≤*v*.label holds */
9. else
10. result = $\bigwedge_{w' \in child(w)} \left(\bigvee_{v' \in child(v)} SubtreeMatching(v', w') \right)$;
11. if (result=false) and (*v*.label='//') then
12. result= $\bigvee_{v' \in child(v)} SubtreeMatching(v', w)$;
13. if (result=false) and (*v*.label='//') then
14. result= $\bigwedge_{w' \in child(w)} SubtreeMatching(v, w')$;
15. return result;

Figure 10. Pattern Tree Containment Algorithm.

4.1 Encoding Query Pattern Trees

The nodes in a global query pattern tree can be numbered using a pre-order traversal. The nodes in the source query pattern trees will be numbered according to this encoding scheme. Figure 7 shows an example of a numbered global query pattern tree G-QPT and a query pattern tree QPT_1. This numbering scheme allows us to simplify the XML representation of the query pattern trees in the database. For example, QPT_1 in Figure 7 can be simplified to <1><2></2><3></3><8></8></1>

By omitting the brackets and replacing each end tag with –1, we can shorten the representation to "1,2,-1,3-1,8,-1". Note the last end tag need not be included. This string encoding scheme is often used to encode a tree to facilitate comparison [7,13]. This also reduces the memory requirement during the mining process since the candidate rooted subtrees are now encoded as strings. Tree operations now become string operations.

4.2 Indexing Frequent RSTs

When generating a (k+1)-edge rooted subtree RST^{k+1} from some k-edge RST, we need to determine if all the possible k-edge rooted subtrees in RST^{k+1} are frequent. To facilitate this checking, we employ a prefix tree to index the previously generated frequent RSTs. The prefix tree behaves like a hash tree. That is, the RSTs stored in the tree are indexed using the string encoding (with the -1's removed) described in the previous section. The lookup time of the prefix tree is about O(L), where L is the length of the string encoding.

4.3 Using Transaction IDs

Since tree matchings are expensive, it is critical to reduce them in the mining process. Transaction IDs are often used in [2] to expedite the mining process. In this section, we examine how TIDs can help reduce the number of tree matching.

Given a RST^k, we know that the expansion along the right most branch will generate all the (k+1)-edge RSTs that share the same prefix as RST^k. The level-wise approach that we have adopted would enumerate all the k-edge RSTs before going on to generate the (k+1)-edge RSTs. We observe that for each RST^k, the RST^{k+1} is enumerated by expanding the nodes along the right most branch of RST^k. We divide the enumeration of RST^k into two sets: (1) those generated by expanding the right most leaf node denoted as G_{leaf}; and (2) those generated by expanding the nodes along the right most branch except the leaf node, which is denoted as $G_{internal}$. For $RST^{k+1} \in G_{internal}$, it has at least two distinct embedded k-edge RSTs:

1. RST^k without expansion (Figure 11(a)),
2. RST^k with the right most leaf node removed from RST^{k+1} (Figure 11(c)).

These embedded k-edge RSTs do not require tree matching as the information is already stored in the prefix tree. Assuming the transactions (QPTs) which contain RST^k and RST_1^k (Figure 11) are denoted as RST^k.TIDList and RST_1^k.TIDList respectively. Clearly, RST^{k+1}.TIDList = RST^k.TIDList $\cap$ RST_1^k.TIDList. Consequently, it is not necessary to matched RST^{k+1} against the query pattern trees in the database. This implies that all the RSTs in $G_{internal}$ need not be matched.

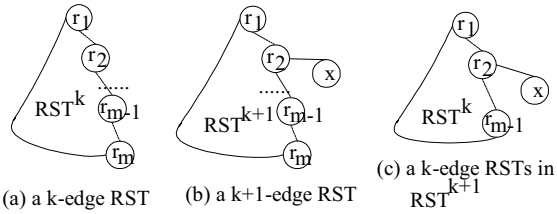

Figure 11. Non-rightmost Leaf Node Expansion.

Algorithm RST-GenTID (F_k, support, F_{k+1},C_{k+1})
Input: F_k -frequent k-edge RSTs,
support = *minSupp**|D|
F_{k+1}, C_{k+1} /*also used for return result */
Output: Candidate set C_{k+1} and part of F_{k+1}
1. C_{k+1}=ϕ; F_{k+1}=ϕ;
2. for each $RST_i^k \in F_k$ do
3. G_{leaf} = {RSTs obtained by expanding right most leaf node of RST_i^k };
4. for each k+1-edge $RST^{k+1} \in G_{leaf}$ do/* prune */
5. if exists a leaf node $l_0 \in$ {n | n is a leaf node RST^{k+1}} and RST^k=k-edge RSTs obtained by removing l_0 from RST^{k+1} s.t. $RST^k \notin F_k$ then
6. S=S - RST^k;
7. $C_{k+1}= C_{k+1} \cup G_{leaf}$;
8. $G_{internal}$ = {RSTs obtained by expanding nodes on right most branch of RST_i^k except the leaf node};
9. for each k+1-edge $RST^{k+1} \in G_{internal}$ do
10. find one embedded k-edge RSTs RST_p^k of RST^{k+1} s.t. $RST_p^k \neq RST_i^k$ holds;
11. RST^{k+1}.TIDList= RST_p^k.TIDList$\cap$ RST_i^k.TIDList;
12. RST^{k+1}.count=|*rst*.tidList|;
13. if (RST^{k+1}.count ≥ support)
14. $F_{k+1} \leftarrow RST^{k+1}$; /* add this RST in F_{k+1} */
15. return;

Figure 12. Algorithm RST-GenTID

With the use of TIDs, the candidate generation algorithm RST-Gen is modified to RST-GenTID (see Figure 12). Lines 3-7 perform the candidate generation. We first obtain the set of G_{leaf}, and prune all invalid candidates. The prefix tree is used to quickly determine if the embedded k-edge RST of the (k+1)-edge RST is frequent. Finally, we add G_{leaf} to the candidate set C_{k+1}, which will be matched against QPTs in the database. Lines 8-14 generate the set of $G_{internal}$ and compute the frequencies of the RSTs by using TIDs without involving expensive tree matchings. If the (k+1)-edge RST^{k+1} satisfies the minimum support, then it is added to the F_{k+1}.

Figure 13 shows the XQPMinerTID algorithm. The difference between XQPMiner and XQPMinerTID lies in the fact that XQPMinerTID obtains a portion of the frequent RSTs without the need to match them to the query pattern trees in the database.

Algorithm XQPMinerTID (D, *minSupp*)
Input: D—pattern tree transaction database
minSupp—the minimum support
Output: Set of all frequent RST sets
1.F_1={all frequent 1-edge rooted subtrees in *D*};
2.support= *minSupp**|D|;
3.for (k=1; $F_k \neq \phi$;k++) do
4. $F_{k+1}=\phi$; $C_{k+1}=\phi$;
/* generate frequent rooted subtrees */
5. **RST-GenTID**(F_k,support,F_{k+1}, C_{k+1});
6. for each transaction $t \in D$ do
7. for each candidate $RST^{k+1} \in C_{k+1}$ do
8. if **Contains**(t,RST^{k+1}) then//prune stage
9. RST^{k+1}.TIDlist$\leftarrow t$.TID;
10. RST^{k+1}.count++;
11. $F_{k+1} \leftarrow \{RST^{k+1} \in C_{k+1} | RST^{k+1}.count \geq support\}$;
12.return $\{F_i | i=1,\ldots,k-1\}$;

Figure 13. Algorithm XQPMinerTID.

5. Performance Study

We implement the algorithms in C++, and carry out experiments to evaluate the performance of our algorithms. All the experiments are performed on a 2.4GHz PC with 1GB RAM, running Windows XP.

We use the DBLP.DTD and Shakespears' Play.DTD as the schemas of XML data sources. A DTD graph is converted into a DTD tree by introducing some "//" and "*" nodes, from which the G-QPT is obtained. To generate the QPTs of XML queries, we first enumerate all the RSTs of the G-QPT. Next, we use the Zipfian distributions to produce the transaction file of QPTs from the RSTs. The Zipfian distribution is used because Web queries and surfing patterns typically conform to Zipf's law.

Each dataset consists of 200,000 QPTs, which follows the Zipfian distribution. Table 1 lists the different characteristics of the two datasets used.

Table 1. Properties of Datasets.

	Datasets	DBLP	Shakespears Play
G-QPT	Num. of nodes	98	67
	Max depth	8	6
	Num. of //	13	0
	Max fanout	12	9
QPT in DB	Ave # of nodes	7.4	7.5
	Max depth	8	6
	Max fanout	12	9

5.1 Effect of minSupp Values on Response Time

This experiment on the DBLP dataset investigates the impact of minSupp value on response time. The results are shown in Figure 14. With the decrease of minSupp value, XQPMinerTID outperforms XQPMiner significantly. XQPMinerTID is about 49 times faster at support 1% and leaps to about 60 times faster at support 0.1% compared to XQPMiner. This is because there are more RSTs to be compared when the support value is lower. XQPMiner matches each RST candidate generated against the QPTs in the database while XQPMinerTID is able to avoid a large number of expensive tree matching by using TIDs.

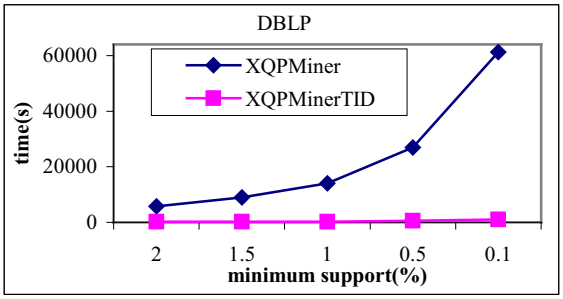

Figure 14. Response Times when Minimum Support Varies (DBLP dataset).

5.2 Effect of Schema Size on Response Time

Next, we compare our schema-guided enumeration approach with non-schema guided approach proposed in [13]. We will call the latter method ZakiTID. This experiment is carried out on the Shakespears' Play datasets. Since ZakiTID does not handle "*" and "//", we generate QPTs without these tags. We vary the number of nodes in the schema, but maintain the same number of frequent RSTs. Figure 15 presents the results. XQPMinerTID outperforms ZakiTID by 5 to 7 times for the schema with 23 nodes. This soars to 26 to 36 times when the size of the schema increases to 67 nodes. This is because without the schema information, Zaki's enumeration method will produce a large number of unnecessary candidates. In contrast, XQPMinerTID is almost unaffected since the response time of XQPMinerTID depends only on the set of frequent query patterns to be mined.

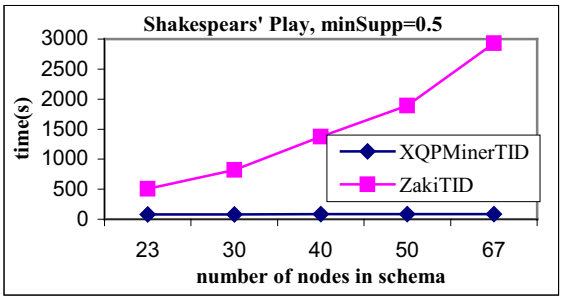

Figure 15. Response Times for Schema-guided and Non-schema Guided RST Enumeration.

5.3 Effect of Number of QPTs on Response Time

In this experiment, we investigate the impact of the number of transactions (or QPTs) in the database on response time. The results for the DBLP dataset are shown in Figure 16. Both algorithms scale linearly with the size of dataset. XQPMinerTID is about 20 to 30 times faster than XQPMiner. The many recursions in the DBLP global query pattern tree, and the large number of nodes in the G-QPT (98 nodes) results in high tree matching cost. In contrast, XQPMinerTID takes much less time for XQPMiner as unnecessary tree matches are avoided by using TIDs.

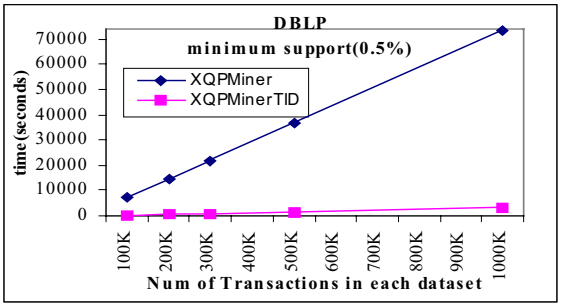

Figure 16. Response Times when the Number of QPTs Increases.

6. Related Work

Finding frequent substructures from graphs gains its focus in DNA/RNA research. [12] devises an algorithm to discover approximate common subtrees in multiple RNA secondary structures in genomics. [5] presents an efficient algorithm for finding frequent substructure from labeled graphs and applies it to the problem of function prediction of chemical compounds. [6] deals with the problem of frequent subgraphs, especially, the issues of graph isomorphism. All these methods are not applicable to discover frequent tree patterns with wildcards and relative paths.

The closest work to ours is [11] which finds the frequent substructures from a collection of semi-structured Web documents. [11] uses a tree matching algorithm to count the support of candidate substructures by introducing the wildcard '?' in the subtree to match any label in the path. The naïve expansion of recursive nodes fails to capture the precise semantic of recursion. Again, this method cannot be used to mine XML query patterns because the query patterns contains the special characters '*' and '//'.

[13] develops a frequent subtree mining algorithm to discover the user navigation patterns in web surfing. The subtree is a generalized one where its interior nodes can shrink. However, the method does not handle '*' and '//' which are peculiar to XML queries. In addition, for each leaf node of the current pattern, all the possible node expansions have to be tested because there is no schema information to guide their enumeration. Hence, the enumeration is inefficient. [1] deals with the same problem by enumerating subtrees in a similar way to [13].

7. Conclusion

In this paper, we have described a schema-guided mining approach to discover frequent rooted subtrees from XML queries. This approach allows us to enumerate only valid candidates RSTs. We have also developed a tree pattern containment algorithm that takes into account the relative path "//" and wildcare "*" when matching RSTs with query pattern trees. Several optimizations have also been proposed, in particular, using TIDs to reduce the number of tree matchings needed. Experiments results reveal that XQPMinerTID outperforms XQPMiner by a factor of 6-60, and has good scalability.

8. References

[1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, S. Arikawa. Efficient substructure discovery from large semistructured data. 2nd SIAM Int'l Conference on Data Mining, 2002.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. VLDB, 1994.

[3] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0. W3C recommendation, 1999.

[4] D. Chamberlin, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. XQuery: A Query Language for XML W3C working draft. World Wide Web Consortium, 2001.

[5] L. Dehaspe, H. Toivonen, R. D. King. Finding Frequent Substructures in Chemical Compounds. ACM SIGKDD, pp: 30-36, 1998.

[6] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based algorithm for mining frequent substructures from graph data. 4th European Conf. on Principles and Practice of Knowledge Discovery in Databases, pp:13-23, 2000.

[7] F. Luccio, A. M. Enriquez, P. O. Rieumont and L. Pagli. "Exact Rooted Subtree Matching in Sublinear Time". See ftp://ftp.di.unipi.it/pub/techreports/TR-01-14.ps.Z

[8] R. Kaushik, P. Bohannon, J. Naughton, H. Korth. Covering Indexes for Branching Path Queries. ACM SIGMOD, 2002.

[9] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. VLDB, 2001.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. VLDB'02, 2002.

[11] K. Wang, H. Liu, Discovering Structural Association of Semistructured data, IEEE Transactions on Knowledge and Data Engineering,12(3), pp:353-371, 2000.

[12] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, C.-Y. Chang, Automated Discovery of Active Motifs in Multiple RNA Secondary Structures. ACM SIGKDD, pp:70-75, 1996.

[13] M. Zaki. Efficiently Mining Frequent Trees in a Forest. ACM SIGKDD, 2002.