Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Bottom-up mining of XML query patterns to improve XML querying*

Yi-jun BEI†, Gang CHEN, Jin-xiang DONG, Ke CHEN

(*School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*)

†E-mail: alphabyj@yahoo.com.cn

Received Oct. 16, 2007; revision accepted Jan. 28, 2008; published online May 9, 2008

Abstract: Querying XML data is a computationally expensive process due to the complex nature of both the XML data and the XML queries. In this paper we propose an approach to expedite XML query processing by caching the results of frequent queries. We discover frequent query patterns from user-issued queries using an efficient bottom-up mining approach called VBUXMiner. VBUXMiner consists of two main steps. First, all queries are merged into a summary structure named "compressed global tree guide" (CGTG). Second, a bottom-up traversal scheme based on the CGTG is employed to generate frequent query patterns. We use the frequent query patterns in a cache mechanism to improve the XML query performance. Experimental results show that our proposed mining approach outperforms the previous mining algorithms for XML queries, such as XQPMinerTID and FastXMiner, and that by caching the results of frequent query patterns, XML query performance can be dramatically improved.

Key words: XML querying, XML mining, Caching, Data mining
doi:10.1631/jzus.A071551 **Document code:** A **CLC number:** TP311.13

INTRODUCTION

In recent years, XML data have become ubiquitous with the rapid increment in both the number and scale of applications such as XML database systems, business transactions, XML middleware systems, and so on. Efficient querying techniques of XML data have become an important topic for the database community. Building an index of XML data (Al-Khalifa *et al*., 2002; Kim *et al*., 2006; Seo *et al*., 2007) has been regarded as an effective way to accelerate retrieval of XML data. However, an indexing mechanism may have certain drawbacks: first, it is often neither possible nor desirable to maintain an entire index in the primary storage; secondly, an indexing mechanism may still incur unnecessary computation for repeated or similar queries. To address these problems, caching techniques have been considered to improve the performance of XML query processing. When a query cache is deployed, users can obtain the answer right away if the query result has already been computed and cached. To date, there have been several reports on the caching techniques of XML queries (Chen *et al*., 2002; 2005; Yang *et al*., 2003a; Hong and Kang, 2005).

To cache the results of useful queries, one of the most effective approaches is to discover frequent query patterns from the user queries, as the frequent query patterns usually contain a wealth of information about user queries. Basically, if we model each XML query as a tree, just like some of the previous works (Yang *et al*., 2003a; Paik and Kim, 2006; Gu *et al*., 2007), the frequent query pattern mining problem is converted to the problem of finding a set of subtrees that occur frequently over a set of trees (or queries).

For the sake of efficiency in discovering frequent query subtrees, we present an algorithm called VBUXMiner which employs a bottom-up enumerating method and prunes unsatisfied patterns as early as

* Project supported by the National Natural Science Foundation of China (No. 60603044), the National Key Technologies Supporting Program of China during the 11th Five-Year Plan Period (No. 2006BAH02A03), and the Program for Changjiang Scholars and Innovative Research Team in University of China (No. IRT0652)

possible. We introduce a novel data structure called the "compressed global tree guide" (CGTG) to accelerate candidate generation and infrequent subtree pruning. We firstly remove all infrequent nodes from the global tree guide before candidate enumeration, and then generate candidates within each prefix equivalence class. Unlike the previous algorithms such as XQPMiner, XQPMinerTID and FastXMiner (Yang *et al*., 2003a; 2003b), which employ a rightmost branch expansion enumeration approach to generate candidates from top to bottom, we perform an efficient bottom-up candidate generation process instead. Moreover, when computing the support of each subtree, previous methods have no guarantee of the minimum number of scans on the data set, whereas in our approach we no longer need to scan the data set, as the support of the subtrees can be easily computed from the CGTG. Like in the mining algorithms XQPMiner, XQPMinerTID and FastXMiner, we do not consider XML queries that contain sibling repetitions either. Experimental results on public data sets show that our proposed mining algorithm is more efficient compared to previous works. The experiments also indicate that the caching mechanism exploiting the mining results is effective in improving the query response time, and it outperforms the traditional LRU (least recently used) and MRU (most recently used) caching policies significantly.

The rest of the paper is organized as follows. In Section 2 we discuss previous works related to query pattern mining approaches and XML caching. In Section 3, we discuss some concepts used in our mining approach. We propose the bottom-up XML subtree mining algorithm VBUXMiner in Section 4. In Section 5 we show how our mining approach is applied to cache the results of XML queries. Section 6 presents the results of the experiments and Section 7 concludes the paper.

RELATED WORKS

In this section, we shall review some related works, including papers on tree mining, XML query mining, and caching techniques for XML queries.

Recently, tree-like structure mining has attracted a lot of attention. Basically, there are two main steps for generating frequent trees. First of all, a systematic method should be conceived for generating non-redundant candidate trees. Secondly, an efficient method is needed to compute the support of each candidate tree and determine whether a tree is frequent. In any case, a straightforward generate-and-test strategy is adopted. Various algorithms have been provided to mine different forms of tree structures such as rooted ordered tree, rooted unordered tree, free tree, etc. Asai *et al.*(2002; 2003) present the rooted ordered and rooted unordered tree mining approaches. Zaki (2002; 2005) gives ordered and unordered embedded tree mining algorithms. Chi *et al.* (2003; 2004) bring forward approaches to rooted unordered and free trees mining. Unlike previous approaches, Chehreghani *et al.*(2007) do not employ the apriori-based approach, but present a top-down approach for mining all maximal, labeled, unordered, and embedded subtrees from a tree-structured database. However, these mining approaches mainly deal with general trees. They do not take schema information into consideration when dealing with special trees like XML query pattern trees.

Many approaches have been proposed to discover information from XML documents or XML queries (Yang *et al*., 2003a; 2003b; Paik and Kim, 2006; Nayak and Iryadi, 2006; Gu *et al.*, 2007; Bei *et al.*, 2007; Kutty *et al.*, 2007). The closest works to ours, which find frequent rooted query patterns from a set of XML queries, are the algorithms XQPMiner, XQPMinerTID, FastXMiner presented in (Yang *et al*., 2003a; 2003b). XQPMiner and XQPMinerTID exploit schema information to guide the enumeration of candidates and employ a rightmost branch expansion enumeration to generate candidates from top to bottom. When computing the frequency of candidates, XQPMiner scans the data set for each candidate, while XQPMinerTID scans the data set only when expansion happens on the leaf node. As a result, XQPMinerTID outperforms XQPMiner due to fewer data set scans. FastXMiner also generates candidate trees with the help of schema information. However, it is more efficient since it needs data set scans only when the candidate tree is a single branch tree. The support of a non-single branch tree can be computed by joining the ID lists of its proper rooted subtrees. Bei *et al.*(2007) develope the algorithm BUXMiner to mine rooted XML query patterns using a bottom-up approach. BUXMiner enumerates candidate trees

from bottom to top based on a compact global tree guide. All infrequent nodes are pruned to accelerate tree enumeration. The support of a candidate tree is computed without scanning the database as it is calculated directly from the global tree guide.

Caching results of XML queries has been considered a useful strategy to improve performance of XML query processing. XCache (Chen *et al*., 2002) is a holistic XQuery-based semantic caching system. Mining approaches for finding frequent queries are also incorporated into caching (Yang *et al*., 2003a; Chen *et al*., 2005). Yang *et al*.(2003a) employ FastXMiner to discover frequent XML query patterns and demonstrate how the frequent patterns can be used to improve caching performance. Chen *et al*. (2005) take into account temporal features of queries for frequent queries discovery and design an appropriate cache replacement strategy by finding both positive and negative association rules. Hong and Kang (2005) integrate heterogeneous data sources on the Web and cache results of queries through XML views of data sources to accelerate query processing.

PRELIMINARY CONCEPTS

Frequent rooted query pattern tree

Definition 1 (Query pattern tree, QPT) An XML query can be modeled as a query pattern tree $QPT=<R, N, E>$, where R is the root node, N is the node set, and E is the edge set. Each node n has a label whose value is in {"*", "//"}$\cup$*labelSet* where the *labelSet* is the label set of all elements and attributes. For each edge $e=(n_1, n_2)$, node n_1 is the parent of n_2.

Definition 2 (Query pattern subtree, QPS) Given two query pattern trees T and S, S is considered to be a query pattern subtree of T iff there exists a one-to-one mapping φ: $V_S \rightarrow V_T$ satisfying the following conditions: (1) φ preserves the labels, i.e., $L(v)=L(\varphi(v))$ $\forall v \in V_S$; (2) φ preserves the parent relation, i.e., $(u,v)\in E_S$ iff $(\varphi(u), \varphi(v))\in E_T$.

Definition 3 (Rooted query pattern subtree, RQPS) Given two query pattern trees T and S, we say that S is a rooted query pattern subtree of T iff S is a query pattern subtree of T and the trees S and T have the same root label.

Definition 4 (Query database tree, QDT) An XML query database, which is a collection of XML queries, can be represented as $QDT=<T, R, Q, \Phi>$, where T is a tree whose root is R; Q is the set of query pattern trees $\{q_1, q_2, \ldots, q_n\}$; R is the virtual root node of the tree with a special label not belonging to *labelSet*; Φ: $V \rightarrow Q$ is a query mapping function from all children of the root R to the trees Q, where V represents the set of all children of the root R. For a complete tree with the root node being the ith node of $v_i \in V$, we have $\Phi(v_i)=q_i$.

Definition 5 (Frequent rooted query pattern tree, FRQPT) Let D denote all the query pattern trees of the issued queries and d_T be an indicator variable with $d_T(S)=1$ if the query pattern tree S is a rooted query pattern subtree of T and $d_T(S)=0$ if tree S is not. The support of query pattern tree S in D can be defined as $\sigma(S)=\sum_{T\in D} d_T(S)/\sum_{T\in D}$, i.e., the percentage of the number of trees in D that contain tree S. A rooted query pattern tree is frequent if its support is more than, or equal to, a user-specified minimum support, defined as *minsupp*.

With the help of the QDT, we can transform the problem of discovering FRQPTs from the original query database into the problem of discovering FRQPTs over the QDT. Let $n_T(S)$ denote the number of occurrences of the rooted subtree S in a tree T. Then the support of a rooted query pattern tree S can be defined as $\sigma(S)=n_{\mathrm{QDT}}(S)/|Q|$. In this way, we can deal with query pattern trees with different root nodes, and discover frequent query pattern trees while only considering the rooted query pattern subtrees. After finding all the FRQPTs over the QDT, the frequent query patterns are obtained by simply removing the virtual root of each FRQPT. Fig.1 shows a query database tree composed of five XML queries. Given the minimum support 0.6, we can obtain six FRQPTs.

Compressed global tree guide

For each user issued QPT, we assign a unique ID, denoted as QPT.ID, which will be used for the construction of a global tree guide in the mining process.

Definition 6 (Global tree guide, GTG) We merge all issued queries over the query database tree to create a global tree guide, where the ID list of each node represents the queries containing the path from the root to the current node. Fig.2 shows a GTG constructed using 15 query pattern trees. The QPT list for node "Java" indicates that there are six queries that contain the path "R/order/items/book/title/Java".

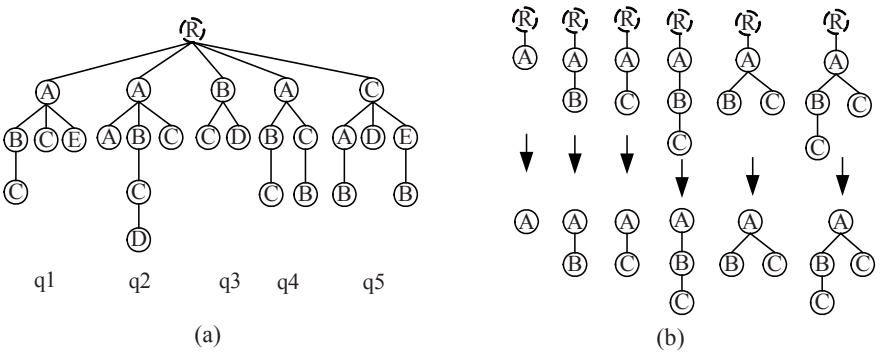

Fig.1 (a) Query database tree (QDT); (b) Frequent rooted query pattern tree (FRQPT)

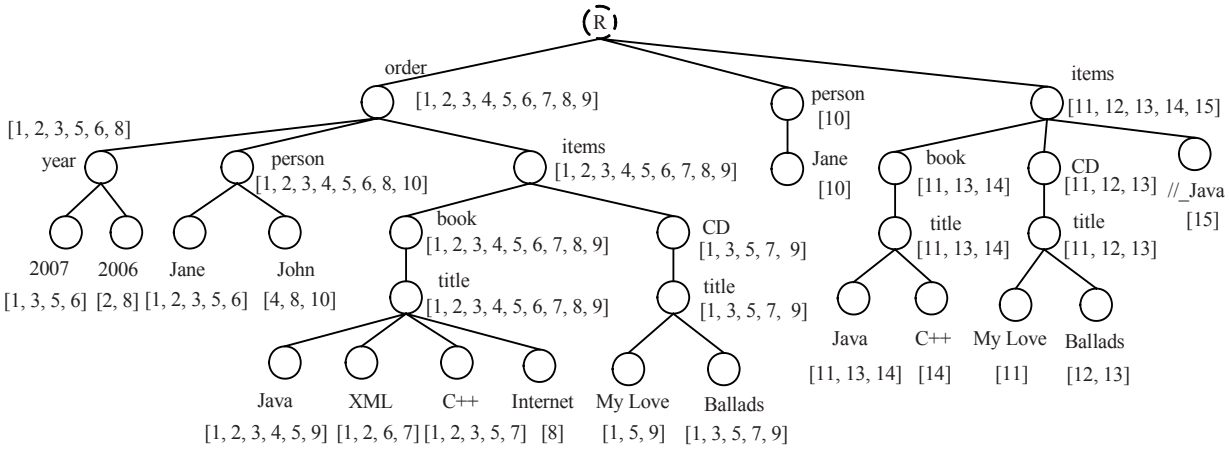

Fig.2 Global tree guide constructed using 15 query pattern trees from the orders

To handle labels like wildcard "*" and descendant path "//", we combine the special label and the following label to produce a new label. For example, a single path tree "R/items//_Java" in the GTG will be considered a single path tree with nodes "R", "items" and "//_Java".

For simplicity, we denote a subtree rooted at the root node of the GTG as RT, and a single path starting at the root node as SRT.

Definition 7 (Frequent node) The support of the node in the GTG is defined as the ratio of the number of QPTs that contain the path from the root to the current node, namely the size of QPT list, to the number of all QPTs. For example, the support of node "Java" is 3/15=0.2. A node in the GTG is frequent if its support is no less than the minimum support.

Lemma 1 The support of the node is no less than the support of its descendant node.

Proof A descendant node can be reached only through its ancestor in a QPT. If a QPT contains the path from the root to the descendant node, then it must contain the path from the root to the ancestor node. Therefore, the support of the ancestor is no less than that of the descendant.

Lemma 2 If a node is infrequent in the GTG, then an RT including it will not be a frequent rooted tree.

Proof Since the support of an RT will be no more than the support of a node in the RT, an RT will be infrequent if an included node is infrequent.

Lemma 3 If a node is frequent in the GTG, an SRT including it as the leaf node must be a frequent tree.

Proof As the support of an SRT equals the support of the leaf node, an SRT will be frequent if the node is frequent.

Assume the minimum support is 0.2. In Fig.2 the node "Internet" is infrequent, and the RT "R/order/items/book/title/Internet" is also infrequent. The node "XML" is a frequent node, and thus the SRT "R/order/items/book/title/XML" is also frequent.

If a node is infrequent, then all its descendant nodes are infrequent as well, due to the lesser support of the descendant nodes. As a result, we prune all the

infrequent nodes in the GTG before candidate enumeration, using a top-down traversal. We traverse the GTG starting at the root level by level and prune infrequent nodes along with all its descendants once we find an infrequent node. For instance, in Fig.2, the second child node of the root as well as its descendants is pruned, since it is an infrequent node.

Furthermore, to save memory space, the node and its child node are compressed into a single node with the following satisfaction: (1) the parent node has only one child; (2) the parent node and the child node have the same ID list of QPTs. For example, in Fig.2 we compress the node "book" and its child node "title" into a single node "book/title".

Definition 8 (Compressed global tree guide, CGTG) Employing the infrequent node pruning scheme and node compressing scheme, we reduce the GTG into a CGTG. Fig.3 presents a CGTG transformed from the GTG in Fig.2 with the minimum support 0.2.

Lemma 4 If a tree S in the CGTG is frequent, then the tree constructed by adding the parent node of the root of the tree S is also frequent.

Proof Suppose the node n is the parent of the root of the frequent tree S. Then n must be the ancestor of all the nodes in the tree S. Thus any query pattern tree containing the nodes in tree S must also contain the node n according to our construction of the CGTG. Because the new tree only has one direct subtree, the support of the new tree generated from its direct subtree S by adding the new root n is equal to the support of tree S. Thus the new tree is a frequent tree due to the frequency of S.

QUERY PATTERN TREE MINING

In this section, we present a bottom-up mining algorithm VBUXMiner for discovering frequent rooted query pattern trees from user queries.

Overview of VBUXMiner

VBUXMiner performs a bottom-up process to generate frequent rooted query pattern trees over the CGTG. To generate frequent query pattern trees rooted at node n in the CGTG, we will have to generate all frequent query pattern trees rooted firstly at the children of n, and then merge these frequent trees. Algorithm 1 shows the high level structure of VBUXMiner. We first scan all query pattern trees to construct a GTG and create a CGTG based on the GTG by means of pruning and compression. And then we use the root node of the CGTG as an input to recursive generation of frequent rooted query patterns from bottom to top over the CGTG. Finally, we remove the virtual node of the discovered frequent pattern trees to obtain the final result.

Algorithm 1 *VBUXMiner*(*D*, *minsupp*)

Input: A set of query pattern trees; specified minimum support

Output: A set of frequent query pattern trees *FRTS*

```
1  GTG=ConstructGTG(D);
2  CGTG=CompressGTG(GTG, minsupp);
3  root=root node of CGTG;
4  FRTS=GenerateFrequentRT(root, minsupp);
5  Remove virtual roots of each discovered tree in FRTS;
6  return FRTS
```

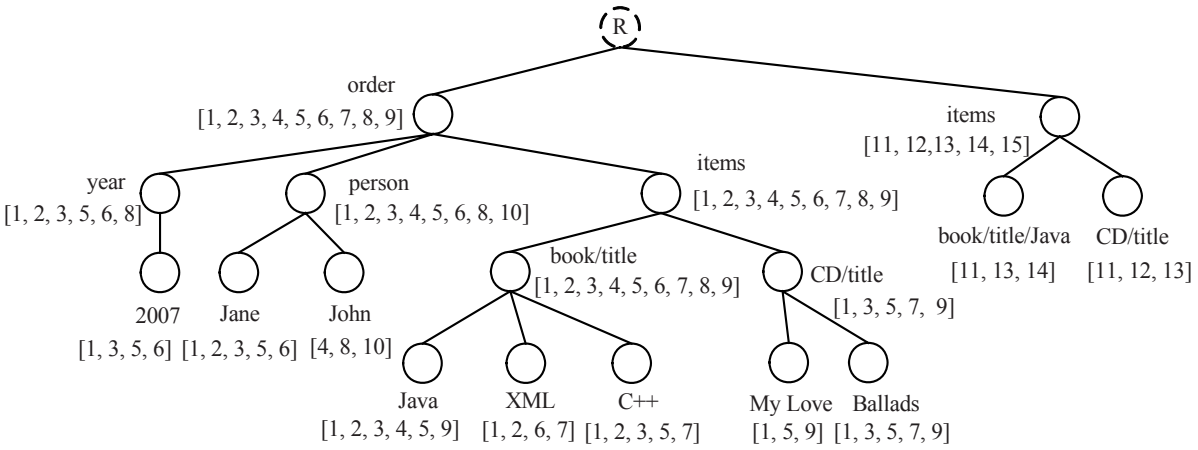

Fig.3 The global tree guide in Fig.2 is compressed employing the infrequent node pruning scheme and the node compressing scheme

CGTG construction

Construction of the CGTG consists of two steps. We first scan the issued queries and generate a GTG, and then apply the node pruning and node compression to the previously generated GTG. When scanning queries, a bread-first traversal is adopted on each query. We traverse each query level by level from top down. For each visited node, we check to see if a node exists in the GTG with the same path as the currently visited node. If one exists, we simply add the ID of the query into the node in the GTG. Otherwise, we append a new child to the node in the GTG which has the same path to the parent node of the visited node, and we add the query ID into the new node. To prune infrequent nodes and compress nodes, we adopt a depth-first traversal on the GTG. For each node in the GTG, we check if it is frequent according to its ID list. If it is not, then we will remove the current node as well as all its descendant nodes. Otherwise we will traverse down to check its child nodes. A node without sibling that has the same ID list as its parent node will be compressed by appending its label to its parent node.

Frequent query pattern tree generation

Definition 9 (Query pattern tree encoding) A string encoding scheme introduced by Luccio *et al*.(2001) is adopted to represent query pattern trees, which is more space-efficient and is simpler to manipulate (Zaki, 2002). The string encoding of a query pattern tree is obtained by traversing the tree in a depth-first order. Following the order of traversal, we record in the string the label for each node. Whenever a backtracking occurs from a child to its parent, a distinguished label (−1 is used here) is appended to the string. For example, the tree *CPT* in Fig.4 can be encoded as a string "items, book, title, Java, −1, XML, −1, −1, −1, −1".

Definition 10 (Prefix equivalence class) We say that a number of query pattern trees are in the same prefix equivalence class, if they share a common prefix tree in the CGTG. Formally, let X, Y be two query pattern trees, and let function $p(X, k)$ return the prefix tree up to the kth node. Then X, Y are in the prefix equivalence class iff there exists a k such that $p(X, k)=p(Y, k)$. For example, trees QPT_1, QPT_2, QPT_3 in Fig.4 have the same prefix tree *CPT*. We say they are in a same equivalence class with the prefix tree *CPT*. Using the previous tree encoding scheme, we can obtain any two members of an equivalence class having the same prefix string which represents the prefix tree. If the prefix tree is represented as a string "Labels, −1", then trees in the equivalence class must have the prefix "Labels". By employing tree encoding, we can easily determine whether query pattern trees are in the same equivalence class. As QPT_1, QPT_2, QPT_3 have the same prefix string encoding "items, book, title, Java, −1, XML, −1, −1, −1", they belong to the same equivalence class.

Frequent query pattern trees rooted at a given node are composed of three parts: (1) the root node itself; (2) the root node appended by the frequent query pattern trees rooted at its child nodes; (3) frequent query pattern trees generated by merging the frequent ones in part (2). In Algorithm 2 we show the algorithm for generating all the three parts of the frequent rooted trees at a given root node over the CGTG.

First of all, we consider the tree with only the root node as a frequent query pattern tree (Lines 1~2). This is because we perform the searching process in a CGTG which has pruned all infrequent nodes.

We generate all frequent rooted query pattern trees at the children (Lines 8~14). According to Lemma 4, the trees are frequent which are generated

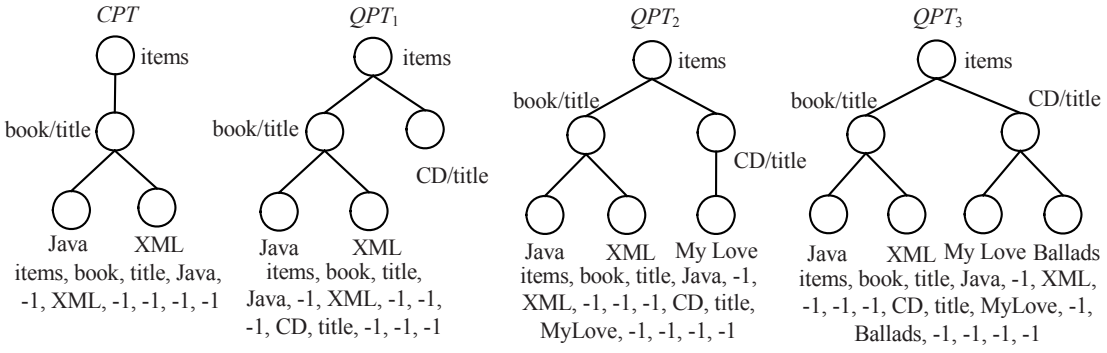

Fig.4 Three query pattern trees having the common prefix tree are in the same equivalence class

Algorithm 2 *GenerateFrequentRT*(*root*, *minsupp*)

Input: Root node of new generated frequent tree root; specified minimum support
Output: A set of frequent query pattern trees *FRTS* rooted at the root node

1 $FRT_{root}.label=(root.label, -1)$;
$FRT_{root}.IDList=root.IDList$;
2 $FRTS=\{FRT_{root}\}$;
3 $EQ=\varnothing$;
4 for (each child of root) do
5 $FRTS_{child}=GenerateFrequentRT(child, minsupp)$;
6 $EQ=EQ\cup\{FRTS_{child}\}$;
7 for (each *ChildSet* in $FRTS_{child}$) do
8 for (each FRT_{child} in *ChildSet*) do
9 $NewFRT.label=(root.label, FRT_{child}.label, -1)$;
10 $NewFRT.IDList=FRT_{child}.IDList$;
11 $FRTS=FRTS\cup\{NewFRT\}$;
12 end for
13 end for
14 end for
15 $MergeList=\{EQ\}$;
16 while ($|MergeList|>0$) do
17 $EQ=MergeList[1]$;
18 $MergeList=MergeList-EQ$;
19 for (i=1 to $|EQ|-1$) do
20 for ($j=i$ to $|EQ|$) do
21 $NewEQ=EQ_i\oplus EQ_j$;
22 $FRTS=FRTS\cup NewEQ$;
23 $MergeList=MergeList\cup\{NewEQ\}$;
24 end for
25 end for
26 end while
27 return *FRTS*

by adding the parent node to the frequent query pattern trees rooted at its child nodes. In this way, the frequent query pattern trees of part (2) can be obtained. Frequent query pattern trees rooted at each child are regarded as being in the same equivalence class because they share the same prefix tree which is the child node itself. Then by adding the root node, we obtain new equivalence classes, with each prefix tree being a combined tree of the root and its child.

Finally, we employ an equivalence class joining strategy on the frequent trees generated at part (2) (Lines 15~26). We join frequent rooted trees from different equivalence classes to construct new candidates and determine whether they are frequent. Assume the root node has n children and there exist n equivalence classes from EQ_1 to EQ_n. We pick up a frequent query pattern tree FRT from EQ_i and join to it all trees from EQ_{i+1} to EQ_n. Through the joining of FRT and all trees in the equivalence class EQ, we generate a new equivalence class whose common prefix tree is a combined tree of FRT and prefix tree of EQ. After the joining of all frequent trees from $FRTS_{i+1}$ to $FRTS_n$, we generate $(n-i)$ equivalence classes. Then we regard the $(n-i)$ equivalence classes as a new group and perform a next equivalence class joining process. This process is repeated until there remains only one equivalence class.

For example, consider the node labeled "items" in Fig.3. As it has two frequent child nodes, there exist two equivalence classes for each child. If we pick up a frequent tree from the equivalence class of the first child, and join it with the second equivalence class, then we will obtain a new equivalence class. In Fig.5, we show frequent subtrees from two equivalence classes of its children and illustrate the joining process. The leftmost tree is a frequent tree picked up from the first equivalence class, while the trees in the middle are all frequent trees in the second equivalence class. Then by joining the above trees, we can obtain four new trees in the same equivalence class with the prefix tree combined with the left tree and the prefix tree of the second equivalence class. As shown in Fig.5, the tree in the virtual box is the prefix tree of the new equivalence class. However, among all the newly generated trees only two trees are frequent according to the definition of a frequent query pattern tree. Because there is only one equivalence class through joining the leftmost frequent tree with all the other equivalence classes, we finish the equivalence class joining process for the current tree.

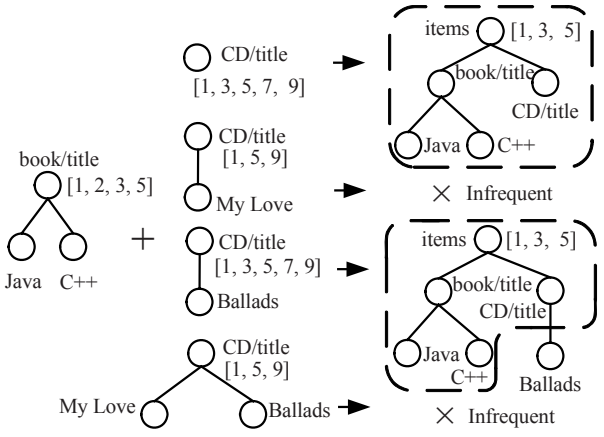

Fig.5 A query pattern tree is joined with all the trees in another equivalence class

Definition 11 (Tree joining) Given a prefix tree T_1, a suffix tree T_2, and a common prefix tree CT of T_1 and T_2, we join the two trees T_1 and T_2 and produce a new tree with prefix T_1. We denote the joining process as $T=T_1 \cup_{CT} T_2$. The ID list of the created tree is the result of joining two ID lists of the trees. Suppose the CT is represented as the string "*CT_Labels*, −1". Then we can denote T_1 as "*CT_Labels*, T_1*_Follow_Labels*, −1" and T_2 as "*CT_Labels*, T_2*_Follow_Labels*, −1". The constructed tree is represented as "*CT_Labels*, T_1*_Follow_Labels*, T_2*_Follow_Labels*, −1". In Fig.6 we show the tree joining process, where the *CT_Labels* is the string "order, year, 2007, −1, −1", T_1*_Follow_Labels* is "person, Jane, −1, −1", and T_2*_Follow_Labels* is "items, book, title, Java, −1, C++, −1, −1, −1, −1".

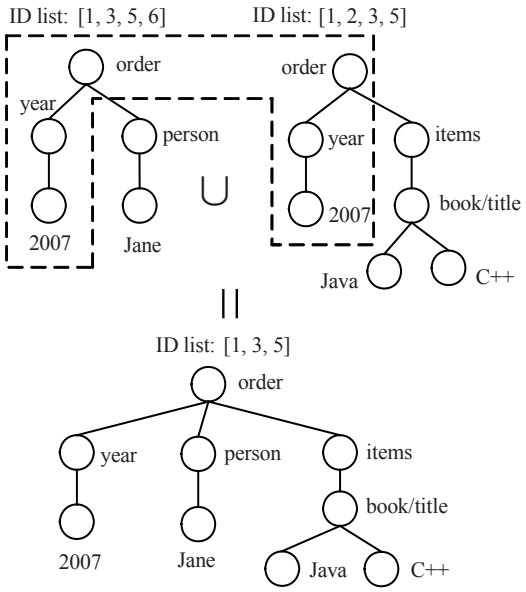

Fig.6 Two trees with the same prefix tree are joined

Definition 12 (Frequent decision) In order to decide whether a new candidate tree is a frequent one, we will have to calculate its support. However, to avoid unnecessary computing, a pruning process is performed before support computing. We prune the new candidate k-size tree if it has infrequent $(k-1)$-size rooted subtrees. Once the candidate passes the pruning process, we then compute the support of the candidate tree using the ID list of QPTs. However, the ID list of the new tree still can be computed quickly by way of a merging based on the ascending order of each recording ID list.

Definition 13 (Automatic ordering) If trees in an equivalence class set EQ are ordered according to the node order in CGTG, then trees in a new equivalence class set NEQ, constructed by means of merging a prefix tree with all trees in EQ, are still ordered in the node order. This is because the tree merging process does not change the node label order, and only appends a new prefix tree to all suffix trees. The new frequent trees are inserted into the new equivalence class according to the original order, which results in an automatic ordering of the trees in each equivalence class.

Definition 14 (Candidate pruning) As previously described, before computing the support of a k-size candidate tree, we carry out a pruning test to make sure all its rooted subtrees are frequent. To improve the mining efficiency, we only check whether its $(k-1)$-subtrees are frequent. According to our candidate generation method, we ensure that all $(k-1)$-subtrees have been enumerated before dealing with the current tree. To perform the pruning step efficiently, we add each frequent tree into a hash table during the creation of frequent trees. The key of each entry in the hash table is the string representation of the tree. Thus it takes $O(1)$ time to check for each $(k-1)$-subtree.

Definition 15 (Space reducing) The main space consumption is incurred by the ID list of QPT for each frequent rooted tree. If a parent node has been computed, then all frequent trees rooted at the child nodes can be removed. In this way, the space consumption of our algorithm is the whole CGTG plus the ID list for frequent query pattern trees rooted at the current node and its children.

QUERY PATTERN TREE CACHING

In this section, we present the technique of applying the frequent query pattern trees into XML query caching. We mainly describe the query pattern tree rewriting scheme for similar queries and the cache replacement strategy. In Fig.7 we show the framework of the frequent queries caching system, which uses the proposed frequent query discovery approach to accelerate the querying process.

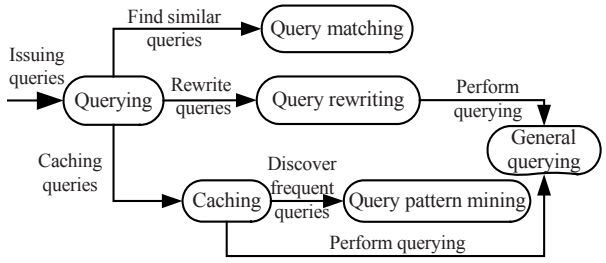

Fig.7 Framework of the XML query caching system

Query pattern tree rewriting

In most cases, queries issued by users are rarely identical. However, there are usually a lot of similarities in the issued queries. To take advantage of the similar though not the same queries, a query rewriting process is needed. This means when a user performs querying, if we do not find a previous query which is the same as the new one, we can obtain the most similar query from the cache and rewrite the new query according to the old one. In order to perform a query rewriting process, there can be four relationships between two similar query pattern trees as shown in the following:

1. Exact matching. The QPT T_1 exactly matches the QPT T_2 if T_1 is a query pattern subtree of T_2 and T_2 is also a query pattern subtree of T_1.

2. Exact containment. The QPT T_1 exactly contains tree T_2 if T_2 is a query pattern subtree of T_1.

3. Semantic matching. We employ a similar idea of "Extended Subtree Inclusion" (Yang *et al*., 2003b) for the definition of "Semantic Matching". Let T_1 and T_2 be two query pattern trees with root nodes t_1 and t_2, respectively. Denote by *children*(n) the set of child nodes of n. We can recursively determine that T_1 semantically matches T_2 if t_1 and t_2 ($t_1 \leq t_2$) satisfy one of the following three conditions:

(1) Both t_1 and t_2 are leaf nodes;

(2) t_1 is a leaf node and t_2="//", then $\exists t_2' \in$ *children*(t_2) we have a semantic match (T_1, T_2');

(3) Both t_1 and t_2 are non-leaf nodes, and one of the followings holds:

(i) $\forall t_1' \in$ *children*(t_1), $\exists t_2' \in$ *children*(t_2) we have a semantic match (T_1', T_2');

(ii) t_2="//" and $\forall t_1' \in$ *children*(t_1) we have the semantic match (T_1', T_2);

(iii) t_2="//" and $\exists t_2' \in$ *children*(t_2) we have the semantic match (T_1, T_2').

4. Semantic containment. The QPT T_1 semantically contains tree T_2 if any query pattern subtree of T_1 semantically matches query patterns T_2.

In Fig.8 we describe four query pattern tree rewriting cases. In each case, the left query pattern tree is an existing one, and the right query pattern tree is a rewritten one. In Fig.8a, because two queries exactly match each other, the result of the issued query can be easily acquired from the cached query. In Fig.8b, the new query exactly contains the old query, so the result of the left query subtree can be obtained from the cached query. We just need to compute the result of the right subtree and merge the result with the query result of the cached one. In Fig.8c, since the new query semantically matches a cached query, we can retrieve the result through two steps. Firstly, the result of the semantically matched query is obtained. Secondly we compute the parent-child relationship of nodes "book" and "title", nodes "title" and "C++" respectively instead of the original grandparent-grandchild relationship of nodes "book" and "C++".

By means of semantic matching, we reduce the search space and only need to compute relationships on different labels between queries. Fig.8d shows a semantic containment case. Since the new query

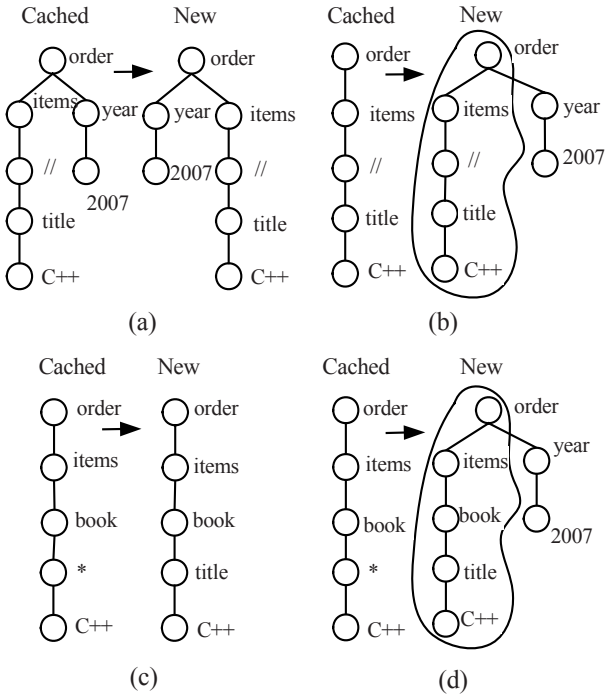

Fig.8 Four relationships between two similar query pattern trees in query pattern tree rewriting. (a) Exact matching; (b) Exact containment; (c) Semantic matching; (d) Semantic containment

semantically contains the cached one, the result of the left query subtree can be obtained with the semantic matching method. Then we calculate the right subtree and merge results of the two subtrees to achieve the final result.

Cache replacement

The results of cached queries are replaced because new frequent query pattern trees are discovered and the old ones are replaced by the new ones. The query pattern tree mining process is automatically performed when the number of queries reaches the predefined threshold. After the mining process, all user queries are discarded. The next mining process will not be launched until the number of issued queries reaches the threshold again. The specified support of the mining process is self-tuned to adapt to the limited size of the cache pool. If too many frequent patterns are found in the current mining process, the support will be automatically increased to a larger value by formula *support=support**(1+*support_increase_percent*) so that fewer frequent query patterns will be mined next time. On the contrary, if too few frequent patterns are discovered, the support is decreased by formula *support=support**(1−*support_decrease_percent*).

After the mining process, new frequent queries will be discovered and answers to some previous queries need to be replaced. We keep track of the following information for each frequent query pattern, namely the recently used frequency, the support, and the time discovered by an iteration of the mining process, as measures for the replacement policy. Among all the statistical data, the recently used frequency is regarded as the most important factor for cache replacement. The used frequency of query patterns is divided into several levels. Query patterns on the lower levels will be given more priority to be selected as victims. The support and the discovered time are only taken into consideration for cache replacement at the same frequency level. If the cache is full, the replacement manager selects the query with the least support and the latest discovered time to be replaced. A query pattern with a larger support should not be selected as a victim because it is contained in more issued queries. To avoid unnecessary computation, a lazy-result-retrieval scheme is employed, i.e., the answer to a frequent query is not retrieved until the query is really used.

EXPERIMENTS

In this section we evaluate both the performance of our mining algorithm VBUXMiner and the XML querying improvement by caching the frequent query patterns mined with our algorithm. We first compare VBUXMiner vs. previous algorithms XQPMinerTID and FastXMiner. Then we investigate the effectiveness of applying the mining algorithm in XML query caching. All the mining algorithms, the prototypes of the caching system, are implemented in Java language, and the experiments are carried out on an Intel Xeon 2.0 GHz computer with 2 GB RAM running RedHat Linux 9.0.

Query mining performance

To simulate XML queries, we employ the XMARK.DTD (http://monetdb.cwi.nl/xml/) as the DTDs and the DBLP.DTD (http://www.informatik.uni-trier.de/~ley/db/) as the schemas to generate the rooted query pattern trees. In order to produce more general queries, we introduce some wildcard "*" and descendant path "//" into the query pattern trees. Three steps are used to create the data sets. First of all, we translate the DTDs into DTD trees, adding four "*" and four "//" into DBLP.DTD, and seven "*" and seven "//" into XMARK.DTD. Secondly, we generate two query databases, each containing 5 000 000 different queries from the two respective DTD trees. Finally, we randomly select a number of queries (ranging from 30 000 to 300 000) from the previous step. As it is known that the FastXMiner and XQPMinerTID algorithms can only be used to discover frequent rooted query pattern trees with the same root node, we only compare the performance of VBUXMiner to them on datasets with the same root node, although VBUXMiner can also mine frequent rooted query pattern trees with different root nodes. Therefore, when generating the data sets we assume that all query patterns in the same data set have the same root node. In Table 1 we show the characteristics of the data sets with a varying number of queries from 30 000 to 300 000 for DBLP and XMARK.

When comparing the performance of the three mining algorithms, namely VBUXMiner, FastXMiner and XQPMinerTID, we need to slightly modify the second and the third algorithms for fair comparison. This is because VBUXMiner does not consider semantic containment of wildcard and descendant

Table 1 Characteristics of data sets for DBLP and XMARK with varying numbers of queries

Number of queries ($\times 10^3$)	DBLP					XMARK				
	Average nodes	Max nodes	Average depth	Max depth	Max fanout	Average nodes	Max nodes	Average depth	Max depth	Max fanout
30	11.95	14	4.77	6	8	10.18	12	4.99	9	11
60	11.94	13	4.76	7	8	10.17	12	4.98	10	11
90	11.96	14	4.77	7	8	10.17	11	4.99	9	11
120	11.94	12	4.76	6	8	10.18	13	4.99	10	11
150	11.95	13	4.77	7	8	10.17	13	4.98	9	11
180	11.95	14	4.77	6	8	10.18	12	4.98	9	11
210	11.95	13	4.76	6	8	10.18	11	4.98	9	11
240	11.94	12	4.77	7	8	10.17	12	4.99	9	11
270	11.95	13	4.76	6	8	10.18	13	4.99	9	11
300	11.96	14	4.76	6	8	10.18	12	4.99	10	11

path when generating frequent rooted query pattern trees. However, the original XQPMinerTID and FastXMiner algorithms both process the semantic containment relationships. Therefore, we replace the containment computation in the original code of these two algorithms with a simplified version, which only consists of an ordinary subtree inclusion determination process. Via such slight modification, we can obtain the same frequent patterns from the three algorithms.

Before the mining process, we assume that all the QPTs are loaded into the main memory. Therefore, all subsequent operations on the QPTs are performed in memory, and there are no disk accesses when scanning the data sets.

Fig.9 shows the performance results of VBUXMiner vs. FastXMiner and XQPMinerTID with a varying number of QPTs from 30 000 to 300 000. The specified minimum support is set to 1%. From the experimental results, we find that VBUXMiner is about 20% faster than FastXMiner and XQPMinerTID. Specifically, when the data set becomes larger, the improvement is also more obvious. Two reasons may lead to the high efficiency of VBUXMiner. First of all, infrequent nodes are pruned using CGTG before candidate generation, which results in less enumeration of candidates. This is in consistency with our experimental results in Fig.10, which demonstrates that the VBUXMiner algorithm generates fewer candidates and thus incurs less computation of the supports of query pattern trees compared to the other two. Secondly, unlike QPMinerTID and FastXMiner, VBUXMiner does not require data set scans, which are needed to compute the support of candidates when determining whether the candidates are frequent. Although FastXMiner and XQPMinerTID employ various optimization schemes to reduce the need for data set scan, it cannot be avoided in some special situations such as leaf node expansion etc. Since we store the data sets in the memory in our experiments, the cost of a data set scan is significantly underestimated. Therefore, we expect higher performance improvements in VBUXMiner in a real disk-resident environment.

A similar result can be obtained from experiments on various minimum supports. In Fig.11, we make comparisons among the three algorithms with varying supports from 0.2% to 2% on data sets with 150 000 QPTs. Just like previous results, VBUXMiner performs more efficiently than the other two algorithms. The improvement is more obvious when the support is low. This is because more candidates need to be enumerated and the number of the support computations of the candidates increases. Likewise, we present the results of the number of enumerated candidates with varying minimum supports in Fig.12.

Cache performance

We apply our mining algorithm in the caching prototype system to improve querying performance. We evaluate the performance of the caching scheme using FRQPT, and compare it with traditional caching policies LRU and MRU. The system accepts tree-patterns as its queries, and obtains the results using the structural join approach (Al-Khalifa *et al*., 2002). When conducting experiments, we load the XML data set into the memory and create indices for XML data before querying.

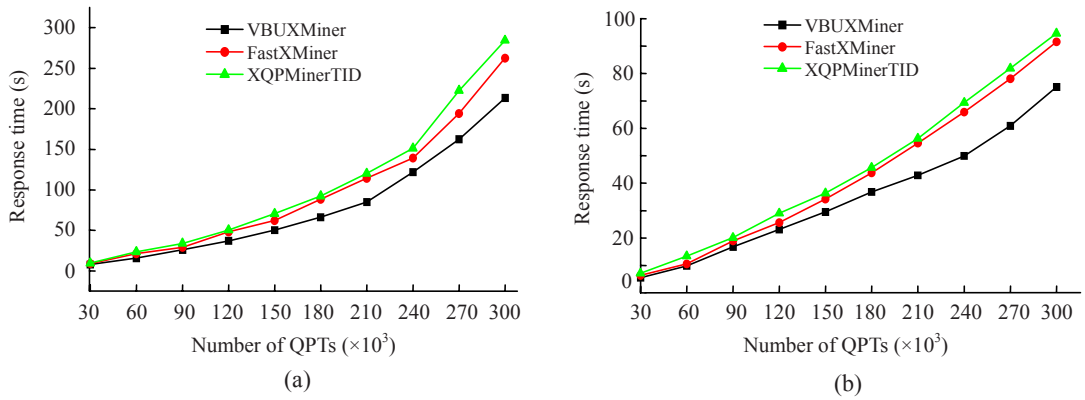

Fig.9 Response time with varying numbers of QPTs. (a) DBLP; (b) XMARK

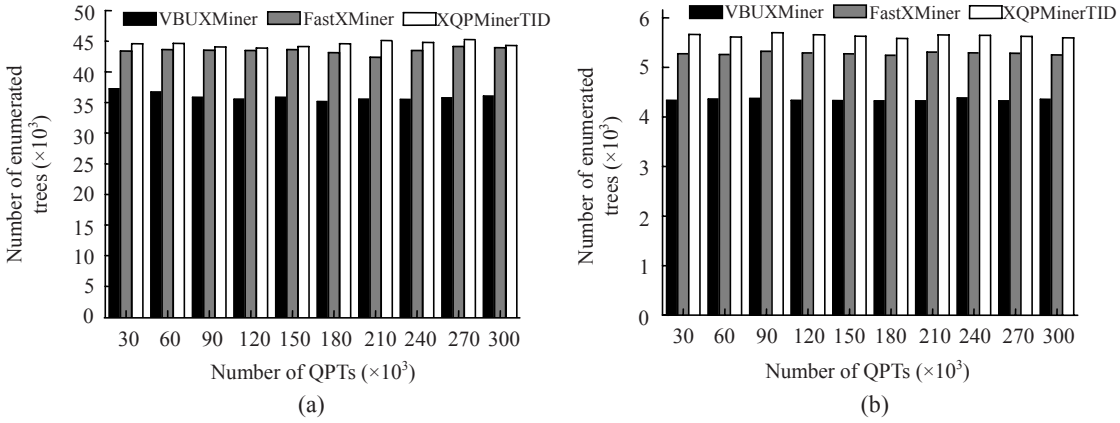

Fig.10 Number of enumerated trees with varying numbers of QPTs. (a) DBLP; (b) XMARK

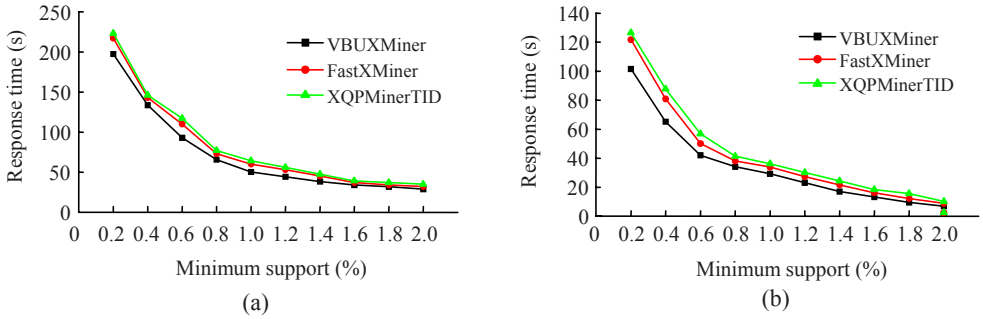

Fig.11 Response time with varying minimum supports. (a) DBLP; (b) XMARK

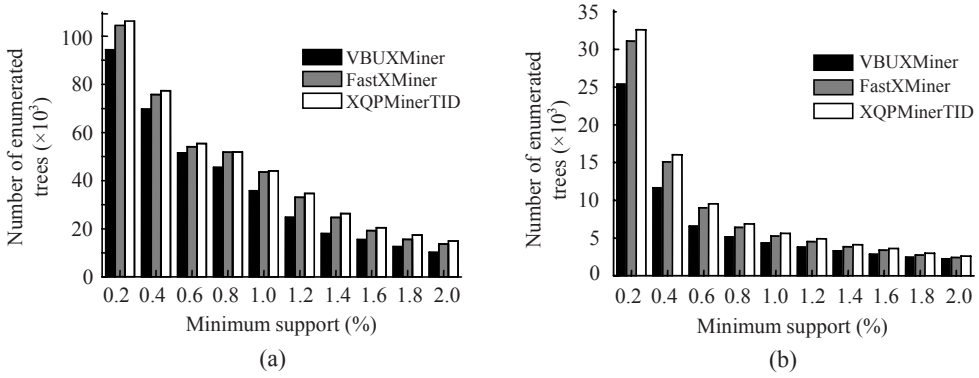

Fig.12 Number of enumerated trees with varying minimum supports. (a) DBLP; (b) XMARK

The data set we use is a 116-M XML document generated by the XMARK tool. Queries are generated randomly using the probabilities in Table 2. The wildcard * and descendant path // are added to make the generated queries more complex.

Table 2 Probabilities of tags

Tag	Prob.	Tag	Prob.	Tag	Prob.
Site	1.0	Australia	0.2	Person	0.7
Regions	0.6	Item	0.7	Person/Name	0.8
People	0.8	Item/Name	0.4	Emailaddress	0.5
Africa	0.2	Incategory	0.3	Address	0.2
Europe	0.5	Quantity	0.2	*	0.1
Asia	0.5	Mailbox	0.1	//	0.2

Fig.13a presents the average response time for query processing with a fixed cache size of 100 queries and varying numbers of queries from 20 000 to 200 000, where Q20k stands for 20 000 queries. The initialized support of the mining procedure is set to 5% and the threshold number of mined queries is 1000 for the frequent queries caching policy. The average response time is defined as the ratio of total running time for answering a set of queries to the total number of queries in this set. From the results we see that FRQPT is more efficient compared to LRU and MRU. This is because the LRU and MRU policies can handle similar but not exactly the same queries. Caching system employing FRQPT policy also has a good scalability. The response time for querying does not increase much when the number of queries varies from 20 000 to 200 000. In Fig.13b we illustrate the hit ratio of the caching system with a varying number of queries. From these results we see that FRQPT policy has the highest hit ratio.

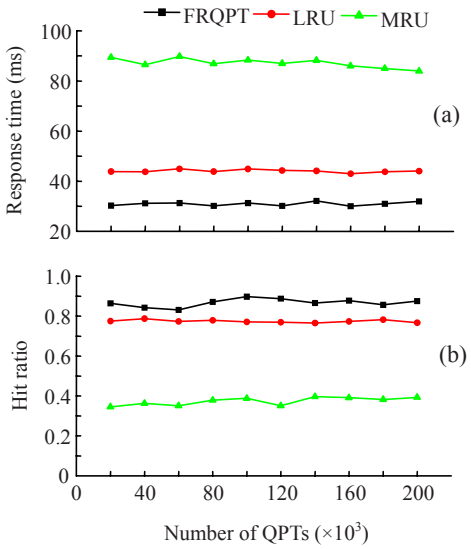

Fig.13 Effective FRQPT caching policy. (a) Average response time; (b) Hit ratio

In Fig.14 we show the average response time for query processing and the hit ratio with varying support values from 0.01 to 0.1 for the data set Q100k. The varying support does not influence the query response time much. This is because we employ a self-tuning scheme to automatically adjust the mining support to be fit for the cache pool. The support may be tuned to a stable value after a number of iterations of the mining process.

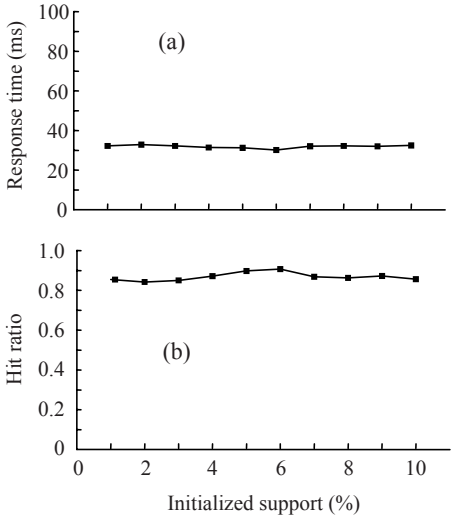

Fig.14 Average response time (a) and hit ratio (b) for varying initialized supports for FRQPT

CONCLUSION

In this paper, we present an efficient mining algorithm called VBUXMiner to extract frequent rooted query pattern trees from XML queries, and apply the mining approach to a caching mechanism to accelerate XML query processing. We mine frequent query patterns over a query schema called "compressed global tree guide" (CGTG), which prunes infrequent nodes and employs a node compressing scheme. We discover frequent query patterns from bottom to top and generate tree candidates at each node through the equivalence class joining process. When deciding whether a candidate is frequent, we

avoid a dataset scan since the support of each candidate tree can be computed by joining the QPT ID lists recorded in the CGTG. To apply the mining approach to XML query caching, we introduce four kinds of query relationships and employ a query rewriting process to deal with similar queries. Our experimental results show that the proposed mining approach outperforms previous mining algorithms XQPMinerTID and FastXMiner in terms of efficiency. The caching policy using our mining results outperforms the traditional LRU and MRU caching policies.

References

Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y., 2002. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. 18th ICDE, p.141-152. [doi:10.1109/ICDE.2002.994704]

Asai, T., Abe, K., Kawasoe, S., Arimura, H., Satamoto, H., Arikawa, S., 2002. Efficient Substructure Discovery from Large Semi-structured Data. Proc. 2nd SIAM Int. Conf. on Data Mining, p.158-174.

Asai, T., Arimura, H., Uno, T., Nakano, S., 2003. Discovering Frequent Substructures in Large Unordered Trees. Proc. 6th Int. Conf. on Discovery Science, p.47-61. [doi:10.1007/b14292]

Bei, Y.J., Chen, G., Dong, J.X., 2007. BUXMiner: An Efficient Bottom-Up Approach to Mining XML Query Patterns. Proc. APWEB/WAIM, p.709-720. [doi:10.1007/978-3-540-72524-4_73]

Chehreghani, M.H., Rahgozar, M., Lucas, C., 2007. Mining Maximal Embedded Unordered Tree Patterns. Proc. IEEE Symp. on Computational Intelligence and Data Mining, p.437-443. [doi:10.1109/CIDM.2007.368907]

Chen, L., Rundensteiner, E.A., Wang, S., 2002. XCache: A Semantic Caching System for XML Queries. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.618. [doi:10.1145/564691.564771]

Chen, L., Bhowmick, S.S., Chia, L.T., 2005. Mining Positive and Negative Association Rules from XML Query Patterns for Caching. Proc. 10th DASFAA, p.736-747. [doi:10.1007/11408079_67]

Chi, Y., Yang, Y., Muntz, R.R., 2003. Indexing and Mining Free Trees. Proc. 3rd ICDM, p.509-512. [doi:10.1109/ICDM.2003.1250964]

Chi, Y., Yang, Y., Muntz, R.R., 2004. HybridTreeMiner: An Efficient Algorihtm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms. Proc. 16th Int. Conf. on Scientific and Statistical Database Management, p.11-20. [doi:10.1109/SSDBM.2004.41]

Gu, M.S., Hwang, J.H., Ryu, K.H., 2007. Frequent XML Query Pattern Mining Based on FP-Tree. Proc. DEXA Workshops, p.555-559. [doi:10.1109/DEXA.2007.78]

Hong, J.W., Kang, H., 2005. Data Integration and Cache-Answerability of Queries through XML View of Data Source on the Web. Proc. IMSA, p.242-247.

Kim, Y., Park, S.H., Kim, T.S., Lee, J.H., Park, T.S., 2006. An Efficient Index Scheme for XML Databases. Proc. SOFSEM, p.370-378. [doi:10.1007/11611257_35]

Kutty, S., Nayak, R., Li, Y., 2007. PCITMiner-Prefix-Based Closed Induced Tree Miner for Finding Closed Induced Frequent Subtrees. Proc. 6th AusDM, p.151-160.

Luccio, F., Enriquez, A.M., Rieumont, P.O., Pagli, L., 2001. Exact Rooted Subtree Matching in Sublinear Time. Technical Report TR-01-14.

Nayak, R., Iryadi, W., 2006. XMine: A Methodology for Mining XML Structure. Proc. APWeb, p.786-792. [doi:10.1007/11610113_74]

Paik, J., Kim, U.M., 2006. A Simple Yet Efficient Approach for Maximal Frequent Subtrees Extraction from a Collection of XML Documents. Proc. WISE Workshops, p.94-103. [doi:10.1007/11906070_9]

Seo, D.M., Yoo, J.S., Cho, K.H., 2007. An Efficient XML Index Structure with Bottom-Up Query Processing. Proc. ICCS, p.813-820. [doi:10.1007/ 978-3-540-72588-6_131]

Yang, L.H., Lee, M.L., Hsu, W., 2003a. Efficient Mining of XML Query Patterns for Caching. Proc. 29th VLDB, p.69-80. [doi:10.1016/B978-012722442-8/50015-X]

Yang, L.H., Lee, M.L., Hsu, W., Acharya, S., 2003b. Mining Frequent Query Patterns from XML Queries. Proc. 8th DASFAA, p.355-362. [doi:10.1109/DASFAA.2003.1192401]

Zaki, M.J., 2002. Efficiently Mining Frequent Trees in a Forest. Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, p.71-80. [doi:10.1145/775047.775058]

Zaki, M.J., 2005. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, **65**:33-52.