XML 查询模式发掘*

陈 义[1+], 王裕国[1], 杨良怀[2]

[1](中国科学院 软件研究所,北京 100080)
[2](北京大学 计算机科学系,北京 100871)

Discovering Frequent Tree Patterns from XML Queries

CHEN Yi[1+], WANG Yu-Guo[1], YANG Liang-Huai[2]

[1](Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)
[2](Department of Computer Science, Peking University, Beijing 100871, China)
+ Corresponding author: Phn: +86-10-64751021, E-mail: chenyi72@tom.com, http://www.ios.ac.cn
Received 2003-10-24; Accepted 2004-05-08

Chen Y, Wang YG, Yang LH. Discovering frequent tree patterns from XML queries. *Journal of Software*, 2004,15(Suppl.):114~123.

Abstract: With XML being the standard for data encoding and exchange over Internet, how to manage XML data efficiently becomes a critical issue. An effective approach to improve the performance of XML management systems is to discover frequent XML query patterns and cache their results. Since each XML query can be modeled as a tree structure, the problem of discovering frequent query patterns can be reduced to frequent structure mining. However, mining frequent query patterns is much more complex than simple structure mining since we have to consider the semantics of query patterns. In this paper, we present an approach to discover frequent XML query patterns efficiently. Compared with previous works, our approach is strictly based on the semantics of XML queries, its mining results are more precise, and can be more effectively utilized by caching system.
Key words: pattern tree; frequent pattern; query caching; XML query

摘 要: 随着 XML 的广泛应用,如何高效地处理 XML 查询受到越来越多的关注.发掘频繁使用的查询模式,并缓存其查询结果是提高查询效率的有效手段之一.由于 XML 查询可以表示为树,因而可以使用频繁结构挖掘(Frequent Structure Mining)的技术来发掘频繁查询模式.提出一种高效的频繁查询模式发掘方法,同以前的类似工作相比,所做工作是严格地基于 XML 查询语义的,因而发掘结果更为准确,也更易于使用.
关键词: 模式树;频繁模式;查询缓存;XML 查询

* **CHEN Yi** was born in 1972. He received his Ph.D. degree at Institute of Software, the Chinese Academy of Sciences in 2004. His recent research interests include XML query processing and data mining. **WANG Yu-Guo** was born in 1941. He is a professor and doctoral supervisor at Institute of Software, the Chinese Academy of Sciences. His current research interests include computer graphics, multimedia and CIMS application. **YANG Liang-Huai** was born in 1967. He is an associate professor at the Peking University. His research is focused on data mining and XML data management.

1 Introduction

As XML prevails over Internet as a medium for data exchange and representation, how to process XML queries efficiently becomes the imperative research issue. Query caching is one of the most promising approaches to reduce the query processing time. By analyzing the user logs, frequently occurred query patterns can be identified and their result can be materialized to improve the system efficiency. How to cache XML query result gains its focus recently[5,11] and finding the frequent query patterns plays a critical role in a XML caching system.

Frequent structure mining (FSM) is a new direction in the field of data mining that has been intensively studied recently[1,9,12]. Given a pattern tree (or pattern graph) S and a set of data trees (or data graphs) $D=\{S_1,\ldots,S_n\}$, usually denoted as the transaction database, we say S occurs in D if we can find an element S_i in D such that there is certain mapping relationship between S and S_i. FSM aims to find the pattern trees (or pattern graphs) that occur frequently over the set D. Since XML queries can be modeled as trees, the problem of discovering frequent query patterns can be reduced to FSM. However, mining frequent query patterns is much more complex than the mining of simple structures because of the semantic issues of query patterns.

In this paper, we propose an approach for frequent query pattern mining strictly based on the semantics of XML queries, and present several techniques to optimize the mining process. Through utilization of the rightmost occurrences of pattern trees in the transaction database, we prove that in order to obtain the frequencies of pattern trees, only single branch pattern trees need to be matched against the transaction database, and the frequencies of multi-branch pattern trees can be figured out through reuse of intermediate results. Experiments show that our method results in substantial performance gains.

The rest of the paper is organized as follows. Basic concepts used in this paper are given in Section 2. Section 3 describes the candidate enumeration method used in this paper. Section 4 presents our approach to discover frequent query patterns. Section 5 gives the results of the experiment study; Section 6 discusses the related works, and we conclude in Section 6.

2 Problem Statement

In this paper, we consider selection patterns in the syntax of XPath[4], a popular pattern language that are generally modelled as query pattern trees[7].

Definition 1 (Query Pattern Tree). A *query pattern tree* is a rooted tree $QPT=\langle V,E\rangle$, where V is the vertex set, E is the edge set. The root of the query pattern tree is a distinguished node denoted by root(QPT). Each vertex v has a label, denoted by v.label, whose value is in {"*"} $\cup$ tagSet, where the tagSet is the set of all element names in the context. A distinguished subset of edges representing ancestor-descendant relationships is called descendant edges.

Figure 1 (a)-(d) shows three QPTs QPT_1, QPT_2, and QPT_3. Descendant edges are shown with dotted lines in diagrams. In what follows, given a query pattern tree $QPT=\langle V,E\rangle$, sometimes we also refer to V and E with QPT if it's clear from the context. Given an edge $e=(v_1,v_2)\in QPT$ where v_2 is a child of v_1, sometimes v_2 will be denoted as a d-child of v_1 if e is a descendant edge, and as a c-child otherwise. Given two c-children (or d-children respectively) of a QPT node, we say they are duplicate siblings if they share the common label l.

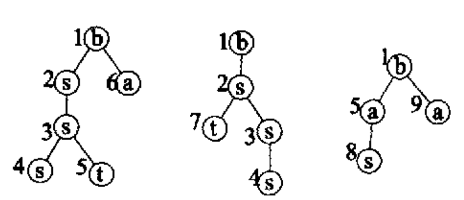


Fig.1 Sample QPTs

Given a query pattern tree QPT, a *rooted subtree* RST of QPT is a subtree of QPT such that $root(RST)=root(QPT)$ holds. One of rooted subtrees of QPT_1 is shown in Fig.2(a). Given a tree node $v\in QPT$, subtree(v) denotes the subtree T=⟨V',E'⟩ of QPT rooted at v. Let QPT be a pattern tree, the size of QPT is defined by the number of its nodes |QPT|. An RST of size k+1 will be denoted as a k-edge RST sometimes. An RST will also be denoted as a single branch RST if it has only one leaf node, and as a multi-branch RST otherwise.

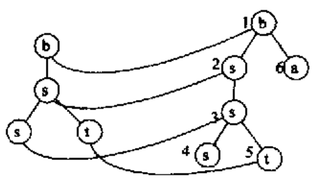
To discover frequent query patterns, one important issue is how to test the occurrence of a tree pattern in the transaction database. Intuitively, we need to discover query patterns whose outputs are more likely to be reused by other queries. The topological mapping method, used by most previous works [1,9], is not applicable to query pattern mining because for query patterns we must take the semantics into account. The subtree embedding approach used by Ref.[12] is too restrictive since it requires the pattern trees preserve ancestor-descendant relationships. In this paper, we use the concept of tree subsumption, a sound (but not complete) approach to test containment of query pattern trees[7].

Definition 2(Tree Subsumption). Given two QPTs QPT and QPT′, QPT is subsumed in QPT′, denoted as QPT⊑QPT′, iff there exists a simulation relation sim between nodes of QPT and nodes of QPT′, such that:

1) ⟨root(QPT), root(QPT′)⟩∈sim;

2) v.label = v'.label or v'.label = "*", if ⟨v,v'⟩∈sim;

3) if ⟨v_1,v'_1⟩∈ sim and v_2 is a c-child of v_1 in QPT, there must exist some c-child v'_2 of v'_1 in QPT′ such that ⟨v_2,v'_2⟩∈ sim; if ⟨v_1,v'_1⟩∈ sim and v_2 is a d-child of v_1 in QPT, there must exist some proper descendant v'_2 of v'_1 in QPT′ such that ⟨v_2,v'_2⟩∈sim.



(a) RST (b) QPT_1

Fig.2 Tree subsumption

Figure 2 shows the subsumption of RST in QPT_1.

Given a transaction database $D=\{QPT_i| i=1,\ldots,n\}$, we say RST occurs in D if RST is subsumed in a query pattern tree $QPT_i \in D$. The *frequency* of RST, denoted as Freq(RST), is the total occurrence of RST in D, and *supp(RST)=Freq(RST)/|D|* is its *support rate*. Given a transaction database D and a positive number $0<\sigma\le 1$ called the minimum support, mining the frequent query patterns of D means to discover the set of RSTs of D, $F_D=\{RST_1,\ldots,RST_m\}$, such that for each $RST \in F_D$, $supp(RST) \ge \sigma$.

For example, assume that there are three QPTs QPT_1, QPT_2, and QPT_3 in the transaction database as shown in Fig.1, and σ = 0.7, the RST in Fig.2 (a) occurs in three QPTs, thus it's frequent with respect to this database with *supp(RST)*=3/4 and *Freq(RST)*=3.

3 Candidate Generation

In our settings, each data tree is a QPT. Given a transaction database D={QPT$_i$| i=1,…,n}, its *global query pattern tree* G-QPT is constructed by merging all QPTs in D. Figure 3 shows an example of G-QPT obtained from the QPTs in Fig.1 (a), (b), and (c). Note that for each QPT with duplicate siblings s_1, …,s_n sharing the common label l, we need re-label them with $l^1,\ldots,l^n$ respectively before the merging process begins, and re-label them back with l after the merging process has finished. Clearly, when duplicate siblings are present, the merging result is not unique. Under such situation we can choose one merging result arbitrarily because it will not influence the succeeding processing.

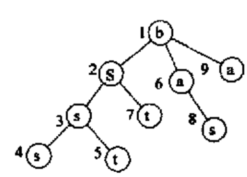


Fig.3 G-QPT

The nodes of G-QPT can be numbered from 1 to |G-QPT| through a pre-order traversal. Because each node of QPT∈D is merged with a unique node of the G-QPT, each node of QPT has the same number as the corresponding node in the G-QPT. For example, in Figure 1, integers outside of circles are numbers of corresponding tree nodes. After labeling each node with a number, the representation of QPT_2 can be simplified to string format "1, 2, 3, 4, -1, -1, 7, -1, -1" as in Ref.[12]. We will call such encoding strings as *string encodings* of QPTs.

As in Ref.[11], in this paper, we only consider pattern trees that are rooted subtrees of some QPTs in the transaction database. Since each QPT is a rooted subtree of the G-QPT, each pattern tree is also a rooted subtree of the G-QPT. We enumerate all candidate RSTs with the schema-guided right most expansion method proposed in Ref.[11]. It's not difficult to prove that as in Ref.[11], given a k-edge rooted subtree RST^k in equivalence class $[RST^{k-1}]$, each k+1-edge rooted subtree can be generated with two kinds of operations: the right most leaf node expansion (RMLNE) of RST^k, or the join of RST^k and another rooted subtree $RST^{k'}$ in the same equivalence class (sometimes the join of RST^k and $RST^{k'}$ will be denoted as $RST^k \bowtie RST^{k'}$). And all the k+1-edge rooted subtrees will be enumerated in ascending order.

Figure 4 shows an example of candidates generated from the G-QPT in Fig.3. Due to the presence of duplicate-siblings, some RSTs are likely to be enumerated more than once in our settings, which will incur extra expense. For example, in, RSTs "1, 6, −1" and "1, 9, −1" are equivalent RSTs indeed. However, due to the space limitation, we will not consider this issue further and simply process them as different RSTs.

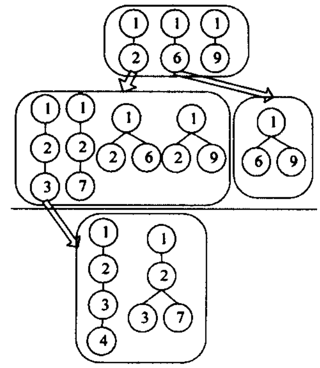


Fig.4 Candidate enumeration

4 Algorithm

In this section we present an Apriori-based algorithm, *FQPMiner*, to discover frequent RSTs, then propose several techniques to optimize the mining process.

4.1 Discovering of frequent RSTs

The main framework of our algorithm *FQPMiner* is shown in Figs.5 and Fig.6. In the algorithm, the notation RST^k denotes a k-edge rooted subtree; F_k is a set of frequent k-edge rooted subtrees. Most of the work is finished in the function *FQPGen* (see Figure 6), which uses the schema-guided rightmost enumeration method to enumerate candidate RSTs level-wise, counts the frequency of each candidate RST, and prunes infrequent RSTs based on the anti-monotone property of tree subsumption.

Algorithm:FQPMiner(D,minSupp)
Input: D—transaction database
minSupp—the minimum support
Output: the set of all frequent RST sets
(1)F_1={1-edge rooted subtrees in D};
(2)for(k=1; $F_k \neq \phi$;k++) do
(3) F_{k+1}=**FQPGen**(F_k, minSupp, D);
(4)return $\{F_i \mid i=1,\dots,k-1\}$;

Fig.5 Algorithm to find frequent RSTs

To count the frequency of an RST, a naive solution is to match it against each QPT in the transaction database. This method will be very inefficient because tree matching is expensive. *FQPGen* can count the supports of candidates more efficiently through utilization of transaction IDs. For each RST, *FQPGen* maintains a TIDList attribute that contains transaction IDs of QPTs in which the RST occurs. We can easily prove that an RST is subsumed in a QPT only if all its rooted subtrees are subsumed in the QPT. Hence, only 1-edge RSTs need to be matched against each QPT in the transaction database. To count the frequency of a k-edge RST RST^k (k>1), if RST^k is a single-branch RST, *FQPGen* need only to match RST^k against QPTs whose transaction IDs are in the set RST^{k-1}.tidlist, where RST^{k-1} is the k-1-edge RSTs that is a rooted subtree of RST^k (lines 4-7 of Fig.6). If RST^k is a multi-branch k-edge RST, then *FQPGen* need only to match RST^k against QPTs whose transaction IDs are in the set

RST_1.tidlist$\cap$ RST_2.tidlist, where RST_1 and RST_2 are k-1-edge RSTs that are rooted subtrees of RST^k (lines 8-18 of Fig.6). To check whether an RST exists in F_k, we can use the index structure EC-Tree proposed by Ref.[11], with which the lookup time of the RST is about $O(L)$, where L is the length of the string encoding of the RST.

In the algorithm, *Contains* (lines 6, 12, and 17) is a function based on the definition of tree subsumption in Section 3. The basic idea is to look for a simulation between two trees by constructing a relation R and then removing from R all pairs (u,u') that will violate condition (3) of the definition of tree subsumption. We will not give its detail because of its strong resemblance with the algorithm presented in Ref.[7].

```
Algorithm: FQPGen (F_k, minSupp, D)
Input: D: transaction database
minSupp: the minimum support
       F_k: the set of frequent k-edge RSTs
Output: the set of all frequent RST sets
(1) F_{k+1}=ϕ;
(2) for each rst ∈ F_k
(3)      for each rst' obtained by expansion of rst
(4)          if rst' is a single-branch RST
(5)              for each transaction t∈D such that t.tid∈rst.tidlist do
(6)                  if Contains(t, rst')
(7)                      rst'.tidlist←t.TID;
(8)          if rst' is a multi-branch RST generated through RMLNE
(9)              obtain RST_k by cutting off one leaf of rst';
(10)             tempList = rst.tidlist∩ RST_k.tidlist;
(11)             for each transaction t∈D such that t.tid ∈ tempList do
(12)                 if Contains(t, rst')
(13)                     rst'.tidlist←t.TID;
(14)         if rst' is generated through join of rst and RST_k
(15)             tempList = rst.tidlist∩ RST_k.tidlist;
(16)             for each transaction t∈D such that t.tid ∈ tempList do
(17)                 if Contains(t, rst')
(18)                     rst'.tidlist←t.TID;
(19)         if (|rst'.tidList|≥ minSupp)
(20)             F_{k+1}← rst;
(21) return F_{k+1};
```

Fig.6 Candidate generation algorithm

4.2 Optimization

In this subsection, we propose a technique for further optimization of the mining process. We prove that through utilization of rightmost occurrences, multi-branch RSTs needn't be matched against QPTs even in our settings.

Definition 3 (Proper Combination). Given an rooted subtree RST and a query pattern tree QPT, where $\langle v_1,\ldots,v_m\rangle$ is the set of nodes of RST sorted in pre-order, assume that RST $\subseteq$ QPT holds and sim is the simulation relation between their nodes, then the *Proper Combinations* of sim is the set PC= $\{\langle v'_1,\ldots,v'_m\rangle|\ v'_i\in V',\ i=1,\ldots,m\}$ such that for each $\langle v'_1,\ldots,v'_m\rangle\in$PC: if v_j is a c-child of v_k, v'_j must be a c-child of v'_k; if v_j is a d-child of v_k, v'_j must be a proper descendant of v'_k, where j, k are any integers such that $1\le j, k\le m$ holds.

Definition 4 (Rightmost Occurrence). Given an rooted subtree RST and a query pattern tree QPT where the list $L=\langle v_1,\ldots,v_m,\ldots,v_k\rangle$ is the set of nodes of RST sorted in pre-order, v_m and v_k is the second rightmost leaf and the rightmost leaf of RST respectively, assume that RST$\subseteq$QPT holds, sim is the simulation relation between their nodes, then the *rightmost occurrence* of RST in QPT is the set rmo(RST,QPT)=$\{\langle v'_m,v'_k\rangle|\langle v'_m,v'_k\rangle$ is a sub-list of some proper combination $\langle v'_1,\ldots,v'_m,\ldots,v'_k\rangle$ of sim$\}$. Note that if the RST has only one leaf node, then the first element of its rightmost occurrence should be null. Given an RST, a transaction database D and its global query pattern tree G-QPT, the *rightmost occurrence* of RST in D is the set rmo(RST,D)=$\{\langle u,v,\{$QPT.tid|for all QPT$\in D$ such that $\langle u,v\rangle\in$rmo(RST, QPT)$\}\rangle|\ \langle u,v\rangle\in$rmo(RST,G-QPT)$\}$.

Definition 5 (Conditional Rightmost Occurrence). Given a rooted subtree RST and a query pattern tree QPT, where $\langle v_1,\ldots,v_i,\ldots,v_m\rangle$ is the set of nodes of RST sorted in pre-order, and v_i is a node of the rightmost branch of RST, assume that RST⊆QPT holds, sim is the simulation relation between their nodes, and $\langle v_i,v'\rangle\in$sim, we define the *Conditional Rightmost Occurrence* satisfying $\langle v_i,v'\rangle\in$sim as the set $\{v'_m \mid$ there exists a proper combination of sim $\langle v'_1,\ldots,v'_i,\ldots,v'_m\rangle$ such that $v'_i=v'\}$. We denote it as $\text{rmo}_{\langle vi,\,v'\rangle\in\text{sim}}$(RST, QPT).

For example, given the transaction database composed of QPT_1, QPT_2, and QPT_3 as shown in Fig.1 (assume 1, 2, and 3 are their respective transaction IDs), and the global query pattern tree G-QPT in Fig.3, if sim is the simulation relation between RST "1, 2, 3, -1, 7, -1, -1" and G-QPT, then the proper combinations of sim are {⟨1, 2, 3, 5⟩, ⟨1, 2, 3, 7⟩}, the rightmost occurrence of the RST in G-QPT will be {⟨3, 5⟩, ⟨3, 7⟩}, the rightmost occurrence of the RST in the transaction database will be {⟨3, 5, {1}⟩, ⟨3, 7, {2}⟩}. Clearly, the frequency of an RST freq(RST) = |{tid|tid∈TIDList, ⟨ u, TIDList ⟩∈ rmo(RST, D)}|.

Lemma 1. Given a transaction database D, its global query pattern tree G-QPT, and two k-edge RSTs RST_1, RST_2 ∈[RST^{k-1}], let RST^{k+1}= RST_1 ⋈ RST_2, p is the node of RST^{k+1} not present in RST_1 (i.e., the rightmost leaf of RST_2), and the junction node q is parent of p, then we have:

1) If RST_1 is a single branch RST, then rmo(RST^{k+1}, D) = {⟨ u, u', $TIDList_1 \cap TIDList_2$⟩| ⟨v, u, $TIDList_1$⟩⊆ rmo(RST_1, D), ⟨v', u', $TIDList_2$⟩⊆ rmo(RST_2, D), where $u \in \text{rmo}_{\langle q,\,q'\rangle\in\text{sim}}(RST_1$, G-QPT), $u' \in \text{rmo}_{\langle q,\,q'\rangle\in\text{sim}}(RST_2$, G-QPT), and $q'\in$ G-QPT }.

2) If both RST_1 and RST_2 are multi-branch RSTs, and the parent of the rightmost leaf of RST_1 has only one child, then rmo(RST^{k+1},D)={⟨u, u', $TIDList_1 \cap TIDList_2$⟩| ⟨v, u, $TIDList_1$⟩⊆ rmo(RST_1, D), ⟨v', u', $TIDList_2$⟩⊆ rmo(RST_2, D), $u \ll v'$, where $u \in \text{rmo}_{\langle q,\,q'\rangle\in\text{sim}}(RST_1$, G-QPT), $u' \in \text{rmo}_{\langle q,\,q'\rangle\in\text{sim}}(RST_2$, G-QPT), and $q' \in$ G-QPT }. Here $u \ll v'$ means v' is the parent (or ancestor respectively) of u if the rightmost leaf of RST_1 is a c-child (or d-child respectively) of its parent.

1) Otherwise, rmo(RST^{k+1}, D) = {⟨u, u', $TIDList_1 \cap TIDList_2$⟩| ⟨v, u, $TIDList_1$⟩⊆ rmo(RST_1, D), ⟨v', u', $TIDList_2$⟩ ⊆ rmo(RST_2, D), u = v', where $u \in \text{rmo}_{\langle q,\,q'\rangle\in\text{sim}}(RST_1$, G-QPT), $u' \in \text{rmo}_{\langle q,\,q'\rangle\in\text{sim}}(RST_2$, G-QPT), and $q' \in$ G-QPT }.

Lemma 1 can be proved based on the definition of tree subsumption. We will not give the detail due to space limitation. Based on Lemma 1, all RSTs generated through join operations are not required to match with QPTs. Now we turn to investigate RSTs generated through rightmost leaf node expansion.

Lemma 2. For any multi-branch k+1-edge rooted subtree RST^{k+1} generated through rightmost leaf node expansion of a k-edge rooted subtree RST_1, there must be another k-edge rooted subtree RST_2 which is formed by cutting off the second rightmost leaf node of RST^{k+1} such that the join of RST_1 and RST_2 will produce RST^{k+1} itself. If RST_2 exists in F_k, then we have: rmo(RST^{k+1}, D) = {⟨v, u', $TIDList_1 \cap TIDList_2$⟩| ⟨v, u, $TIDList_1$⟩⊆ rmo(RST_1, D), ⟨v', u', $TIDList_2$⟩⊆ rmo(RST_2, D), $u' \ll u$}, otherwise, RST^{k+1} must be infrequent.

Based on Lemma 1 and Lemma 2, we have the following result:

Theorem 1. By using the rightmost occurrence of RST in D, only those single-branch RSTs need to be matched with QPTs. The frequencies of other RSTs can be computed through reusing intermediate results.

Based on the above discussion, we are in a position to describe our algorithm *FQPMinerRMO*. Its main framework is similar to *FQPMiner*, but the candidate generation algorithm *FQPGen* is adapted into *FQPGenRMO* in Fig.7. *FQPGenRMO* associates each RST with its rightmost occurrence in the transaction database. Lines 6-7 deal with single-branch RSTs. We use *GetRMO* algorithm to match single-branch RSTs against QPTs to obtain their rightmost occurrences (line 7). *GetRMO* is an extension to *Contains* algorithm used in *FQPMiner*, but its detail is not included in this paper. Lines 8-12 cope with multi-branch RSTs generated through RMLNE using Lemma 2, Lines 13-17 handle RSTs generated through join using Lemma 1. For simplification of representation, only case (1) of Lemma 1 is illustrated in Fig.7. Line 18 obtains TIDList of a rooted subtree from its rightmost occurrence. Lines

19-20 check whether the result subtree is frequent, and those frequent ones are added as members into F_{k+1}.

While expanding an n-edge RST rst_n to generate an n+1-edge RST rst_{n+1}, we obtain the rightmost occurrence rmo(rst_{n+1}, D) through computation over rmo(rst_n, D) and rmo(rst'_n, D), where rst'_n is another n-edge RST. We can prove that $rst_n \le rst'_n$ always holds (definition of order relationship ≤ can be found in Ref.[11]), it implies that after the expansion of an RST rst_n, its rightmost occurrence rmo(rst_n, D) will not be used, hence, needn't be maintained, any longer. Based on the above discussion, *FQPGenRMO* can remove the rightmost occurrence information as early as possible (line 21), and the memory cost will be reduced drastically.

```
Algorithm: FQPGenRMO(Fk, minSupp, D)
Input: D: transaction database
minSupp: the minimum support
      Fk: the set of frequent k-edge RSTs
Output: the set of all frequent RST sets
(1) Fk+1=φ;
(2) for each rst ∈ Fk
(3)      for each rst' obtained by expansion of rst
(4)           rmo(rst', D) ={};
(5)           if rst' is a single-branch RST
(6)                for each transaction t∈D such that t.tid ∈rst.tidlist do
(7)                     rmo(rst', D) = rmo(rst', D) ∪{GetRMO(rst', t)}
(8)           if rst' is a multi-branch RST generated through RMLNE
(9)                obtain RSTk by cutting off the second rightmost leaf of rst';
(10)               for each pair (<v, u, tidlist1>∈rmo(rst, D), < v', u', tidlist2>∈rmo(RSTk, D))
(11)                    if u' << u
(12)                         rmo(rst', D) = rmo(rst', D) ∪{<v, u', tidlist1∩tidlist2>};
(13)          if rst' is generated through join of rst and RSTk, and j is the junction node
(14)               GetCondRMO(rst, G-QPT, j, rmo(rst, G-QPT), cond-rmo);
(15)               GetCondRMO(RSTk, G-QPT, j, rmo(RSTk, G-QPT), cond-rmok);
(16)               for each q such that <q, u>∈ cond-rmo, <q, u'>∈ cond-rmok
(17)                    rmo(rst', D) = rmo(rst', D) ∪{< u, u', tidlist1∩tidlist2>|
                              <v, u, tidlist1>∈ rmo(rst, D), <v', u', tidlist2>∈rmo(RSTk, D)};
(18)          rst'.tidlist={t.TID| t.TID∈tidlist1, <p, tidlist1>∈ rmo(rst', D)};
(19)          if (|rst'.tidList|≥ minSupp)
(20)               Fk+1← rst';
(21)          remove rmo(rst, D);
(22)          remove rst.tidlist;
(23) return Fk+1;
```

Fig.7 Utilization of rightmost occurrences

Figure 8 shows the algorithm *GetCondRMO*, which are used for the computation of the rightmost occurrences of those RSTs generated through join operation (line 14-15 of Fig.7). *GetCondRMO* obtains the conditional rightmost occurrences of an RST in the G-QPT. *GetCondRMO* uses post-order enumeration of the nodes in RST. The main loop visits RST nodes in descending order (see line 2 of Fig.8). Assume q is the current RST node. The algorithm fetches all G-QPT nodes with the same label value as q (line 4). For each fetched node d of the G-QPT, the algorithm try to match subtree of RST rooted at q against subtree of the G-QPT rooted at d. Since subtrees of RST rooted at the child nodes of q have already been matched in preceding loops, the algorithm need only to check whether for each c-child node (or d-child node respectively) q' of q, there is a c-child (or descendant) d' of d, such that q' and d' are matched in preceding loops (line 6-13). RST is matched with the G-QPT if the root of RST is matched with the root of the G-QPT at the ending point (line 25). For each node d of the G-QPT that is matched with an ancestor node q of the rightmost leaf node (or junction node respectively) of RST, the ancestor-descendant relationship between d and d' is recorded (line 16-23), where d' is a node of the G-QPT that will be matched with the rightmost leaf node (or junction node respectively) of RST when d is matched with q. The ancestor-descendant relationship between d and d' will at last be used to obtain conditional rightmost occurrence (line 25-26).

5 Experiments

In this section, we present experimental results to evaluate the effectiveness of our algorithm on a range of datasets and parameters. All experiments were performed on a 1.8GHz PC with 512MB RAM, running Windows 2000 Server. The algorithms were implemented in C++.

```
Algorithm: GetCondRMO(RST, G-QPT, c, rmo, cond-rmo)
Input: RST: a rooted subtree
G-QPT: the global pattern tree
c: the junction node of RST
       rmo: the rightmost occurrence of RST in G-QPT
       cond-rmo: the conditional rightmost occurrence (used for returning result)
Output: the conditional rightmost occurrence
rml.match = rmo; //rml is the rightmost leaf of RST
for all other q ∈ RST //iterate all nodes of RST except for rml in postorder
       q.match = {};
       for all d ∈ G-QPT such that d.label = q.label
            new map_d;
            for all q' ∈ children(q)
                 map_d.q' = {};
                 if q' is a d-child
                      for all d' ∈ q'.match such that d is an ancestor of d'
                           map_d.q' = map_d.q' ∪ {d'};
                 if q' is a c-child
                      for all d' ∈ q'.match such that d' is a c-child of d
                           map_d.q' = map_d.q' ∪ {d'};
            if not exist p such that map_d.p = {}
                 q.match = q.match ∪ {d};
                 if q is parent of the rightmost leaf
                      d.rmo = map_d.rml;
                 if q is c or a descendant of c
                      and an ancestor of the rightmost leaf except for its parent
                      d.rmo = {p | p ∈ q.rmo, q ∈ map_d.rmc };
                 if q is parent of c
                      d.cond = map_d.rmc;
                 if q is an ancestor of c except for its parent
                      d.cond = {m | m ∈ n.cond, n ∈ map_d.rmc };
            delete map_d;
            if q is the root of RST and d is the root of G-QPT
                 cond-rmo = cond-rmo ∪ {<m, n>| m ∈ d.cond, n ∈ m.rmo };
return;
```

Fig.8 The algorithm to obtain the conditional rightmost occurrence

We use DBLP.dtd as the schemas of XML data sources. Given a DTD, we first construct a G-QPT by introducing duplicate siblings, descendant edges, and wildcards. Then we generate all RSTs of the G-QPT, and use Zipf and uniform distribution of RSTs to produce the transaction file of QPTs. Zipf distribution is used since Web queries and surfing conform to Zipf's law[3], while uniform distribution is used only to see what differences might happen to our approaches with different QPT distribution. The characteristics of each dataset are listed in Table 1. A RST with descendant edges takes more time to compare with QPTs than a RST without descendant edges. Consequently, the number of descendant edges in G-QPT tells the difficulty of tree matching. On the other hand, the average number of nodes, maximum depth and fanout of QPTs also show the complexity of the dataset.

Table 1 Properties of datasets

Datasets	G-QPT				QPT in DB			DB size of 100K dataset (KB)
	#of nodes	Max depth	#of d-child	Max fanout	Avg # of nodes	Max depth	Max fanout	
Zipf	98	8	24	12	7.4	8	12	3549
Uniform	98	8	24	12	9.2	8	12	4843

Two groups of experiments are performed. The first group of experiments is to show the performance of our algorithms at a range of different minimum support values. We use 100K datasets, which contain 100,000 QPTs. The second group aims to present the scale-up of our algorithms at minimum support value 1%. The total number of QPTs in each datasets ranges from 50K, 100K, 200K, 300K to 500K.

The performance of *FQPMiner* and *FQPMinerRMO* is shown in and. For 100K data set of Zipf distribution, *FQPMinerRMO* is 5-7 times faster than *FQPMiner*. The reason is that *FQPMiner* will match each candidate RST against QPTs in the transaction database while *FQPMinerRMO* processes the majority of candidate RSTs without matching operation. In addition, matching time of single-branch RSTs is much less than that of multi-branch RSTs. Consequently, *FQPMinerRMO* takes much less time than *FQPMiner*. This fact is further confirmed in the data set of uniform distribution, where *FQPMinerRMO* can be 8-12 times faster than *FQPMiner*.

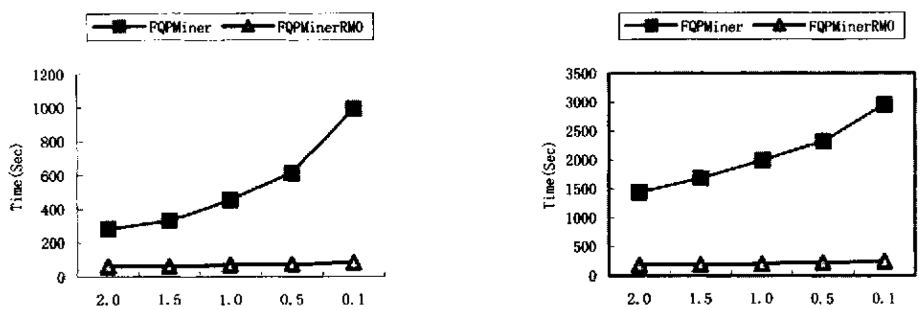


Fig.9 Experiment results

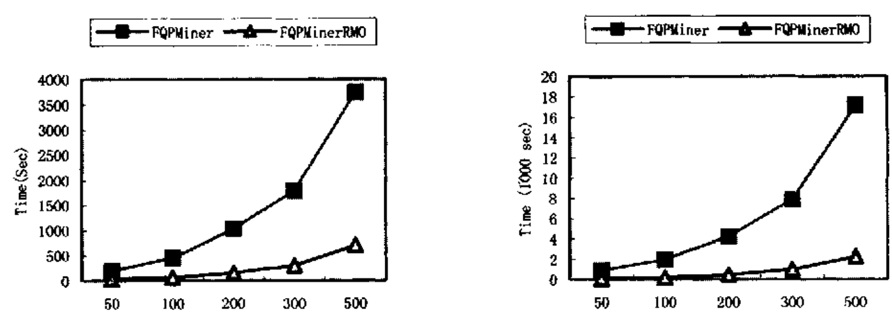


Fig.10 Experiment results

The result of scale-up experiment is shown in. At a given minimum support value 1%, we performed the experiments by varying the number of transactions(QPTs) in database file DBLP(Zipf) from 50,000 to 500,000. From the, we find that the execution time of both *FQPMinerRMO* and *FQPMiner* algorithms scales almost linearly with the size of dataset. Meanwhile, we also find that *FQPMinerRMO* continues to be about 5-7 times faster than *FQPMiner*. This fact is further confirmed in the data set of uniform distribution (see), where *FQPMinerRMO* continue to be 8-10 times faster than *FQPMiner*.

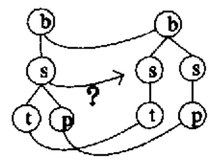
6 Related Work

Data mining is an important research area in the field of knowledge engineering. Recently researchers gradually shift their focus to mining frequent structures like graphs[6,9] or trees[8,12,1,10,11]. Different from our work, most previous works assume that transaction database is consisted of simple trees or graphs without descendant edges or wildcards, and their methods are not directly applicable to our applications since they can't handle special

XPath constructs, which would make the computation much more complex.

Reference [12] is more closely related with our works. In its settings, each edge works as a descendant edge. It maintains prefix matches for each candidate pattern, and proposed a method to compute the prefix matches of k+1-edge patterns from prefix matches of k-edge patterns incrementally. Since the support counts of candidate patterns can be derived their prefix-matches, Ref.[12] can get the support counts of candidate patterns without matching them against transactions. However, under special cases, the memory cost of prefix matches of a candidate could be exponential to its size. For instance, given a transaction tree DT composed of a chain of n nodes and a pattern tree PT composed of a chain of m+1 nodes, where each node is labeled with the same label l, Ref.[12] has to maintain $n!/((n-m)!*m!)$ prefix matches for the match between PT and DT. Clearly, the algorithm of Ref.[12] is intractable since they have to generate and compare each prefix match.

The discovering of frequent XML query patterns was initially proposed in Refs.[10,11]. In Refs.[10,11], several efficient algorithms are proposed through utilization of tidLists. However, Refs.[10,11] utilize tidLists based on the following assumption: if all proper subtrees of a multi-branch pattern tree occur in a data tree, then the pattern tree itself must also occur in the data tree. The above assumption may lead to imprecise mining results because based on the semantics of XML queries, it doesn't hold under certain special cases. For example, although each proper rooted subtree of RST_2 in Fig.11 is subsumed in QPT_5, RST_2 itself is not.



(a) RST_2 (b) QPT_4

Fig.11 Inpreciseness

7 Conclusions

In this paper, we present an efficient algorithm to discover frequent rooted subtrees from XML queries through utilization of rightmost occurrences. Our method is strictly based on the semantics of XML queries, and we believe this is very important for effective utilization of the mining results.

Future work includes incremental computation of frequent RSTs. To incorporate the result of this paper into caching system, an important issue is to guarantee the freshness of the materialized data. The caching system must guarantee the consistence of the mining result with the history database D. However, if the pattern of user activity changes at a relatively high rate, the accuracy of the mining result may deteriorate fast. Because re-computation will incur a high overhead, finding a method to discover frequent RSTs incrementally becomes very important.

References:

[1] Asai T, Abe K, Kawasoe S, *et al.* Efficient substructure discovery from large semistructured data. In: Proc. of the 2nd SIAM Int'l Conf. on Data Mining (SDM2002). 2002.

[2] Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proc. of the 20th Int'l Conf. on Very Large Databases (VLDB). 1994. 487~499.

[3] Breslau L, Cao P, Fan L, *et al.* Web caching and zipf-like distributions: Evidence and implications. In Proc. of the IEEE INFOCOM. 1999. 126~134.

[4] Clark J, DeRose S. XML Path Language (XPath) version 1.0 w3c recommendation. Technical Report REGxpath-19991116, World Wide Web Consortium, 1999.

[5] Hristidis V, Petropoulos M. Semantic Caching of XML Databases. WebDB, 2002

[6] Kuramochi M, Karypis G. Frequent subgraph discovery. In: Proc. of the ICDM'01. 2001. 313~320.

[7] Miklau G, Suciu D. Containment and equivalence for an XPath Fragment. PODS 2002 65~76.

[8] Wang K, Liu HQ. Discovering typical structures of documents: A road map approach. SIGIR, 1998.

[9] Yan X, Han J. CloseGraph: Mining closed frequent graph patterns. SIGKDD 2003. 2003.

[10] Yang LH, Lee ML, Hsu W, Acharya S. Mining frequent query patterns from XML queries. DASFAA, 2003. 2003. 355~362.

[11] Yang LH, Lee ML, Hsu W. Efficient mining of frequent query patterns for caching. In: Proc. of the 29th VLDB Conference, 2003. 2003.

[12] Zaki M. Efficiently mining frequent trees in a forest. ACM SIGKDD, 2002.