Mining XML Frequent Query Patterns

Cheng Hua[1], Hai-jun Zhao[1], Yi Chen[2]
1 Guangdong Electronic Business Market Application Key Laboratory, Guangdong University of Business Studies. #21, Chisha Road, Haizhu District, Guangzhou, Guangdong Province. 510320, P.R.C.
huacheng@gdcc.edu.cn
WWW home page: http://gdec.gdcc.edu.cn
2 Ricoh Software Research Center (Beijing) Co., Ltd.

Abstract. With XML being the standard for data encoding and exchange over Internet, how to find the interesting XML query characteristic efficiently becomes a critical issue. Mining frequent query pattern is a technique to discover the most frequently occurring query pattern trees from a large collection of XML queries. In this paper, we describe an efficient mining algorithm to discover the frequent query pattern trees from a large collection of XML queries.

1 Introduction

With the increase in XML applications such as e-business transactions, XML middleware systems, effective and efficient delivery of XML data has become an important issue. Regular path expression (RPE) is a common feature of XML query languages, and processing RPE queries can be expensive since it involves navigation through the hierarchical structure of XML, which can be deeply nested.

Mining frequent sub tree is a technique to discover the most frequently occurring sub trees from a large collection of relevant information, and it has been widely applied in domains like bioinformatics, web mining, and structured-based document clustering, and so on. In [1, 2, 3], some discussions have been given on mining frequent query pattern. In this paper, we will describe an efficient mining algorithm to discover the frequent query pattern trees from a large collection of XML queries.

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. We discuss the related work and conclude in Section 5.

Please use the following format when citing this chapter:

Hua, C., Zhao, H., Chen, Y., 2007, in IFIP International Federation for Information Processing, Volume 251, Integration and Innovation Orient to E-Society Volume1, Wang, W. (Eds), (Boston: Springer), pp. 26-34.

2 Preliminaries

In this section, we first define the concept of a query pattern tree which forms the basis of the XQPMiner and XQPMinerTID. Then we explain why the simple tree matching technique is not applicable in finding frequent query patterns for XML data. Finally, we give a formal definition of the query pattern mining problem.

2.1 Query Pattern Tree

For each XML query q_i issued, we can extract related information by transforming the XML query into XML algebra[4]. This information includes the result that users want, the filtering conditions applied and the XML files involved in the query. Such information can be represented in the following form:

q_i*{resultPattern; predicates; documents}*

where resultPattern is the result schema pattern; predicates is the filtering conditions used in the XML query; and documents is the XML data files involved in this query.

Note that this transformation only includes the paths or patterns of the original schema instead of the restructured part. For example, given the following XML query in XQuery [5] syntax:

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

Q_1 can be expressed using the algebra proposed in [4] as follows:

$\nu(\text{result})(\nu(\text{book})(\in(\$b/\text{title},\$b/\text{author},\$b/\text{price})(\sigma_{\$a/\text{last}/\text{data}()=\text{"Buneman"}}(\phi_{\$a=\$b/\text{author}}$
$(\phi_{\$b=/\text{book}}(s(\text{book.xml})))))))$

After resolving the path expressions involved in the query, Q_1 relevant information is extracted below:

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

Some preprocessing is necessary. For example, substituting parent with the actual parent node or its binding variable. After extracting the path expressions, we will obtain three types of label: element tag name, wildcard *, and relative path //. The wildcard "*" indicates any label (tag) in DTD; and the relative path "//" indicates zero or more labels(descendant-or-self). Here, we use the same notations as those in XQuery[4] and XPath[6]. According to the binding variable relationship, the pattern tree can be easily constructed by combining *resultPattern* and *predicates* by extracting the path and ignoring the constants. Note, when adding the path(s) of *predicates* to *resultPattern*, if the content of the path(s) is already contained in *resultPattern*, the path(s) will not be added. Take Q_1 for example, the content of path "/book/author/last/" is contained in *resultPattern*, so this path will not be added to *resultPattern*. Formally, a pattern tree is defined as follows

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

For simplicity, we use label to represent a node. The query pattern tree QPT_1 of Q_1 can be represented as:

```
<book>
    <title></title>
    <author></author>
    <price></price>
 </book>
```

The corresponding graph representation is shown in Figure 1(a).

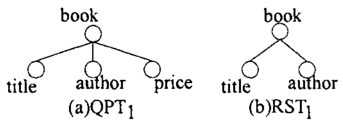

Figure 1: The Query Pattern Tree For Q1 And A Root Subtree.

Definition 2 (***Rooted Subtree***): Given a query pattern tree QPT=<V,E>, a rooted subtree *RST*=<V',E'> of QPT is a subtree of QPT that satisfies the following conditions:

- *root*(RST)=*root*(QPT)
- $V' \subseteq V$, $E' \subseteq E$

One of rooted subtrees of QPT_1 is shown in Figure 1(b).

Let *T* be a tree, the size of T is defined by the number of its nodes |T|.

2.2 Tree Pattern Matching

In general, a tree T=<V, E> matches another tree T'=<V', E'> if there exists a mapping φ which satisfies:

- *root*(T')=φ(*root*(T)) and $\forall v \in V$, $\exists v' \in V'$,s.t. v'=φ(v),where *v*.label=v'.label.
- φ preserves the parent-child relation: if (v1,v2)∈E, then (φ(v1), φ(v2))∈E'

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

Unfortunately, this definition is not applicable to our problem here because of the presence of wildcards "*" and relative path expressions "//". For instance, when comparing two trees T1 and T2 in

Figure 2, it is obvious that the path "book/section/figure/title" in T2 can match the path "book//title" in T1, because "//" in "book//title" can be zero or more labels between book and title. It is the same with "book/section/figure/image" and "book//image". Hence the tree T2 can match T1. In other words, T2 is contained in T1, which is written as T2 $\subseteq$ T1. One might try to expand those non-deterministic paths to deterministic paths such as expanding the path "book//title" to "book/section/figure/title". But this is only feasible when the XML DTD (schema) is a DAG. The method fails for a DTD with cycles.

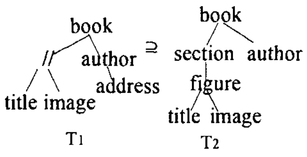

Figure 2: An Example Of Pattern Tree Containment

Expanding "//" remains crucial. This is because without the context information, one cannot tell whether a path is contained in "//" or not. For example, while it is clear that the path "/book/section/figure/title" is contained in "/book//title", we are not sure if the path "/book/section/figure/" is contained in "/book//title". The reason is that the former two paths share the same leaf. A more complicated example of query pattern QPT is given in Figure 3. We cannot merge the two child nodes "//" under node "book" because node "title" and node "image" may not share the same parent node. Thus we have RST_1 and RST_2 are contained in QPT while RST_3 is not contained in QPT by simple tree matching.

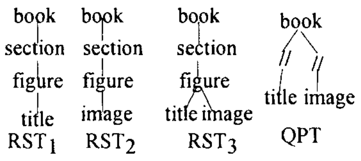

Figure 3: A Complex Query Pattern

We will expand a node "//" in QPT from XML schema as follows. Assuming the node "//" to be expanded has a child n. The expansion is straightforward if no cycles exist in XML schema. When a cycle exists and one of the expanded paths is root/.../p/n where n's parent p has a child that points back to p's ancestor, we'll introduce a node "//" between p and n. Consider Figure 4 as example. The XML schema includes a cycle, and the QPT to be expanded is "part//num". By straightforward expansion, we'll have "part/subpart/num", "part/subpart/part/subpart/num" and so on. By using the "//"node, we can concisely represent it as "part/subpart//num". After such expansion, we add context information to the QPT and do not introduce a cycle in QPT.

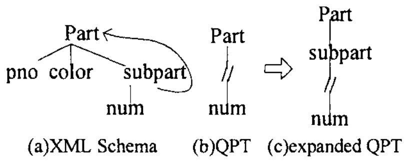

Figure 4: An Example Of Path Expansion

One may wonder why not extract the result XML schema and mine it? There are two reasons. One is that it's unrealistic to extract the schema online. Another reason is that this method effects only if the retrieved XML data is static. For dynamic changing XML data, it will fail.

To decide whether one pattern tree is contained in another, the exact tree matching cannot be naively used because of the existence of wildcards and relative path. By analyzing the relationship among labels and paths, we can derive that any node matches a node with label "*", and, on the other hand, it is contained in a node with label "//". Thus, the path "/book/*/figure" is contained in the path "/book//figure", while "/book/section/figure/title" is contained in "/book/*/figure/title". Note that the notion of containment here consists in the structure containment not in the extent. So our definition is different from [1].

Based on the above discussion, we can infer that the labels in the two pattern trees satisfy the partial order relationship≤:

- Given label l, *l*≤l, i.e., a node with label l matches with a node with the same label; Hence, we have *≤* and //≤//.
- *l*≤*≤//,i.e., a node with label l matches a node with label * and a node with label * matches a node with label //.

Therefore, the tree matching definition should be generalized. To decide if a RST is contained in a QPT, basically, it can be stated as follows:

(1) root nodes are matched. In our setting, they must have the same label.

(2) If node $w \in$RST is matched with node $v \in$QPT, it satisfies:

a) w.label≤v.label

b) each subtree of w is contained in some subtree of QPT

The corresponding procedure will be given in later section.

2.3 Frequent query pattern mining problem

After transforming a set of XML queries $\{q_1,\ldots,q_N\}$ into query pattern trees $\mathbf{D}=\{QPT_1,\ldots,QPT_N\}$, mining the frequent query pattern means to discover the frequent rooted subtrees (RSTs) in the query pattern trees. A natural approach is to divide the queries into different categories according to the XML data files that are involved.

Given a rooted subtree RST, RST matches a query pattern tree QPT in D, or RST occurs in D, if there exists a QPT that contains RST. The total occurrence of RST in D is denoted as Freq(RST), and its support rate is denoted by supp(RST)=Freq(RST)/|D|. For a positive number σ, RST is σ-frequent in D if supp(RST)$\geq\sigma$. Hence, the query pattern mining problem can be stated as:

Frequent Query Pattern Mining Problem:

Given a query pattern tree database $D=\{QPT_1,\ldots,QPT_N\}$, and a positive number $0<\sigma\leq 1$ called the minimum support, find all σ-frequent rooted subtrees *F* such that $\forall RST\in F$, supp(RST)$\geq\sigma$.

For instance, consider the example in Figure 5. The RST occurs in two of query pattern trees and thus is frequent with respect to this database with supp(RST)=2/3 and Freq(RST)=2.

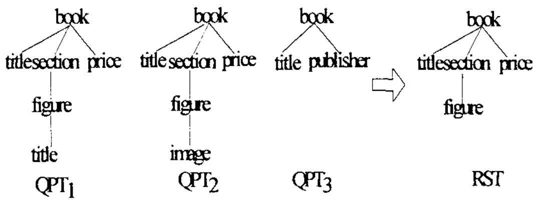

Figure 5: An Example Of Frequent Rooted Subtree

3 Discovering Frequent Rooted Subtrees

In this section, we propose an efficient algorithm for discovering the frequent query patterns. In our frequent pattern mining setting, the dataset of transactions *D* is a set of pattern trees. Each transaction $t\in D$ is a labeled directed pattern tree extracted from a XML query. Given a minimum support *minSupp*, we would like to find the frequently occurred rooted subtrees(RSTs) in at least *minSup**|D| transactions.

The main framework of our algorithm QPTMiner is shown in Figure 4. The notation RST^k denotes a k-edge rooted subtree; F_k a set of frequent k-edge rooted subtree; and C_k a set of k-edge candidate RST. Edges in the algorithm correspond to items in traditional frequent itemset discovery. Given a set of QPTs, QPTMiner initially enumerates all the rooted subtrees of every QPT; and put them in a candidate set C_k, and counts the frequency for each of these candidates. Next, QPTMiner drops those RSTs that do not satisfy the minimal support requirement.

```
Algorithm:QPTMiner(D,minSupp)
Input: D—pattern tree transaction database
minSupp—the minimum support
Output: the set of all frequent RST sets
(1) { $F_i = \phi$| i = 1, ..., n};
(2) for (k = 1, k++, k <= n)
(3)     read $C_k$ from database;
(4) for each $QPT_i$ in D
(5)     $S_i$=enumerate($QPT_i$)
(6)     for each $RST_i^k$ in $S_i$
(7)         for each candidate $RST^k \in C_k$ do
(8)         if Contains( $RST_i^k$ , $RST^k$) then
(9)             $RST^k$.count++;
(10)        else
(11)                $C_k \leftarrow RST^k$
(12)                $RST^k$.count = 1;
(13) for (k = 1, k++, k <= n)
(14)     $F_k=\{RST^k \in C_k | RST^k.count \geq minSupp*|D|\}$;
(15)     save $C_k$ to database;
(15) return {$F_i$| i = 1, ..., n };
```

Figure 6: Algorithm To Find Frequent Rooted Subtree

We compute the frequent pattern trees in an incremental way. After being computed, the frequent query pattern RST^k and their count RST^k.count is maintained in a database (C_k in the algorithm), and every time when we have to re-compute the frequent pattern trees, we read previous result from the database. In this way, previous result can be reused and the computation cost minimized.

The support of each candidate is counted based on the query pattern tree matching definition in section 2, the Contains algorithm can be constructed to compare two trees recursively from root to leaf to decide whether a RST is contained in a QPT. Due to the space limitation the detail of Contain algorithm is not included in this paper, interested reader also can find it in [1].

4 Conclusion

In this paper, we have described a schema-guided mining approach to discover frequent rooted subtrees from XML queries. This approach allows us to enumerate only valid candidates RSTs. We have also developed a tree pattern containment algorithm that takes into account the relative path "//" and wildcare "*" when matching RSTs with query pattern trees.

Future work includes 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. 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.
2. L. H. Yang, M. L. Lee, W. Hsu. *Approximate Counting of Frequent Query Patterns over XQuery Stream*, (DASFAA), 2004.
3. Yi, Chen. *Discovering Ordered Tree Patterns from XML Queries*, PAKDD, 2004.
4. S. Boag (XSL WG), D. Chamberlin, MF. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language, *W3C Recommendation* 23 January 2007, http://www.w3.org/TR/2007/REC-xquery-0070123/.
5. P. Fankhauser, M. Fernández, A. Malhotra, etc. The XML Query Algebra, *W3C Working Draft* 04 December 2000,.
6. http://www.w3.org/TR/xpath.