Exploit Sequencing Views in Semantic Cache to Accelerate XPath Query Evaluation

Jianhua Feng, Na Ta, Yong Zhang, Guoliang Li
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China
{fengjh, liguoliang}@tsinghua.edu.cn, dan04@mails.thu.edu.cn, zhangy@tsinghua.org.cn

ABSTRACT
In XML databases, materializing queries and their results into views in a semantic cache can improve the performance of query evaluation by reducing computational complexity and I/O cost. Although there are a number of proposals of semantic cache for XML queries, the issues of fast cache lookup and compensation query construction could be further studied. In this paper, based on sequential XPath queries, we propose ***fastCLU***, a **fast** **C**ache **L**ook**U**p algorithm and ***effiCQ***, an **effi**cient **C**ompensation **Q**uery constructing algorithm to solve these two problems. Experimental results show that our algorithms outperform previous algorithms and can achieve good performance of query evaluation.

Categories and Subject Descriptors
H.2.4 [**Systems**] Subjects: Query processing

General Terms: Algorithms, Performance, Languages

Keywords: XML, XPath, Query Evaluation, Semantic Cache

1. INTRODUCTION
The popularity of XML inspires the need to quickly retrieve XML data. In XML databases, a semantic cache of materialized views, which are queries combined with their result nodes, can help accelerate the process of evaluating XML queries in that when there is a cache hit, there is no need to evaluate the query against the whole database and retrieve the result from lower storage, the cached data can accomplish the task simply.

We study a group of XPath queries in XPath fragment $XP^{\{/, //, [], *\}}$, which contains four features: child axes (/),descendant axes (//), wildcards (*) and predicates ([]). There are two steps in exploiting the semantic cache of an XML database to answer queries: cache lookup and compensation query construction for evaluation. We propose algorithm *fastCLU* to accomplish the first step based on Basic Path and Predicate Condition Sets of sequential XPath queries. A view V can answer Q if there exists another query CQ which gives the result of Q when queried against the result of V. CQ is the compensation query and usually has less executing cost than Q. V is the matching view of Q. The other algorithm *effiCQ* constructs the compensation query efficiently for the second step. For example, suppose there are three views: V_1= a[[b[k<100]][j]]/f/g[c[d][.//e]], V_2=a[b/c]/u//v, V_3=a[b[k<50]]/*/x, and a query: Q_1=a[[b[k<100]][j]]/f/g[c[d][e]][h]. Q_1 can be answered by view V_1 by restricting the *e* node in V_1 to be the child of the *c* node and the output *g* node to have an *h* child. Thus compensation query CQ_1=*g/[c/e][h]*.

2. Problem Definition
Generally an XPath query can be modeled as a tree pattern composed of a node set, an edge set of child and descendant edges, a root node and an output node. To simplify the cache lookup process, we convert an XPath query into an equivalent sequential representation which has a Basic Path and a Predicate Condition Set. The **Basic Path** of an XPath query Q is the path containing all nodes from Q's root node to Q's output node. Nodes in the Basic Path the **path nodes** and other nodes are referred to as **predicate nodes**. The number of nodes in a Basic Path BP is the **depth** of BP, denoted as d_{BP}. Child and descendant axes in a Basic Path are denoted explicitly by "/" and "//".

For each path node n_{BP} of an XPath query Q, suppose there are n_c leaf nodes $\{l_{n1}, l_{n2}, ..., l_{nc}\}$, which are leaves of sub-trees of n_{BP} whose root nodes are not path nodes, we call them **predicate leaf nodes**. For all the predicate leaf nodes of n_{BP}, we construct a including n_c path expressions rooted at n_{BP} and ended at one of the n_c predicate leaf nodes, we call this set the **Predicate Condition Set** of n_{BP} and denote it as $PCSN(n_{BP})=\{pc_i \mid 1\le i\le n_c$, pc_i is a path from n_{BP} to the i^{-th} predicate leaf node of n_{BP} $\}$. The set of all of Q's path nodes' Predicate Condition Sets is the **Predicate Condition Sets** of Q and is denoted as PCSQ(Q).

The homomorphism from one query pattern to another ensures the containment relationship the other way round. In other words, for two query patterns P_1 and P_2, if there is a homomorphism from P_1 to P_2, P_2 is contained in P_1[3]. Thus a materialized view V can answer a query Q if Q is contained in V. Sequential representation of XPath queries can help reduce the time cost of homomorphism mapping checking from queries to views.

Figure 1 gives examples of tree patterns and homomorphism. The Basic Paths of P_1, P_2 and P_3 are *a/d//e*, *a/d/e/f* and *a/d/k/e* respectively. The depth of *a/d//e* is 3. There is a homomorphism from P_1 to P_2 in Figure 1(a).

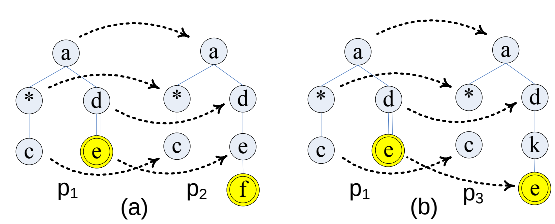

Figure 1. Homomorphism and containment of queries

Definition 1. Basic Path Containment: for two XPath queries Q_1 and Q_2, let their corresponding Basic Paths be $BP_1=n_1n_2...n_{l1}$ and $BP_2=n_1'n_2'...n_{l2}'$ respectively, BP_2 is **contained in** BP_1 if (1) $l_1\le l_2$ and (2) for any pair of symbols s_i, s_i' ($1\le i\le l_1$) at the i^{-th} position of BP_1 and BP_2 respectively, one of the following conditions is satisfied: (a) s_i'.tagName=s_i.tagName, (b) s_i="*", (c) s_i'=s_i="/", (d) s_i'="/" or "//" while s_i="//".

Definition 2. PCSN Containment: for two path nodes n_1 and n_2, $PCSN(n_1)=\{p_i \mid 1 \le i \le np_1$, np_1 is the number of predicate leaf nodes of $n_1\}$, $PCSN(n_2)=\{p_j \mid 1 \le j \le np_2$, np_2 is the number of predicate leaf nodes of $n_2\}$, $PCSN(n_2)$ is **contained** in $PCSN(n_1)$ if (1) $np_1 \le np_2$; (2) for each path expression $p=s_1s_2...s_{l1}$ in $PCSN(n_1)$, there is $p'=s_1's_2'...s_{l2}'$ in $PCSN(n_2)$, such that $l_1 \le l_2$ and p is segmented by "//"into k parts which do not contain "//" and have exactly the same occurrences in p', and the "//" symbols in p are mapped to "/", "//" or path fragments in p' between k segments.

Definition 3. PCSQ Containment: for two queries Q_1 and Q_2 $PCSQ(Q_1)=\{PCSN(n_i)|1 \le i \le d_{BP1}, n_i \in BP_1\}$, $PCSQ(Q_2)=\{PCSN(n_j') \mid 1 \le j \le d_{bp2}, n_j' \in BP_2\}$, $PCSQ(Q_2)$ is **contained** in $PCSQ(Q_1)$ if (1) BP_2 is contained in BP_1; (2) PSCN(n)=PCSN(n') for all of P_1's path nodes n except P_1's output node n_o; and (3) let n_o maps to n_o' in Q_2, $PCSN(n_o')$ is contained in $PCSN(n_o)$.

Since PCSQ containment actually requests Basic Path containment, therefore, the **criteria** of query/view answerability can be put as follows: if PSCQ(Q) is contained in PCSQ(V) for a query Q and a view V, V can answer Q. This makes the foundation of our algorithms.

3. Algorithms: *fastCLU* and *effiCQ*

FastCLU runs like this: First find a set of candidate views whose Basic Paths contain the Basic Path of the input query Q, and rank the candidate views by depth of the Basic Paths, views with greater Basic Path depth precede views with smaller Basic Path depth. Then check Predicate Condition Sets containment between Q and the current view in candidate set. If a matching view is found, this view is passed to algorithm *effiCQ* to construct compensation query. If none of candidate views contains Q, there is a cache miss and Q has to be evaluated against data in lower storage. Note that although [1] also uses string matching in cache lookup, it considers a view in the cache as a whole, and its matching process involves a time-consuming predicate condition set generation and containment test. Meanwhile, our algorithm does not require such a generate-and-test course and does not need the superset of Q's predicate conditions, which makes it more time efficient. Due to space limit, details of *fastCLU* is omitted.

EffiCQ is outlined as follows to present it clearly.

Algorithm effiCQ: compensation query construction

Input: Q, an XPath query; **V**, a matching view of Q

Output: CQ, the compensation query of Q

Let $BP_Q=n_1/(or//)n_2/(or//).../(or//)n_d$, $BP_V=n_1/(or//)n_2/(or//).../(or//)n_{dV}$

$BP_{CQ}=n_k/(or//)n_{k+1}/(or//)...\ n_{dV}/(or//).../(or//)n_{dQ}$;

/* n_k is the node before the first different axis symbol of BP_Q and BP_V if there is any, otherwise it is the output node of V */

for each path expression PE_j in $PCSN(n_{dV})$ of Q do {

if (PE_j is contained in but not equal to some path expression PE_j' of $PCSN(n_{dV})$ of V) OR (PE_j is not contained in any path expression PE_j' of $PCSN(n_{dV})$ of V)

put PE_j into $PCSN(n_{dV})$ of CQ; }

if (n_{dV} is not the output of Q)

attach the predicate conditions of $n_{i+1}, n_{i+2}, ..., n_{dQ}$ to $n_{i+1}, n_{i+2}, ..., n_{dQ}$ to CQ;

return CQ;

As presented, *EffiCQ* constructs the compensation query CQ to answer a query Q by its matching view V found by *fastCLU*. CQ is queried against V to return result of Q.

4. EXPERIMENTAL EVALUATION

We compare our algorithms with the view selection method in [1], which is based on string matching and referred to as algSM, and the naive semantic cache, which requires exact equivalence of view and query. We used a 300 MB XML document generated by the XMark [2] generator. Testing programs run in Windows 2000 system with 768MB memory.

Cache Lookup Performance. Figure 2 shows how the hit rate varies with the zipf exponent *z* used for creating attribute predicates. Hit rate of *fastCLU* is 1.29 and 7.48 times of that of algSM and the Naive Cache. This is because *fastCLU* can handle such cases that a descendant axis in Basic Path of a view is mapped to a child axis in Basic Path of a query, which algSM will treat as a cache miss.

Query Processing Performance. Figure 3 shows the average time to answer a query by the three algorithms to illustrate the speedup gained by *fastCLU* and *effiCQ*. We cached 2,000 queries and submitted 20,000 test queries and set *z*=1.75. Our strategy of caching path nodes and *effiCQ* help to enlarge the answering capacity of our semantic cache; consequently, a higher hit rate and a shorter average processing time of one query is achieved.

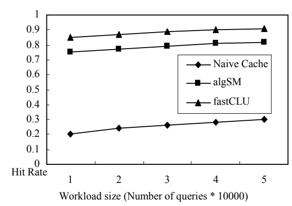

Figure 2. Hit rate vs. workload size

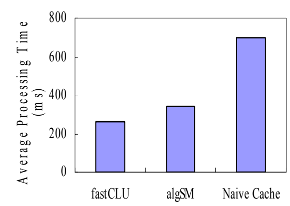

Figure 3. Average processing time

5. CONCLUSION

In this paper, we propose algorithm *fastCLU*, which uses equivalent sequential representation of XPath queries to accelerate cache lookup, and algorithm *effiCQ*, which constructs compensation queries efficiently with lower computational cost to evaluate XPath queries. Experimental results demonstrate that our algorithms can achieve high performance for query evaluation.

6. ACKNOWLEDGEMENT

This work is in part supported by the National Natural Science Foundation of China under Grant No.60573094, the National Grand Fundamental Research 973 Program of China under Grant No.2006CB303103, the National High Technology Development 863 Program of China under Grant No.2006AA01A101, and Tsinghua Basic Research Foundation under Grant No. JCqn2005022.

7. REFERENCES

[1] Bhushan Mandhani, Dan Suciu. Query Caching and View Selection for XML Databases. VLDB, 2005.

[2] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[3] Wanhong Xu, Z. Meral Ozsoyoglu. Rewriting XPath Queries Using Materialized Views. VLDB, 2005.