Efficient Mining of XML Query Patterns for Caching

Liang Huai Yang Mong Li Lee Wynne Hsu

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

Abstract

As XML becomes ubiquitous, the efficient retrieval of XML data becomes critical. Research to improve query response time has been largely concentrated on indexing paths, and optimizing XML queries. An orthogonal approach is to discover frequent XML query patterns and cache their results to improve the performance of XML management systems. In this paper, we present an efficient algorithm called FastXMiner, to discover frequent XML query patterns. We develop theorems to prove that only a small subset of the generated candidate patterns needs to undergo expensive tree containment tests. In addition, we demonstrate how the frequent query patterns can be used to improve caching performance. Experiments results show that FastXMiner is efficient and scalable, and caching the results of frequent patterns significantly improves the query response time.

1. Introduction

Since its inception in 1998, XML has emerged as a standard for data representation and exchange on the World Wide Web. The rapid growth of XML repositories has provided the impetus to design and develop systems that can store and query XML data efficiently. Given that XML conforms to a labeled tree or graph, the basic features in query languages such as XPath [7] or XQuery [8] are regular path expressions and tree patterns with selection predicates on multiple elements that specify the tree-structured relationships.

For example, for a query *book [title="XML", year="2000"] // author [lastname="Buneman"],* a query engine has to find matches for a *book* element which has the children *title* with content "XML" and *year* with content "2000", and the *book* element also has a descendent author which contains a child *lastname* with content "Buneman". Processing such XML queries can be expensive because it involves navigation through the hierarchical structure of XML, which can be deeply nested. Current research to improve query response times has been focused on indexing paths [9, 13, 16] and optimizing various classes of XML queries [2, 5].

Caching has played a key role in client-server databases, distributed databases and Web-based information systems because network traffic and slow remote servers can lead to long delays in the delivery of answers. Work on semantic/query caching examines how user queries, together with the corresponding answers can be cached for future reuse [6, 11]. The advantage of this is that when a user refines a query by adding or removing one or more query terms, many of the answers would have already been cached and can be delivered to the user right away. This avoids the expensive evaluation of repeated or similar queries.

Traditional caching strategies typically consider the contents in a cache as belonging to a priority queue. The LRU (and its many variations) is a well-established replacement strategy that evicts the least recently accessed objects from the cache when it is full. The recent move towards intelligent web caching tries to adapt to changes in usage patterns by constructing predictive models of user requests by mining web log data [4, 18].

In this paper, we examine how the query performance in XML management systems can be improved by caching XML query results. The results to frequent XML queries are cached in anticipation of future retrievals. This entails the discovery of frequent query patterns [24]. Mining these patterns requires more than simple tree matching as the XML queries contains special characters such as the wildcard "*" and relative path "//". The matching process can be expensive since the search space is exponential to the size of the XML schema.

Motivated by the need to reduce expensive tree matching, we develop theorems to prove that only a small subset of the generated candidate patterns needs to undergo costly tree containment tests. We present an

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

efficient algorithm called FastXMiner, to discover frequent XML query patterns, and demonstrate how these query patterns can be incorporated into a query caching system for XML data. Experiment results show that FastXMiner is efficient and scalable, and that utilizing the frequent query patterns in caching strategy increases the cost-saving ratio and reduces the average query response time.

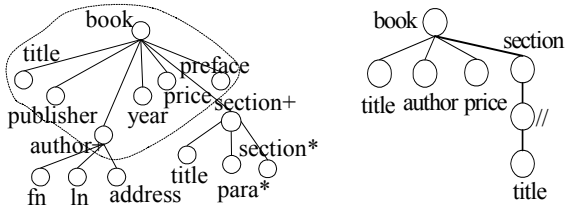
The rest of the paper is organized as follows. Section 2 discusses some concepts used in mining query patterns. Section 3 describes our approach to mine frequent query patterns efficiently. Section 4 shows how the discovered query patterns can be exploited in caching. Section 5 presents the results of our experiments. Section 6 discusses the related work, and we conclude in Section 7.

2. Preliminaries

In this section, we define some of the basic concepts used in query pattern tree mining.

2.1 Query Pattern Tree

XML queries can be modelled as trees. We call them query pattern trees. Consider the example BOOK DTD tree in Figure 1(a). A query to retrieve the title, author and price of books where books/section//title has value "XML Schema" would have the query pattern tree as shown in Figure 1(b). 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) in DTD, while the relative path "//" indicates zero or more labels (descendant-or-self).



(a) Example BOOK DTD (b) A Query Pattern Tree.

Figure 1. A DTD and a XML Query Pattern Tree.

Definition 1 (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 the pattern tree 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.

Definition 2 (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'⊆ V, E' ⊆ E.
We call an RST a *k*-edge rooted subtree if it has *k* edges.

2.2 Tree Inclusion

In order to decide if a RST is included in some QPT, we need to define the semantics of tree inclusion. Several definitions of tree inclusion exist including subtree inclusion [20], tree embedding [25] and tree subsumption [14]. The most relevant definition for this work is the subtree inclusion, which states that *a subtree t' is included in some tree t if and only if there exists a subtree of t that is identical with t'*. However, this definition is too restrictive for XML query pattern trees where handling of wildcards and relative paths are necessary.

Consider the two trees T_1 and T_2 in Figure 2. Let (p,q) denote that a node p in T_1 is mapped to a node q in T_2. Since we are dealing with rooted subtrees, we can carry out a top-down matching. Here, (*book,book*) is mapped first. Next, we check that each subtree of *book* in T_1 matches with some subtree of *book* in T_2. This requires that the subtree rooted at *section* of T_1 (denoted as subtree(*section*)) has to be matched against the subtrees rooted at '//' and *author* of T_2. We need to consider whether '//' indicates zero or many nodes in the path:

Case 1: '//' means zero length.

Then subtree(*section*) must be included in either subtree(*title*) or subtree(*image*) of T_2, which is not the case here.

Case 2: '//' means many nodes.

This implies that *section* has been mapped to some 'unknown' node in T_2. From all the possible subtrees of *section*, only one subtree, i.e., subtree(*figure*), must be included by subtree('//').

It is obvious that subtree(*author*) of T_1 is included in subtree(*author*) of T_2. The inclusion of T1 in T2 is shown in Figure 2 via dashed lines. We conclude that T_2 includes T_1, denoted as $T_1 \subseteq T_2$. Note that if we have applied an exact subtree inclusion definition, then T_1 would not be included in T_2.

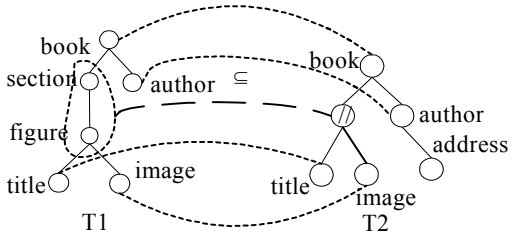


Figure 2. Example of Tree Inclusion.

Before we define our concept of tree inclusion, we first introduce the notion of the *partial order relationship ≤ of labels* in QPTs.

Definition 3 (Partial Ordering of Labels): Given two labels x and x', if x=x', then we say x ≤ x'. For any label x ∈ tagSet, we define x ≤ * ≤ //, that is, a node with label

x matches a wildcard, which in turn matches a node with label //.

Definition 4 (Extended Subtree Inclusion): Let subtree(p) and subtree(q) be two subtrees with root nodes p and q respectively. Let children(v) denotes the set of child nodes of v. Then we can recursively determine if subtree(p) is included in subtree(q), denoted by subtree (p) $\subseteq$ subtree (q), as follows:

p $\leq$ q and satisfies:

(1) both p and q are a leaf nodes; or

(2) p is a leaf node and q = '//', then $\exists$q'$\in$children(q) such that subtree(p)$\subseteq$ subtree(q'); or

(3) both p and q are non-leaf nodes, and one of the following holds:

i. $\forall$p'$\in$children(p), $\exists$q'$\in$children(q) such that subtree(p')$\subseteq$subtree(q'); or

ii. q='//' and $\forall$p'$\in$children(p), we have subtree(p') $\subseteq$ subtree(q); or

iii. q='//' and $\exists$q'$\in$children(q) where subtree(p) $\subseteq$ subtree(q');

In addition, a DTD may contain recursions such as *part has (sub)part(s)*. Consider the test for whether a path "a/b/b" is included in "a/b//c". If we do not know that there exists a path "a/b/b//c", then we cannot conclude that the first path is included in the second path. This is because it is possible for a DTD declaration to include "a/b/d/e/c" or "a/b/b/f" but not "a/b/b//c". In order to handle these situations, we need to take into account the DTD and perform some expansions of the QPTs. Interested readers are referred to [24] for the details.

Clearly, performing such extended subtree inclusion test is expensive. Below, we describe an efficient algorithm for mining query pattern trees that aims at minimizing the number of tree containment tests.

2.3 Query Pattern Tree Mining Problem

Given a set of query pattern trees **D**={QPT_1, …., QPT_N}, mining the frequent query pattern implies discovering the frequent rooted subtrees (RSTs) in the query pattern trees. A rooted subtree RST matches a query pattern tree QPT in D (or RST occurs in D) if there exists a QPT that includes the RST. The total occurrence of an RST in D is denoted by freq(RST), and the support level supp(RST) is given by freq(RST)/|D|, where |D| denotes the number of QPTs in database D. We say that RST is σ-frequent in D if supp(RST) $\geq \sigma$ for some positive number σ.

Frequent Query Pattern Mining Problem: Given a query pattern tree database D={QPT_1,….,QPT_N}, and a positive number $0 < \sigma \leq 1$ called the minimum support, find F, the set of all σ-frequent rooted subtrees, that is, rooted subtrees RST such that supp(RST) $\geq \sigma$.

Consider the query pattern trees and a 3-edge rooted subtree RST in Figure 3. 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 [3]. 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 expensive tree inclusion test.

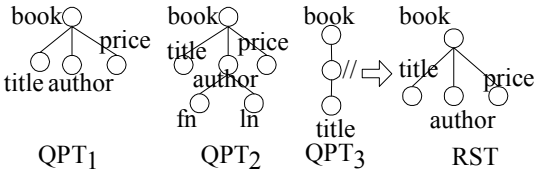


Figure 3. Example of a Frequent Query Pattern Tree.

3. Mining Query Pattern Trees

Mining frequent query patterns is expensive because of the potentially large number of candidate RSTs that can be generated, and the expensive tree containment tests that these candidate RSTs need to undergo. The work in [24] describes an Apriori-based algorithm XQPMiner that exploits the underlying schema to avoid the exhaustive enumeration of candidate RSTs. In this section, we build upon this work, and present a novel strategy to further reduce the costly tree inclusion tests.

We utilize a tree-encoding scheme to partition candidate RSTs into equivalence classes that are further divided into three groups. We prove that only the group that contains *single-branch* candidate RSTs needs to be matched against the XML query patterns in the database. This leads to a large reduction in the number of tree inclusion tests required. Subsequent experimental studies show that this technique dramatically improves the efficiency of the mining process.

Based on the above result, we develop an efficient frequent query pattern mining algorithm called **FastXMiner** (see Figure 4). In the algorithm, the notation RST^k denotes a k-edge rooted subtree; F_k is a set of frequent k-edge rooted subtree; and C_k is a set of k-edge candidate RST. Edges correspond to items in traditional frequent itemset discovery. The size of frequent RSTs is increased by adding one edge at a time. In our implementation, we associate each RST^k with a list of TIDs, denoted as RST^k. tidlist, which indicates this RST^k is included in which QPTs. Similarly, $|RST^k.tidlist|$ is the number of TIDs in RST^k.

FastXMiner initially enumerates all the frequent 1-edge RSTs by scanning the database D once. In the subsequent passes, we generate the frequent (k+1)-edge RSTs from the frequent k-edge RST in two phases. In the first phase, the algorithm **FastRSTGen** (more details on this in Section 3.3) is called to generate the candidate set C_{k+1} by using the previously found frequent set F_k. Any unqualified candidate RSTs is pruned. The frequency for each candidate RST is counted, and those RSTs that do

not satisfy the minimal support criteria are pruned. The candidate set C_{k+1} contains all the RSTs to be matched with the QPTs in the database.

In the second phase, the algorithm **Contains** is called to determine if RST_{k+1} is included in the pattern tree *t*. This test is based on the extended tree inclusion definition. Details of the Contains algorithm are given in [24].

Algorithm FastXMiner (D, *minSupp*)
Input: D—pattern tree transaction database
minSupp—the minimum support
Output: Sets of frequent RSTs
1.F_1={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. FastRSTGen (F_k, support, F_{k+1}, C_{k+1});
6. for each transaction $t \in D$ do
7. for each *single-branch* candidate $RST^{k+1} \in C_{k+1}$ do
8. if **Contains** (*t*, RST^{k+1}) then //prune
9. RST^{k+1}.tidlist$\leftarrow$*t*.TID;
10. $F_{k+1} \leftarrow \{RST^{k+1} \in C_{k+1} \mid |RST^{k+1}.tidlist| \geq support\}$;
11.return $\{F_i \mid i = 1, \ldots, k-1\}$;

Figure 4. Algorithm FastXMiner.

The following subsections will describe the theory behind FastXMiner.

3.1 Candidate Generation

The first step of FastXMiner is to enumerate all the frequent RSTs in *D*. To facilitate this process, we construct a global query pattern tree G-QPT by merging the query pattern trees in the database. Figure 5(a) shows the global query pattern obtained from the query pattern trees in Figure 3.

The nodes in the G-QPT can be numbered using a pre-order traversal. Since each QPT$\in$D is contained in G-QPT, each node in QPT has the same number as the corresponding node in G-QPT (see Figure 5). 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 usage during mining, but also simplifies the representation of the query pattern trees. For example, QPT_1 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. This string-encoding scheme is often used to facilitate tree comparison [25].

Definition 5 (Order of String Encodings): Any two string encodings S_{e1} and S_{e2} can be transformed into the corresponding strings S_1 and S_2 by removing all the –1 from S_{e1} and S_{e2}. We denote the order of $S_{e1}$and S_{e2} by using S_1, S_2, that is, $S_{e1} \leq S_{e2}$ iff $S_1 \leq S_2$.

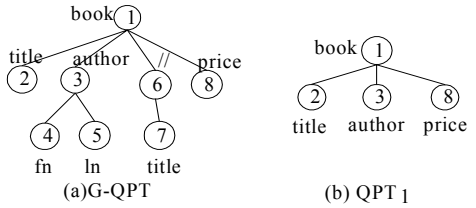


Figure 5. Numbering Scheme for G-QPT and QPT.

Definition 6 (Prefix of a RST): A *prefix* of an RST's string encoding *S* is defined as the list of nodes up to the i^{th} node in *S*, and is denoted as *prefix*(*S,i*). Here, –1 is not considered as a node. To simplify discussion, we will also use *prefix*(*RST, i*) to refer to *prefix*(*S, i*).

After obtaining the global query pattern tree, the RST enumeration problem is now reduced to the problem of enumerating the RSTs in a G-QPT. Starting with all the possible 1-edge RSTs, we use the G-QPT to systematically guide the generation of 2-edge RSTs level-wise by expanding the rightmost branch, from which 3-edge RSTs are obtained, and so on. Figure 6 shows a 2-edge RST *R* and the set of corresponding 3-edge RSTs generated based on the G-QPT in Figure 5.

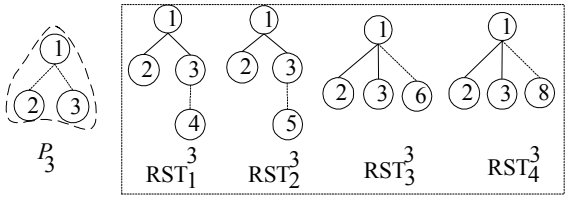


Figure 6. G-QPT-Guided Enumeration.

We introduce the equivalence relation $=_{prefix}$ to partition the RSTs into equivalence classes. Given two k-edge rooted subtrees RST_1^k and RST_2^k, let s_1 and s_2 denote their respective string encodings. Then $RST_1^k =_{prefix} RST_2^k$ if the equation *prefix*(s_1,k) = *prefix*(s_2,k) holds. Based on the numbering scheme and the rightmost expansion method, we derive the following lemma.

Definition 7 (Rightmost Branch Expansion): Given a k-edge rooted subtree RST^k, the (k+1)-edge RST set formed by expanding the rightmost branch of RST^k is denoted as *rmbe*(RST^k).

Lemma 1: *rmbe*(RST^k) is an equivalence class based on the relation $=_{prefix}$, which shares the same prefix P_{k+1}=*prefix*(RST^k, k+1). The equivalence class is denoted as $[P_{k+1}]$. □

Example: Consider the prefix P_3 = "1,2,-1,3". There are 4 rooted subtrees in the G-QPT in Figure 5(a) that share this prefix: RST_1^3, RST_2^3, RST_3^3, RST_4^3 (see Figure 6). These RSTs form the equivalence class (EC)[P_3].

We next investigate the basic properties for the RST enumeration. Lemma 2 states that the string encodings of any $RST_1^k =_{prefix} RST_2^k$ can at most differ by two.

Lemma 2: For any two k-edge rooted subtrees RST_1^k and RST_2^k that belong to the same equivalence class $[P_k]$, if s_1 and s_2 are their respective encodings, then $1 \le diff(s_1, s_2) \le 2$, where diff() is a string comparison function.

Proof: Let P_k denote the k-prefix of this EC, and let the rightmost branch to be $n_1,n_2,..,n_m$. Assume RST_1^k and RST_2^k are expanded at nodes n_i and n_j with nodes x and y respectively, $i,j \in \{1,\ldots,m\} \wedge i \neq j$. Since the expansion is on the rightmost path, the string encodings of RST_1^k and RST_2^k are $P_k\{,-1\}^{m-i},x\{,-1\}^i$ and $P_k\{,-1\}^{m-j},y\{,-1\}^j$. It is easy to see that if $n_i=n_j$, then $diff(s_1,s_2)=1$;else, $diff(s_1,s_2)=2$. □

The Apriori property [3] states that two frequent k-itemset with the same (k-1)-itemset prefix can be joined to produce a k-itemset candidate. This property also holds here.

Definition 8 (Join of two RSTs): Given two k-edge RSTs RST_1^k and RST_2^k which share the same prefix, the join result of RST_1^k and RST_2^k is denoted as RST_{12}^{k+1}, that is, $RST_{12}^{k+1} = RST_1^k \bowtie RST_2^k$, where RST_{12}^{k+1} is a k+1-edge RST candidate.

Theorem 1 (Join Result Encoding): Suppose RST_1^k and RST_2^k satisfy $RST_1^k =_{prefix} RST_2^k$. If P_k is their k-prefix, then their string encodings are $s_1= P_k\{,-1\}^{m-i},x\{,-1\}^i$ and $s_2= P_k\{,-1\}^{m-j}$, $y\{,-1\}^j$ respectively. The string encoding of RST_{12}^{k+1} must be one of the following form:

Case 1: $i = j$. If x < y, then we have $P_k\{,-1\}^{m-i},x,-1,y\{,-1\}^i$. Otherwise we have $P_k\{,-1\}^{m-i},y,-1,x\{,-1\}^i$. This preserves the numbering order.

Case 2: $i > j$. $P_k\{,-1\}^{m-i},x,-1\{,-1\}^{i-j},y\{,-1\}^j$.

Case 3: $j > i$. The string encoding is symmetric to case i>j, i.e., $P_k\{,-1\}^{m-j},y,-1\{,-1\}^{j-i},x\{,-1\}^i$.

Proof: Since RST_1^k and RST_2^k are two k-edge RSTs in the same equivalence class, Lemma 2 holds. If diff $(s_1,s_2) = 1$, i.e., i = j, then this indicates that RST_1^k and RST_2^k have been expanded at the same node n_i. The join of RST_1^k and RST_2^k is tantamount to inserting a rightmost node of RST_1^k into the node n_i of RST_2^k. To preserve the numbering order, we insert the rightmost node of an RST with smaller number into the node n_i of the other RST. This proves Case 1.

If diff $(s_1, s_2) = 2$, then this corresponds to Cases 2 and 3. Here, we provide the proof for Case 2. The proof for Case 3 is similar. From the string encodings of RST_1^k and RST_2^k, we know that RST_1^k and RST_2^k are derived by expanding the nodes n_i and n_j respectively in P_k, where n_i and n_j are two nodes in the rightmost path $n_1,n_2,\ldots,n_m$ of P_k. From the numbering scheme, the node x which is added as a child of n_i has a larger number than the node y which is added as a child to n_j (Figure 7). The effect of joining RST_1^k and RST_2^k is equivalent to adding node y of RST_2^k to node n_j of RST_1^k. □

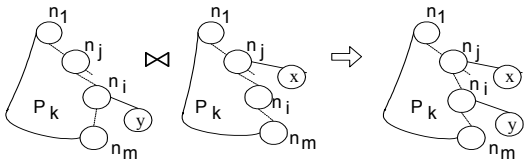


Figure 7. Join of RST_1^k and RST_2^k.

Theorem 1 allows us to simplify the join of two RSTs. We only need to compare the two RST's string encodings and find the first position where they differ. The string encoding of a new candidate RST can be obtained by inserting the node with the (smaller) number n plus –1 ({ ,n,-1}) into the position of the other string encoding. This method avoids the expensive tree comparisons needed in the traditional candidate generation.

Example: Consider the joins of RST_1^3 and RST_2^3, RST_1^3 and RST_4^3 in Figure 8. Their respective string encodings are given by s_1=1,2,-1,3,4,-1.-1, s_2=1,2,-1,3,5,-1,-1 and s_4=1,2,-1,3,-1,8,-1. These encodings have in common the prefix P_3=1,2,-1,3, and can be rewritten as $s_1=P_3$,4,-1.-1, $s_2=P_3$,5,-1,-1, $s_3=P_3$,-1,8,-1. We have $diff(s_1,s_2) = 1$. Since the smaller differing node 4 is found in s_1, we insert {,4,-1} into s_2 before node 5, and obtain the string encoding s_{12} of $RST_1^3 \bowtie RST_2^3$: P_3,4-1,5,-1,-1. On the other hand, we have $diff(s_1,s_4)=2$. We find the first differing node($\neq$–1) 4 in s_1. By inserting {,4,-1} before the respective position of s4, we get the string encoding s_{14} of $RST_1^3 \bowtie RST_4^3$: P_3,4,-1,-1,8,-1.

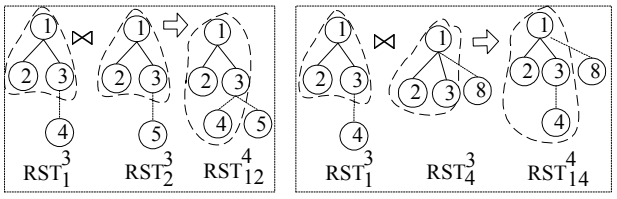


Figure 8. Joining RSTs.

Lemma 3: After sorting the string encodings of $[P_k]$ in ascending order, the resulting $[P_k]=\{RST_1^k, RST_2^k, \ldots, RST_N^k\}$ is an ordered list. For all i<j and $i,j \in \{1,\ldots,N\}$, let $RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$. Then $prefix(RST_{ij}^{k+1},k+1) = prefix(RST_i^k,k+1)$ holds, and $JR(RST_i^k) = \{RST_{ij}^{k+1} | RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k, j=i+1,\ldots,N\}$ is the result of rightmost branch expansion of RST_i^k except the rightmost leaf node.

Proof: Let s_i be the string encodings of RST_i^k, and let RST_i^k and $RST_j^k \in [P_k]$, i<j. Accordingly, we have $s_i<s_j$. Given the numbering scheme in the G-QPT, it is obvious that the differing node with the smaller number lies in s_i. Suppose node y has a parent node n_i in P_k, and the corresponding differing node in s_j is x whose parent node in P_k is n_j. We have reduced this problem to the case shown in Figure 7. From Theorem 1, we know that $prefix(RST_{ij}^{k+1},k+1) = prefix(RST_i^k,k+1)$. Consequently,

any RST_{ij}^{k+1} ($\in\{RST_{ij}^{k+1} | RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, j=i+1,…,N}) also belongs to the same equivalence class [prefix(RST_i^k,k+1)].

Let *rmln*(RST_i^k) denote the rightmost leaf node of RST_i^k, and P_k=prefix(RST_i^k,k). According to Lemma 1, all the RSTs RST_j^k (j=i+1,…,N) have the same prefix P_k as RST_i^k. When RST_i^k joins with RST_j^k, the RST_{ij}^{k+1} obtained corresponds to a rightmost node expansion. Since { RST_j^k | j=i+1,…,N} contains all the possible nodes of the rightmost branch of RST_i^k except the leaf node *rmln*(RST_i^k), we have JR(RST_i^k)={ $RST_{ij}^{k+1} | RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, j=i+1,…,N}, which is the result of the rightmost branch expansion of RST_i^k except the rightmost leaf node. □

Corollary 1: Let $[P_k]$={ RST_1^k , RST_2^k,…, RST_N^k } be an ordered list. Then the join result set JR(RST_i^k)={ $RST_{ij}^{k+1} | RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, j=i+1,…,N} is in ascending order. □

Example: Consider the equivalence class EC in Figure 6. $[P_3]$={ RST_1^3, RST_2^3, RST_3^3, RST_4^3} is already sorted. Since prefix(RST_{12}^4,4) = prefix(RST_{13}^4,4) = prefix(RST_{14}^4,4) = prefix(RST_1^3,, 4) holds, it can be shown that prefix(RST_{23}^4,4) = prefix(RST_{24}^4,4) = prefix(RST_2^3,4) and prefix(RST_{34}^4,4) = prefix(RST_3^3,4) also holds.

The join result from Lemma 3 causes the RSTs to grow horizontally. All the RSTs of [RST_i^k] can be generated when it is combined with the vertical growth, that is, the rightmost leaf node expansion. Let n_{rml}= *rmln*(RST_i^k) be the rightmost leaf node of RST_i^k, the result of rightmost leaf node expansion is given by *rmlne*(RST).

Lemma 4: Let n_{rml}=*rmln*(RST_i^k), and RST_i^k = P,n_{rml}{,-1}^{m}. Suppose $n_1,\ldots,n_c$ are the children of n_{rml} in ascending order. We have *rmlne*(RST_i^k)={P,n_{rml},n_i,-1{,-1}^{m} | i=1,…,c}, and *rmlne*(RST_i^k) is in ascending order. □

Consider the prefix tree P_3 in Figure 6. Expanding the rightmost leaf node (node 3) of P_3 will generate the candidates RST_1^3 and RST_2^3. Since RST_1^3 and RST_2^3 are obtained by adding nodes 4 and 5 to rmln(P_3) respectively, they are in ascending order.

Based on the results of Lemma 3 and Lemma 4, the following theorem produces an equivalence class of RST.

Theorem 2: Given a k-edge $RST_i^k \in [P_k]=\{RST_1^k$, RST_2^k,…, $RST_N^k\}$ sorted in ascending string encodings, let *rmlne*(RST_i^k)={RST_{ir}^{k+1}| $RST_{ir}^{k+1}$is the rightmost leaf node expansion of RST_i^k} and JR(RST_i^k)={$RST_{ij}^{k+1} | RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, j=i+1,…,N}. Then [prefix(RST_i^k,k+1)]= *rmlne*(RST_i^k)∪JR(RST_i^k) holds.

Proof: Lemma 1 shows that *rmbe*(RST_i^k) forms the equivalence class [prefix(RST_i^k,k+1)] by expanding the right most branch of RST_i^k. Lemma 3 states JR(RST_i^k) is the result of rightmost branch expansion of RST_i^k except the rightmost leaf node. Lemma 4 gives the result of rightmost leaf node expansion *rmlne*(RST_i^k). This concludes the proof of Theorem 2: [prefix(RST_i^k, k+1)] = *rmlne*(RST_i^k)∪JR(RST_i^k) holds. □

Corollary 2: From Lemma 4 and Corollary 1, the equivalence class [prefix(RST_i^k,k+1)]=rmlne(RST_i^k) ∪JR(RST_i^k) obtained in Theorem 2 is in ascending order, and generates all the candidate RSTs without repetition. □

Example: Figure 9 illustrates the application of Theorem 2 to the G-QPT in Figure 5. Note that the equivalence class generated is already in ascending order.

Theorem 2 essentially partitions the RSTs in an equivalence class into two categories: (a) JR(RST_i^k), the set of RSTs generated by joining RSTs, and (b) rmlne(RST_i^k), the set of RSTs generated by the rightmost leaf node expansion.

Lemma 5: Suppose RST_1, $RST_2 \in [P_k]$ are contained in the same QPT. Let $RST_{12}=RST_1 \bowtie RST_2$. Then RST_{12} is also contained in the QPT. Let $[P_k]$={ RST_1^k, RST_2^k,…, RST_N^k } and the join result JR(RST_i^k)={ $RST_{ij}^{k+1} | RST_{ij}^{k+1} = RST_i^k \bowtie RST_j^k$, j=i+1,…,N}. We have $\forall RST_{ij}^{k+1} \in$JR($RST_i^k$), RST_{ij}^{k+1}.tidlist = RST_i^k.tidlist∩ RST_j^k.tidlist. □

Lemma 5 removes the need to match the RSTs generated by joins, that is, $RST_{ij}^{k+1} \in$JR(RST_i^k), with the QPTs in the database. Next, we examine the RSTs in the equivalence class [prefix(RST_i^k,k+1)] that have been generated by the right-most leaf node expansion rmlne(RST_i^k). This set of RSTs can be divided into *single-branch* RSTs and *multi-branch* RSTs. The former are RSTs with one leaf node, while the latter are RSTs with multiple leaf nodes.

Lemma 6: By associating transaction IDs with the QPTs, only *single-branch RSTs* in rmlne() need to be matched with the QPTs in the database. The frequency count for the remaining k+1-edge RSTs is computed from the intersection of the tidlists of corresponding k-edge RSTs.

Proof: Let [RST_i^k]=JR(RST_i^k)∪rmlne(RST_i^k). Given a multi-branch (k+1)-edge $RST^{k+1} \in$rmlne(RST_i^k), we obtain *m* k-edge RSTs by removing one leaf node from each branch at a time. Since the rightmost branch expansion enumerates all the candidates, so these RSTs must be in some equivalence class. The join result of any two of these RSTs produces the (k+1)-edge RST^{k+1} itself. Let $RST^{k+1}= RST_i^k \bowtie RST^k$, where RST^k is produced by removing one leaf node from RST^{k+1}, such that the deleted leaf node is not the rightmost leaf node of RST^{k+1}. If $RST^k \in F_k$, then RST^{k+1}.tidlist can be computed via RST^{k+1}.tidlist= RST_i^k.tidlist∩ RST^k.tidlist. In contrast, for a *single-branch* (k+1)-edge $RST^{k+1} \in$rmlne(RST_i^k), only one k-edge RST exists. Hence, it must be matched against QPTs in the DB. □

Lemma 6 further reduces the number of RSTs to be matched. Consider Figure 9 where RST^3 ("1,3,-1,6,7,-1,-

1") is in the rightmost leaf node expansion of RST^2 ("1,3,-1,6,-1"). RST^3 is a 2-branch RST. By removing its leaf node 3, we obtain the RST ("1,6,7,-1,-1"), which is in equivalence class ["1,6"]. However, this result cannot be applied to a single branch $RST^{k+1} \in \text{rmlne}(RST_i^k)$. Since a single branch RST^{k+1} cannot be the join result of two k-edge RSTs, it has to be matched against the QPTs in the DB.

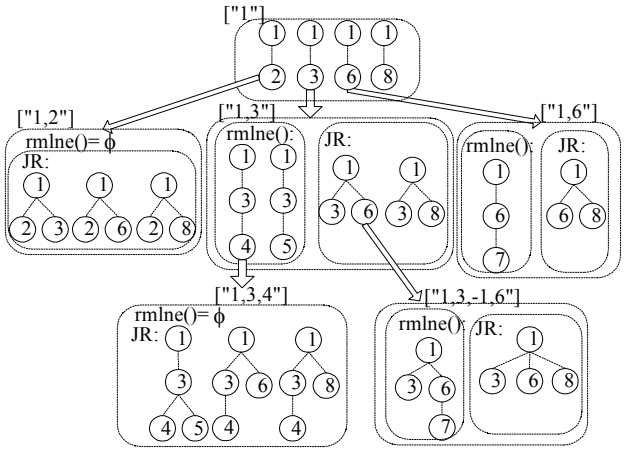


Figure 9. Generating Equivalence Classes.

3.2 Pruning RSTs

Next, we examine how infrequent RSTs can be pruned early. This is achieved by applying the Apriori property: If a subset of an itemset is not frequent, then the itemset itself cannot be frequent. The frequent itemsets of size k-1 serve as filters for candidate itemsets of size k: If a k-edge RST is frequent, then all its (k-1)-edge RSTs must be frequent.

Lemma 7: Let s be the string encoding of RST^k. Then all the RST^{k-1} can be generated by deleting "n,-1" from s each time found when scanning s, where n≠-1.

Proof: By recognizing that each "n,-1" in s represents a leaf node of RST^k, we know that the string encoding s' produced by deleting "n,-1" from s is a (k-1)-edge RST. By scanning s once and each time producing a (k-1)-edge RST when found a "n,-1", all RST^{k-1} are produced. □

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.

Figure 10 shows an example prefix tree of some RSTs generated from G-QPT in Figure 5. Note that nodes at depth k in the prefix tree stores the string encoding of frequent k-edge RST.

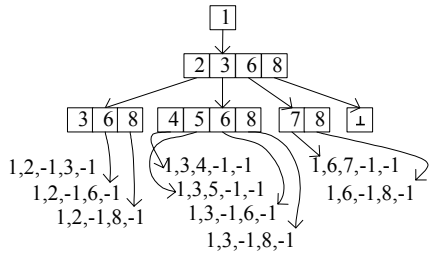


Figure 10. Example Prefix Tree.

Algorithm FastRSTGen (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}- are 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 equivalence class E∈F_k do
3. for each $RST_i^k \in$E do
4. RMLNE = rmlne (RST_i^k);
5. for each *rst*∈RMLNE do
6. if (IsSingleBranch (rst))
7. C_{k+1}←rst;
8. else if exists k-edge RST^k of *rst* in prefix tree and $RST^k \neq RST_i^k$ then
9. *rst*.tidList=RST^k.tidList∩ RST_i^k.tidList;
10. if (|*rst*.tidList|≥ Support)
11. F_{k+1}←*rst*;
12. for $RST_j^k \in$E, i<j do
13. RST^{k+1}= $RST_i^k \bowtie RST_j^k$;
14. RST^{k+1}.tidList= RST_i^k. tidList∩ RST_j^k.tidList;
15. if (|RST^{k+1}.tidList|≥ Support)
16. F_{k+1}←RST^{k+1};
17.return;

Figure 11. Algorithm FastRSTGen.

3.3 Algorithm FastRSTGen

The theorems in sections 3.1 and 3.2 essentially partition the candidate RSTs in an equivalence class into two categories. The first category of RSTs does not need to be matched against the QPTs in the database, while the second category of RSTs are the *single-branch* RSTs that need to be matched against the QPTs in the database.

Figure 11 shows the candidate generation algorithm FastRSTGen. Lines 4-11 process the RSTs obtained by the rightmost leaf node expansion of RST_i^k. Lines 6-7 handle the single-branch RSTs that will be returned as C_{k+1} to be matched against the QPTs in the database. Lines 8-11 prune the non-single-branch *rst* based on the results of Lemmas 5 and 6. That is, if there exists a k-edge RST^k of *rst* that is different from RST_i^k, then *rst*.tidList will be computed from the intersection of RST^k.tidlist and RST_i^k.tidList. If *rst* meets the minimum support criteria, then *rst* is frequent and will be added to F_{k+1}. Lines 12-16 compute the set JR(RST_i^k).Each join result RST^k can be

computed from the tidlists, and added to F_{k+1} if it is frequent.

4. Caching Query Pattern Trees

Frequent QPTs captures the frequent queries issued in the past and they form the ideal candidates for caching. However, many of these queries have overlaps in the answer sets. To fully utilize the limited cache space, we propose the following rewriting heuristic:

Rewriting heuristic: Let F be the set of frequent QPTs. Given a new incoming QPT, we find the most similar frequent pattern tree QPT_{i0} in F. If the difference between QPT and QPT_{i0} is within certain threshold, then the rewritten query is mQPT=merge(QPT,QPT_{i0}). The relevant answers to QPT will be presented to the user, while the query result to mQPT will be cached.

Once the query has been rewritten, we can incorporate the discovered frequent query patterns with existing cache replacement policy. We first differentiate the frequent query patterns from the infrequent ones since the former are more likely to be issued subsequently. When cache replacement is needed, answers to infrequent query patterns are replaced first. If the space for admitting the new query result is still not sufficient, then the cached results corresponding to some frequent query patterns will be replaced according to the existing replacement policy.

Assuming the query result set to be replaced is $\{q_1,q_2,\ldots,q_r\}$, and p_i is the predicted probability of future accessing of q_i, c_i is its execution cost of query q_i, and s_i is its size. The benefit for keeping an incoming query q_i result in cache is: $p_i c_i / s_i$. Therefore, replacement will only occur if

$$pc/s \geq \sum_{j=1}^{r} (p_j c_j / s_j) .$$

Figure 12 outlines a new replacement policy called FQPT_LRU that incorporates frequent QPTs into the least recently used (LRU) replacement policy. The most recently used (MRU) policy can be similarly adapted.

Algorithm FQPT_LRU
1. Replace query results that do not correspond to any frequent query patterns.
2. If there is sufficient space for the results of new query, then admit query, else replace those cached results related to frequent query patterns according to LRU.

Figure 12. Algorithm FQPT_LRU.

5. Performance Study

In this section, we evaluate the performance of FastXMiner and demonstrate the effectiveness of our strategy to cache the frequent query patterns found. The mining algorithms were implemented in C++ and the caching system was in Java. We carried out experiments on a Pentium IV 2.4 GHz with 1 GB RAM, running under Windows XP.

5.1 Efficiency of FastXMiner

Here, we compare the performance of FastXMiner with XQPMiner, a G-QPT schema guided enumeration method [24]. We record the response times of both algorithms when the minimum support varies. We also investigate their scalability by varying the number of QPTs in the database.

The datasets used are SigmodRecord, Shakespears' Play (SSPlay for short) and DBLP whose schemas are SigmodRecord.DTD, SSPlay.DTD and DBLP.DTD. 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 queries, we first enumerate all the RSTs of the G-QPT. Then we use the Zipfian and uniform distributions to produce the transaction file of QPTs from the RSTs.

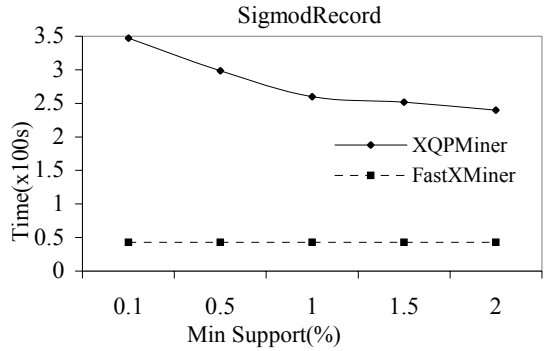
Different datasets have different characteristics (see Table 1). The number of nodes in G-QPT, the maximum depth and fanout of G-QPT give an indication of how many rooted subtrees the G-QPT will have. The total number of RSTs in a G-QPT affects the mining process since they have to be matched and pruned against the QPTs in the database. A RST with "//"consumes more time to compare with QPTs than a RST without "//". In contrast, the average number of nodes, maximum depth and fanout of QPTs reflect the complexity of the dataset. All the datasets follow the default Zipfian distribution except when specified.

Table 1. Characteristics of Datasets Used.

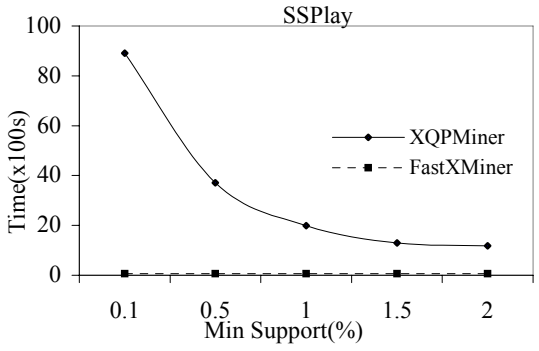
	Datasets	DBLP	DBLP - Uniform	SSPlay	Sigmod Record
G-QPT	Num. of nodes	98	98	67	11
	Max depth	8	8	6	5
	Num. of //	13	13	0	0
	Max fanout	12	12	9	4
QPT in DB	Ave # of nodes	7.4	9.2	7.5	5.5
	Max depth	8	8	6	5
	Max fanout	12	12	9	4

5.1.1 Effect of Minimum Support

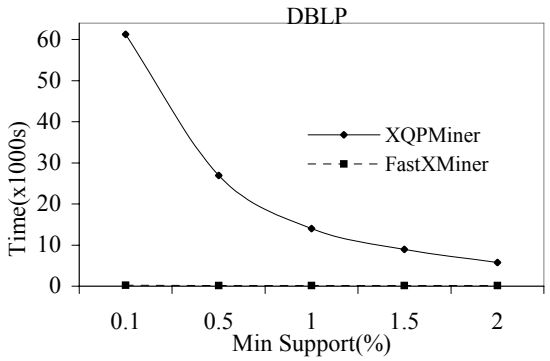
We first investigate the effect of minimum support on the performance of FastXMiner and XQPMiner. Each of the datasets consist of 200,000 QPTs. Figure 13 shows the results. We observe that the response time for FastXMiner is barely affected by the decrease in minimum support, while that for XQPMiner grows rapidly. For example, for the DBLP (uniform) dataset, the time taken by XQPMiner at 2% minimum support is about 6 times more than that at 0.1%. In contrast, the time needed for FastXMiner at support 2% is only about 1.2 times more than that at 0.1%.



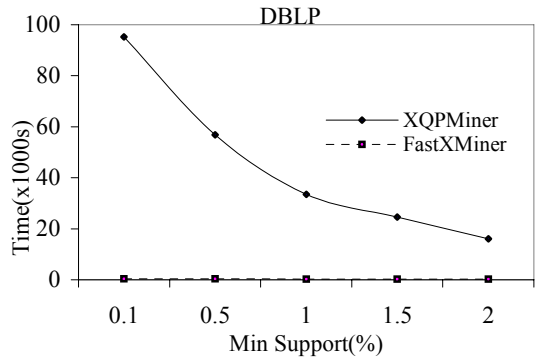
(a) SigmodRecord, 200K, Zipf



(b) SSPlay, 200K, Zipf



(c) DBLP, 200K, Zipf



(d) DBLP, 200K, Uniform

Figure 13. Effect of Varying Minimum Support.

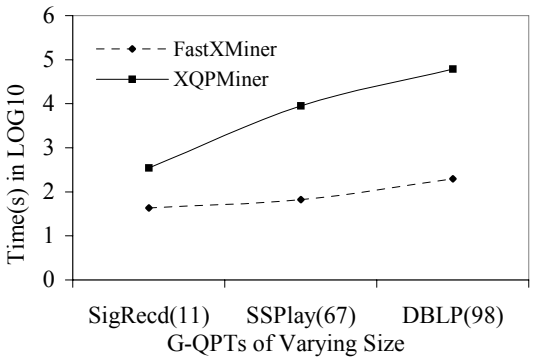


Figure 14. Effect of Varying G-QPT Size.

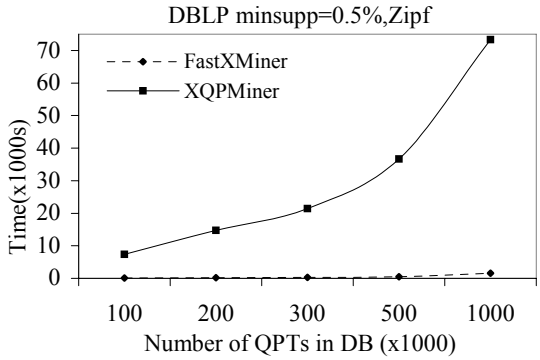


Figure 15. Effect of Varying Number of QPTs.

This is because with decreasing minimum support, XQPMiner needs to match an increasing number of candidate RSTs against the QPTs in the database while FastXMiner is able to avoid a large number of these matches.

We also note from the above experiment that as the number of nodes in the G-QPT increases, FastXMiner outperforms XQPMiner significantly. Figure 14 shows the response time (in log10) of the two methods for varying sizes of G-QPT at minimum support 0.1%. FastXMiner is faster than XQPMiner by 6~8 times for the SigmodRecord DTD, 18~132 times faster for the SSPlay DTD, and 53~273 times faster for the DBLP DTD.

5.1.2 Scalability

Next, we investigate the impact of the number of transactions (or QPTs) in the database on response time. The number of QPTs ranges from 100,000 to 1 million. Figure **15** shows the results for the DBLP dataset. FastXMiner has excellent scale-up as compared to

XQPMiner. For 1 million QPTs, FastXMiner needs only 26 minutes while XQPMiner needs more than 20 hours. This confirms the effectiveness of our approach to reduce the set of RSTs to be matched against the database.

5.2 Effectiveness of Caching Frequent QPTs

In this section, we demonstrate how the frequent query patterns discovered can be used to improve caching performance. The XML caching system is implemented in Java. We use the index scheme [19] to populate the SQL Server 2000 with the DBLP data and create the corresponding indexes. The system accepts tree-patterns as its queries, and utilizes structural join method[3] to produce the result. Note that a tuple here refers to a tree, instead of a row. No optimization techniques are used.

To improve the XML query execution speed, we extract the data of dblp/inproceedings, and add two more elements: review and comments. These two attributes are used to store the rarely retrieved data and relatively large amounts of text (about 1-2KB). The adapted XML data file size is 81.2MB with 3 levels.

Two sets of experiments that investigate the effect of varying number of queries and varying cache size are carried out using 4 replacement policies, namely, traditional LRU and MRU, FQPT_LRU and FQPT_MRU. The latter two policies are obtained by incorporating frequent query patterns into LRU and MRU respectively.

Table 2 shows the probabilities used to generate the XML queries for the DBLP data. These queries fall into two categories: frequent queries and infrequent queries.

Table 2. Probabilities of Queries.

Probability	Query types
0.04	Infrequent query patterns
0.16	6 group of frequent query patterns

The infrequent query pattern set is generated with probability of 0.04. Within the infrequent query set, we generate the actual XML query using the following probabilities: Dblp(1), inproceedings(1), key(0.1), author(0.7), title(0.9), year(1), pages(0.5), crossref(0.4), booktitle(0.7), ee(0.5), url(0.2), review(0.9), comments(0.3). Note that *dblp* and *inproceedings* always appear, and *year* is used as the predicate. A uniform distribution is employed for the predicate values.

There are 6 groups of frequent XML queries. Each of them is generated with probability 0.16. We show the query template for two groups.

- Group 1: Dblp(1), inproceedings(1), key(1), title(1), year(1), crossref(1), ee(0.9)
- Group 2: Dblp(1), inproceedings(1), key(1), title(1), year(1), URL(1), ee(0.9)

We generate 100,000 XML queries that are processed by FastXMiner with a minimum support of 6%.

5.2.1 Performance Metrics

Hit-rate is not an appropriate evaluation metric because some queries can only use part of the cached results. Instead, we utilize *cost-saving ratio* and *average response time* as the performance metrics in our experiments.

Cost Saving Ratio is defined as $\sum_i H_i C_i / \sum_i C_i R_i$, where H_i is the number of times Q_i can be answered by the cache, R_i is the total number of times Q_i is issued, and C_i is the execution cost of query Q_i.

Average Response Time is the average time taken to answer a query. It is defined as the ratio of total execution time for answering a set of queries to the total number of queries in this set.

5.2.2 Effect of Varying Number of Queries

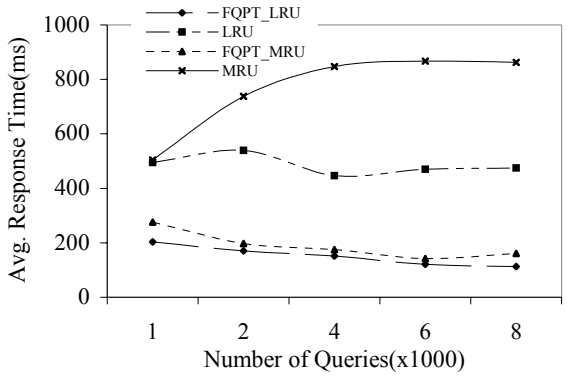
In this set of experiment, we investigate the effect of number of queries on cache performance. We vary the number of queries from 1000 to 8000, and fix the cache size at 40MB. The result is shown in Figure 16. We observe that FQPT_* replacement policies give better performance than those without frequent query patterns. The average response time is about 3~4 times lower. Furthermore, the average response time of FQPT_* decreases as the number of queries increases. The graph for cost saving ratio (Figure 16(b)) shows a similar trend.

5.2.3 Effect of Cache Size

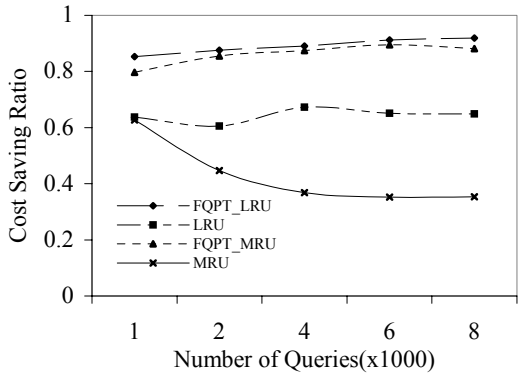
Next, we investigate the effect of cache size on query performance. We vary the cache size from 10MB to 80MB, and fix the number of queries at 4000. Figure 17(a) shows that the gap in the average response time for the FQPT_* policies and those without using frequent query patterns increases with the cache size initially, and then peaks at around 40M. As the cache size continues to increase, the gap gradually narrows since most of the data would be found in the cache. Again, the graph for cost saving ratio indicates similar trend (Figure 17(b)).

6. Related Work

Finding frequent substructures from graphs first gains its focus in DNA/RNA research. [23] devises an algorithm to discover approximate common subtrees in multiple RNA secondary structures. [12] gives an efficient algorithm to find frequent substructure describing the carcinogenesis of chemical compounds from labelled graphs, and applied it to predict the functions of chemical compounds. The prevalence of the World Wide Web has also prompted works to find frequent substructures in Web documents [1, 21, 22, 25]. The objective is to discover the frequent substructures from a collection of semi-structured data objects (files) of similar structure. [22] employs 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.

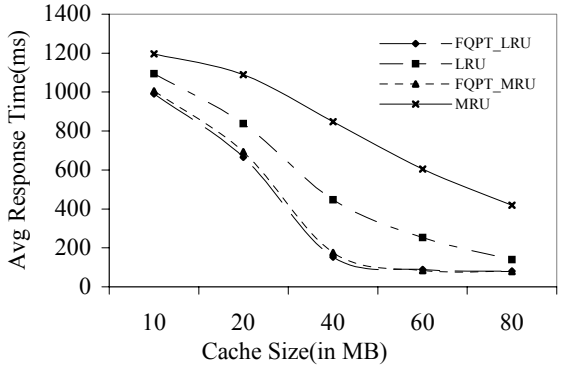


(a) Average Response Time (Cache Size 40MB)

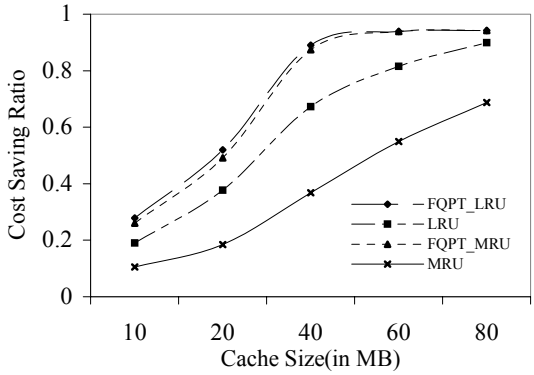


(b) Cost Saving Ratio (Cache Size 40MB)

Figure 16. Effect of Varying Number of Queries.



(a) Average Response Time (4000 queries)



(b) Cost Saving Ratio (4000 queries)

Figure 17. Effect of Varying Cache Size.

[25] 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. 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. This would not be efficient if applied to our work.

FREQT[1] and TreeFinder[21] aim to find frequent subtrees from a collection of semi-structured documents. FREQT considers only subtree inclusion and proposes a subtree enumeration method that is similar to [25]. TreeFinder employs tree subsumption to approximate the result in order to achieve scalability. TreeFinder first transforms the tree into label pairs representing the transitive closure of ancestor relationship. A standard Apriori method is used to mine frequent label pair sets. For each support tree set of a frequent label pair set, TreeFinder has a generalization step to construct the maximal common tree. The main limitation of TreeFinder is that it can only find a subset of the actual set of frequent trees.

All the above techniques are not appropriate for mining XML query patterns since these patterns contains special characters wildcard '*' and relative paths '//'.

Semantic caching has received extensive attention both in database and web areas [6, 11]. The recent move towards intelligent web caching tries to adapt to changes in usage patterns by constructing predictive models of user requests by mining web log data [4, 18]. However, caching XML query results is still a relatively new area [10]. To the best of our knowledge, this is a first work to exploit frequent query patterns for caching XML data.

7. Conclusion and Future Work

In this paper, we have described an efficient algorithm, FastXMiner, to discover frequent rooted subtrees from XML queries. FastXMiner enumerates only valid candidates RSTs. We develop theorems to prove that only a small subset of the generated candidate patterns needs to undergo expensive tree containment tests. Experiments results reveal that FastXMiner has good response time and scales well.

We have also discussed how the results of the discovered frequent queries patterns can be incorporated into a caching system. The experimental results demonstrate that incorporating frequent query patterns can help to improve the performance of a XML query system significantly.

Future work includes extending the mining algorithm to handle query patterns with predicates, and investigating how frequent query patterns can be applied to the problem of view selection. By incorporating user information, the discovery of frequent query patterns will reflect the user preferences and requirements. This is especially useful in designing data warehouses for XML.

References

[1] T. Asai, K. Abe, S. Kawasoe, et. al. Efficient Substructure Discovery from Large Semi-structured Data. 2nd SIAM Int. Conference on Data Mining, 2002.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, et. al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. IEEE ICDE, 2002.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. VLDB, pp:487-499, 1994.

[4] F. Bonchi, F. Giannotti, C. Gozzi, G. Manco, et.al. Web log data warehousing and mining for intelligent web caching. Data and Knowledge Engineering, 39(2):165-189, 2001.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. ACM SIGMOD, 2002.

[6] B. Chidlovskii, U. M. Borghoff. Semantic Caching of Web Queries. VLDB Journal 9(1): 2-17, 2000.

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

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

[9] C.W. Chung, J.K. Min, K. Shim, APEX: An Adaptive Path Index for XML data, ACM SIGMOD, 2002.

[10] L. Chen, E. A. Rundensteiner, S. Wang. XCache-A Semantic Caching System for XML Queries. Demo in ACM SIGMOD, 2002.

[11] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, M. Tan. Semantic Data Caching and Replacement. VLDB, pp:330-341, 1996.

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

[13] T. Grust. Accelerating XPath Location Steps. ACM SIGMOD 2002.

[14] F. Giunchiglia and T. Walsh. Tree Subsumption: Reasoning with Outlines. 10th European Conference on Artificial Intelligence, pp:72-76, 1992.

[15] 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.

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

[17] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. IEEE Int. Conference on Data Mining, pp: 313-320, 2001.

[18] B. Lan, S. Bressan, B. C. Ooi, K. L. Tan. Rule-assisted prefetching in Web-server caching. ACM CIKM, pp: 504 - 511, 2000.

[19] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. VLDB, pp:361-370, 2001.

[20] R. Ramesh and L.V. Ramakrishnan. Nonlinear pattern matching in trees. Journal of the ACM, 39(2):295-316, 1992.

[21] A. Termier, M. C. Rousset, M. Sebag. TreeFinder: a First Step towards XML Data Mining. IEEE 2nd International Conference on Data Mining, 2002.

[22] K. Wang, H. Liu, Discovering Structural Association of Semistructured data, IEEE TKDE, 12(3):353-371, 2000.

[23] 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.

[24] L. H. Yang, M. L. Lee, W. Hsu. Mining Frequent Query Patterns in XML. 8th Int. Conference on Database Systems for Advanced Applications (DASFAA), 2003.

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