Improving XML Querying with Maximal Frequent Query Patterns

Yijun Bei, Gang Chen, and Jinxiang Dong

College of Computer Science, Zhejiang University, Hangzhou, P.R. China 310027
alphabyj@yahoo.com.cn, cg@zju.edu.cn, djx@zju.edu.cn

Abstract. Querying on XML data is a computational-expensive process due to the complex nature of both the XML data and the query. In this paper, we propose an approach to expedite XML query processing by caching the results of a specific class of queries, namely the maximal frequent queries. We mine the maximal frequent query patterns from user-issued queries and cache the results of such queries. We propose a recursive algorithm for query processing using the cached query results. Query rewriting is employed to deal with four kinds of similar queries namely exact matching, exact containment, semantic matching and semantic containment. We perform experiments on the XMARK datasets and show that the proposed methods are both effective and efficient in improving the performance of XML queries.

Keywords: Caching, XML Query, Ming, Maximal Frequent Pattern.

1 Introduction

With the proliferation of XML as a standard for data representation and data exchanging, effective and efficient querying techniques of XML data becomes an important topic for the database community. For the sake of flexible query mechanisms, query languages like XPath [1] and XQuery [2] have been provided for XML data retrieval. XML queries represented by these query languages can be modeled as trees which are called query pattern trees.

Caching has played an important part in improving performance of XML query processing, especial for repeated or similar queries. Users can obtain answer right away if the query results have been computed and cached. Several caching approaches [3, 4, 5 and 6] have bee provided for XML queries. However these approaches only consider frequent query pattern trees rooted at the root of DTD. As a result, cache schemes in these approaches don't take effect on queries not rooted at the root of DTD. In our approach, we discover frequent query patterns rooted at any level of a DTD and can deal with arbitrary quires. One nice property of frequent trees is the apriori property, i.e., if a query pattern is frequent, then all of its query sub-patterns are also frequent. In case that we cache a query pattern, we will have to cache all its sub-patterns, which results in some unnecessary cache queries. In order to make full use of limited cache size, we will cache only maximal frequent XML queries.

In this paper we introduce the strategy of caching only maximal frequent XML queries. We present a recursive algorithm for obtaining query result by query

Y. Shi et al. (Eds.): ICCS 2007, Part I, LNCS 4487, pp. 261–269, 2007.
© Springer-Verlag Berlin Heidelberg 2007

rewriting and restructuring. To make full use of limited size of cache pool, we describe a cache replacement scheme that that utilizes recent accessing frequency as well as the query pattern support and query pattern discovered time.

The rest of the paper is organized as follows. Section 2 discusses previous work related to the XML caching and mining. In section 3, we present some concepts used in mining and caching process. In section 4 we describe an algorithm for query processing. Section 5 presents results of experiments and we conclude in section 6.

2 Related Work

Indexing strategy can accelerate XML query processing. However, it will cost much space to store the index in memory. Caching results of XML queries has been considered as a good strategy to improve performance of XML query processing. Semantic caching [3 and 4] has received attention in XML data retrieval. In [3] a compact structure called MIT is employed to represent the semantic regions of XML queries. XCache in [4] is a holistic XQuery-based semantic caching system. [5 and 6] incorporate mining approach to find frequent queries for caching. An efficient algorithm called FastXMiner in [5] is presented to discover frequent XML query patterns. Authors in [6] took into account the temporal features of user queries for frequent queries discovery and design an appropriate cache replacement strategy by finding both positive and negative association rules. However these two approaches only consider frequent query pattern tree rooted at the root of DTD.

For the problem of finding tree-like patterns, many efficient tree mining approaches have been developed. Basically, there are two steps for finding frequent trees in a database. Firstly, non-redundant candidate trees are generated. Secondly, the support of each candidate tree is computed. Asai presented a rooted ordered and a rooted unordered tree mining approach in [7] and [8] respectively. Zaki gave ordered and unordered embedded tree mining algorithms in [9, 10].

3 Background

3.1 Maximal Frequent Query Pattern Tree

Query Pattern Tree. A *query pattern tree* is a rooted tree *QPT* = <N, E> where:

- N is the node set. Each node n has a label whose value is in {"*", "//"} ∪ labelSet where the labelSet is the label set of all elements, attributes and texts.
- E is the edge set. For each edge e = (n_1, n_2), node n_1 is the parent of n_2.

Query Pattern Subtree. Given two query pattern trees T and S, we say that S is a *query pattern subtree* of T or T is a *query pattern supertree* of S, iff there exists a one-to-one mapping φ: $V_S \to V_T$, such that satisfies the following conditions:

- φ preserves the labels, i.e., $L(v) = L(\varphi(v))$, $\forall v \in V_S$.
- φ preserves the parent relation, i.e., $(u,v) \in E_S$ iff $(\varphi(u), \varphi(v)) \in E_T$.

Frequent Query Pattern Tree. Let Q denote all the query pattern trees of the issued queries and d_T be an indicator variable with $d_T(S) = 1$ if query pattern tree S is a query

pattern subtree of T and $d_T(S) = 0$ if tree S is not a subtree of T. The support of query pattern tree S in Q can be defined as $\sigma(S) = \sum_{T \in Q} d_T(S) / \sum_{T \in Q}$, i.e., the percent of the number of trees in Q that contain tree S. A query pattern tree is frequent if its frequency is more than or equal to a user-specified *minimum support*.

Maximal Frequent Query Pattern Tree. A frequent query pattern tree is *maximal* if none of its proper supertrees is frequent.

We show a set of query pattern trees in Figure 1(a) and three query pattern subtrees in Figure 1(b). Given the minimum support 0.6, then the subtree S_1 is not a frequent one, while S_2 and S_3 are frequent query pattern trees. However, S_2 is not a maximal frequent query pattern tree, since one of its supertree S_3 is also a frequent one.

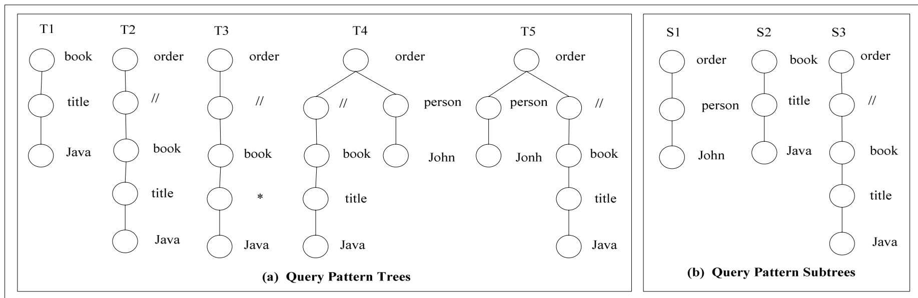

Fig. 1. Maximal Frequent Query Pattern Tree

3.2 Query Rewriting

Few users issue the exactly same queries when performing XML Querying. However, issued queries are often similar and have little difference with others. Therefore, when user issues a new similar query, a query rewriting should be performed to utilize cached queries. We introduce 4 relationships between two similar query pattern trees.

Exact Matching. If query pattern tree T_1 is a query pattern subtree T_2 and T_2 is also a query pattern subtree of T_1, then tree T_1 exactly matches tree T_2.

Exact Containment. If a query pattern tree T_2 is a query pattern subtree of T_1, then T_1 exactly contains tree T_2.

Semantic Matching. We employ a similar idea of *Extended Subtree Inclusion* [5] 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. Let children(n) denotes the set of child nodes of n. We can recursive determine if tree T_1 semantically matches tree T_2, as follows:

$t_1 \le t_2$ and satisfies:

- both t_1 and t_2 are leaf nodes; or
- t_1 is a leaf node and t_2 = "//", then $\exists\, t_2' \in \text{children}(t_2)$ such that semanticmatch(T_1,T_2'); or
- both t_1 and t_2 are non-leaf nodes, and one of the following holds:

1) $\forall t_1' \in$ children(t_1), $\exists t_2' \in$ children(t_2) such that semanticmatch(T_1',T_2');

2) t_2 = "//" and $\forall t_1' \in$ children(t_1) we have semanticmatch(T_1',T_2);

3) t_2 = "//" and $\exists t_2' \in$ children(t_2) we have semanticmatch(T_1,T_2');

Semantic Containment. If any query pattern subtree of T_1 semantically matches query patterns T_2, then we have T_1 semantically contains tree T_2.

Figure 2 outlines 4 query rewriting cases. In Figure 2(a), result of the issued query can be easily acquired as the new query exactly matches the cached query. In Figure 2(b), the new query exactly contains the cached query, so result of left query subtree can be obtained from the cached one, and computation of the right one is enough. In Figure 2(c), since the new one semantically matches a cached one, result of the issued query can be retrieved through two steps. Firstly, results of semantic one is obtained. Secondly we compute the parent-child relation of nodes book and title, node title and java respectively instead of original grandparent-grandchild relation of node book and java. Using semantic matching, we reduce the search space and only need to compute relations on different labels between queries. Figure 2(d) presents a semantic containment case. Since the new query semantically contains the cached one, the result of left query subtree can be obtained with semantic matching method. Then we compute the right subtree and merge results of the two subtrees to achieve final result.

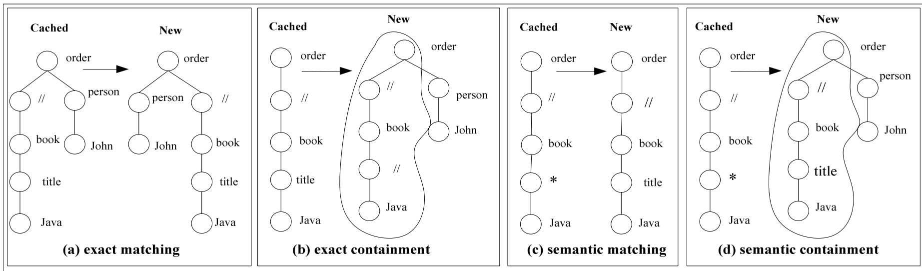

Fig. 2. Query Rewriting

4 Query Processing

4.1 Maximal Frequent Query Patterns Discovery

We use existent efficient mining algorithm for discovery of maximal frequent query patterns and perform the mining process automatically. We record all user queries and launch a mining process when the count of queries reaches our predefined value. Further more, the specified minimum support will be automatically tuned to be fit for cache pool. Suppose the mined maximal frequent queries are too many, which is larger than our cache pool size. We will automatically tune our minimum support threshold to be a larger one and discover less frequent queries next time. On the

contrary, on condition that the frequent queries are less, we need decrease the minimum support to adapt to cache pool.

4.2 Querying Algorithm

Figure 3 outline an algorithm for XML querying using frequent patterns caching. The querying process comprises two parts: matched query discovery and general query performing. The input of the algorithm is a query pattern tree and the output is a set of matched root nodes. First of all, we try to find an exactly matched or semantically matched query from cache pool. If an exactly matched query is found, then result can

```
Algorithm Query(qpt)
Input: qpt (query pattern tree)
Output: nodeset (matched root node set)
matchedqpt = obtain query pattern exactly matching qpt from cache;
if (matchedqpt != null ) // exactly matched query exist
    nodeset = result of matchedqpt;
    return;
end if
matchedqpt = obtain query pattern tree semantically matching qpt from cache;
if (matchedqpt != null)    // semantically matched qpt exist
    if (semantic match from ordinal label l to wildcard "*" or descendent path "//")
        compute child-parent relation between l and parent node of "*" (or "//");
        compute parent-child relation between l and child ndoe of "*" (or "//");
        merge join the two result set;
    else if (semantic match from wildcard "*" to descendent path "//")
       compute grandparent-grandchild relation between parent of "//" and child of "//";
    end if
    nodeset = result of matchedqpt satisfying previous relation computation;
    return;
end if
root = root node of qpt;
nodeset = get all nodes from XML document equals to label root;
for (each direct query pattern subtree dqpst of root)
    if (nodeset == Φ)
        return;
    end if
    subnodeset = query (dqpst);
    switch( type of root of dqpst)
        case ordinal_label:
            nodeset = merge join parent-child nodes(nodeset, subnodeset);
        case "*"
            nodeset = merge join grandparent-grandchild nodes(nodeset, subnodeset);
        case "//"
            nodeset = merge join ancestor-descendent nodes(nodeset, subnodeset);
    end switch
end for
return;
```

Fig. 3. Querying Algorithm

be acquired directly from cache. In case that a semantically matched query is found, result from cache is required a simple modification, as follows:

- Suppose semantic match is from an ordinal label *l* to wildcard "*" or descendent path "//". Two relations, i.e., child-parent relation between label *l* and parent of "*" (or "//"), parent-child relation between label *l* and child of "*" (or "//"), need computed.
- Suppose semantic match is from wildcard "*" to descendent path "//". In order to transform from ancestor-descendent relationship to grandparent-grandchild relationship, a grandparent-grandchild relationship should be computed between parent of "//" and child of "//".

From cache pool, we don't obtain exactly contained or semantic contained queries due to time-consuming computations of the two relationships. We adopt an approximate approach to simulate them. We perform a top to down search starting at the root of the query until we find exact matched or semantic matched query. In this way, we not only avoid complex computation, but also may enhance cache hit ratio in respect that a partial query is highly possible to be cached.

If there not exist a matched query in cache pool, a general querying process needs performed. We join the node set of root with result of each direct subtree of root. Assume the matched root node set denoted as *nodeset* and result of direct subtree is denoted as *subnodeset*, the merge join process is defined as follows:

- If the root of direct subtree has an ordinal label, parent-child relation computation is required to join the two node sets *nodeset* and *subnodeset*.
- If the root of direct subtree has a wildcard "*" label, grandparent-grandchild relation computation is required to join the node sets *nodeset* and *subnodeset*.
- If the root of direct subtree has a descendent path "//" label, ancestor-descendent relation computation is required to join the node sets *nodeset* and *subnodeset*.

4.3 Cache Replacement

We keep track of recent accessing frequency, support and discovered time of maximal frequent query patterns. Recent accessing frequency is considered as the most important factor for cache replacement. We divide accessing frequency of query patterns into several levels. Query patterns in lower levels will be selected as victims. Support and discovered time are also taken into account for cache replacement. For query patterns in the same level, we will need to consider the support and discovered time. When the cache is full, the replacement manager selects the query with least support and latest discovered one. Query pattern with a larger support should not be selected as a victim since more issued queries containing current query pattern. To avoid unnecessary computation, a lazy-result-retrieval scheme is employed. Result of frequent query is not retrieved until the query needs to be used.

5 Experimental Evaluation

In this section, we present experimental results of the prototype system of XML querying algorithm by caching maximal frequent query patterns. We implemented it

in Java and carried out all experiments on an Intel Xeon 2GHz computer with 1GB of RAM running operating system RedHat Linux 9.0. We loaded XML dataset into memory and created indices for XML data before querying. Structural joins [11] approach is employed to decide node relationships when performing querying.

5.1 Dataset

We generate a XML document with size 116MB using tools provided by XMARK [12] for performance test. Queries are generated randomly using the following probabilities: site(0.5), regions(0.8), people(0.7), , africa(0.1), europe(0.5), asia(0.4), australia(0.2), item(0.5), item/name(0.3), incategory(0.4), quantity(0.2), person(0.6), person/name(0.5), emailaddress(0.6), *(0.1), //(0.1). The wildcard * and descendent path // are added to make the queries more complex. In the following experiments, we will set the cache size to 50 which means at most results of 50 queries will be cached.

5.2 Cache Performance

We evaluate the performance of caching scheme using maximal frequent query patterns (MFQP), and compare it with caching policy using frequent patterns (FQP), LRU caching policy (LRU) and MRU caching policy (MRU).

In Figure 4(a) we show the average response time for query processing with varying number of queries from 10,000 to 100,000, in which Q10K for total number of queries Q = 10,000, Q100K for Q = 100,000. The support of frequent queries mining is 0.1 for frequent queries cache 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. MFQP is the most efficient compared to FQP, LRU and MRU. Caching policy with mining approach outperforms traditional LRU and MRU caching policy due to the discovery and store of useful queries. FQP policy will cache some small-size frequent patterns whose information may have been contained by large-size frequent patterns. In a sense, FQP policy wastes some cache space. Caching system employing MFQP policy also has a good scalability as the response time for querying doesn't vary much with varying number of queries from Q10K to Q100K.

In Figure 4(b), we show the average response time for query processing with varying support of queries from 0.01 to 0.1 for Q50K query set. We can see that the varying support dose not influence query response time much. However, when the

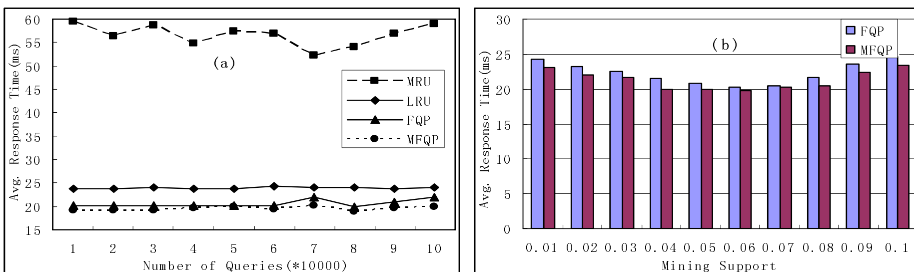

Fig. 4. Average Response Time for Querying

support is in the vicinity of 0.06, we have the minimum response time for XML querying. For a larger support less frequent queries will be mined and the cached results will be less. While for a less support the mining time will increase which will result in an additional cost for querying XML data.

Figure 5 (a) and (b) shows the hit ratio for our frequent queries caching system with varying number of queries and support respectively. From experiments we can see that MFQP policy has the highest hit ratio, which further demonstrates caching with MFQP policy outperform the other three policies.

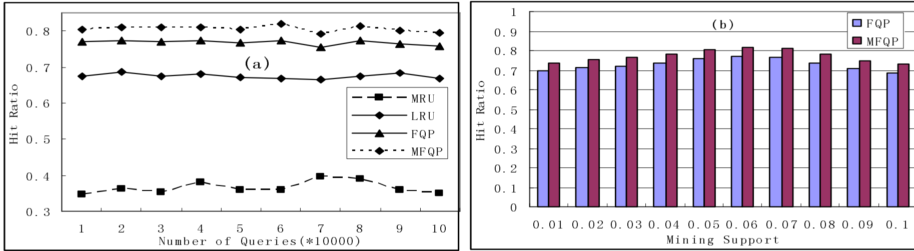

Fig. 5. Hit Ratio for Caching Queries

6 Conclusion

In this paper, we propose an approach for XML querying by caching maximal frequent query patterns. We discover the maximal frequent query patterns from user-issued queries and cache the results of such queries. We present a recursive algorithm for obtaining query results using discovered patterns. To deal with similar queries, we introduce four kinds of query relationships and employ a query rewriting process. Experiments show that our approach is efficient and outperforms the ordinal frequent patterns caching policy and traditional LRU, MRU caching policy.

References

1. Clark, J., DeRose, S.: XML path language (XPath) version 1.0 w3c recommendation. Technical Report REC-xpath-199991116, World Wide Web Consortium (1999)
2. Charmberlin, D., Florescu, D., Robie, J., Simon, J., Stefanescu, M.:XQuery: A query language for XML. W3C Working Draft. http://www.w3.org/TR/xquey (2001)
3. Hristidis, V., Petropoulos, M.: Semantic caching of xml databases. In Proc. Of the 5th WebDB (2002).
4. Chen, L., Rundensteiner, E.A., Wang S.,: Xcache-a semantic caching system for xml queries. In Demo in ACM SIGMOD (2002).
5. Yang, L. H., Lee, M.L., Hsu W.: Efficient mining of xml query patterns for caching. In Proc. of 29th VLDB (2003).
6. Chen, L., Bhowmick, S.S., Chia, L.T.: Mining Positive and Negative Association Rules from XML Query Patterns for Caching. In DASFAA (2005) 736-747.

7. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Satamoto, H., Arikawa, S.: Efficient Substructure Discovery from Large Semi-structured Data, SDM (2002) 158-174.
8. Asai, T., Arimura, H., Uno, T., Nakano, S.: Discovering Frequent Substructures in Large Unordered Trees, 6th Int'l Conf. on Discovery Science (2003).
9. Zaki, M. J.: Efficiently Mining Frequent Trees in a Forest, 8th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (2002).
10. Zaki, M. J.: Efficiently Mining Frequent Embedded Unordered Trees, Fundamenta Informaticae (2005)
11. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In Proceedings of the IEEE International Conference on Data Engineering (2002) 141-152
12. http://monetdb.cwi.nl/xml/