A Framework of Semantic Cache for Secure XML Query Answering: an Interesting Joint and Novel Perspective

Jianhua Feng, Na Ta, Guoliang Li , Yu Liu, Dapeng Lv
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China
{fengjh, liguoliang}@tsinghua.edu.cn, {dan04, liuyu-05, lvdp05}@mails.tinghua.edu.cn

ABSTRACT

Secure XML query answering to protect data privacy and semantic cache to speed up XML query answering are two hot spots in current research areas of XML database systems. While both issues are independently explored in depth, these two have not been studied together, that is, the problem of semantic cache for secure XML query answering has not been addressed yet. In this paper, we present an interesting joint of these two aspects and propose an efficient framework of semantic cache for secure XML query answering, which can improve the performance of XML database systems under secure circumstances. Our framework combines access control, user privilege management over XML data and the state-of-the-art semantic XML query cache techniques, to ensure that data are presented only to authorized users in an efficient way. To the best of our knowledge, the approach we propose here is among the first beneficial efforts in a novel perspective of combining caching and security for XML database to improve system performance. The efficiency of our framework is verified by comprehensive experiments.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *query processing*.

General Terms

Performance, Security.

Keywords

XML, Semantic Cache, Security

1. INTRODUCTION

XML has become more and more popular as the de facto standard of data representation, data exchange and storage on the Internet and in a number of applications these days. With large amount of XML data accessed and exploited by different organizations and individuals, the problem of access control over these XML data and privacy preservation of their corresponding owners has been attracting more and more attentions from the research community. How to provide information safely so that only secure data are presented to authorized user while the data intended to be hidden are not revealed? This problem makes one major discussion of this paper.

Infoscale 2007, June 6–8, 2007, Suzhou, China.

Currently, secure XML query answering is a hot joint of data security and XML databases. For a given XML document, there may be different groups of users granted different access privileges to the content of this document. In order to protect sensitive data from intended or accidentally unauthorized accesses, access control policy is used to define what elements are granted access to which users. There are challenges for secure XML querying, however. As Fan, Chan and Garofalakis pointed out in [7], firstly it is non-trivial to construct a sound and complete security view with respect to a given security policy. Secondly, for a large XML document, materializing and maintaining multiple user views may introduce expensive cost and degrade the performance of the system consequently. A third challenge is how to design effective algorithms to transform a query within the constraints of a certain access level into a safely equivalent query over the original XML document, which will affect the safety and correctness of the final result.

There are a number of recent research works [2, 3, 5, 6, 8, 9, 15] on access control of XML data, but each of them has some shortcomings. For example, [5, 6] make use of materialized views which are expensive to maintain, [8, 15] may reject certain user queries because of denial of access to some elements, [3] needs expensive integrity check due to its access annotation in each element, while [9] involves costly runtime security checks.

On the other hand, time efficiency is a critical performance requirement for the database when retrieving XML data to answer a query. This requirement has initiated the boom of research of query caching techniques, including the recent semantic cache, to help XML databases answer queries faster. A semantic cache for an XML database contains materialized views, which are evaluated XML queries combined with their corresponding result node sets. When a query Q is submitted, the cache lookup process first tries to find a materialized view which contains the result of Q. If one such view exists, there is no need to fetch data from the underlying storage, the system simply returns the part of the matching view's result which is the result of Q so that the evaluation step is completed quickly.

We choose XPath [17] as the query language in our framework. XPath is a W3C recommendation for navigating XML documents and selecting a set of element nodes. It composes the core of the more complicated XML query language XQuery [18]. We concentrate on a major subset of XPath, $XP^{\{/,\ //,\ [],\ *\}}$, which is defined as:

$$p ::= n \mid * \mid . \mid p/p \mid p//p \mid p[p]$$

$XP^{\{/,\ //,\ [],\ *\}}$ covers the child axis "/", the descendant axis "//", the predicate filter "[]" and the wildcard "*". And "n" is a tag's name,

"." represents the context node itself and can be omitted if there is no ambiguity.

Existing semantic XML query cache proposals and techniques mainly focus on XPath query caching based on tree pattern query containment relationship between the view and the query. These caching methodologies fall into two major categories: one caches XML queries as tree structures [1, 4, 19]; the other caches XML queries as strings [12]. The latter method can achieve high efficiency but is not good at handling deep XPath query trees, therefore, it does not scale very well.

Thus, the other aspect we study in this paper is the caching algorithm which follows the sequential style of [12]. Our sequential algorithm translates XPath queries into equivalent sequential forms and makes use of *Main Path containment* and *Predicate Condition Sets containment* (to be defined and discussed in detail in Section 4) to determine whether some materialized view in the cache can answer a query Q or not. Then it constructs the compensation query CQ for Q and executes CQ on the result of the matching view to finally answer Q.

In a nutshell, what we focus in this paper is efficient XPath query evaluation under secure circumstances. We propose a framework characterized by three highlights: (1) It grants access privileges to different users at the level of document DTD to ensure secure XML query answering, which protects sensitive data at a more abstract level; (2) It caches the submitted queries for each user group separately to speed up query evaluation; (3) Moreover, it takes into consideration the mutual relationship between different user access levels, which may be containment, overlapping or disjointness, to share cached data among different groups to further utilize the cache. In other words, our algorithm not only explores the probability of reuse cached data within one group of users, but also considers data sharing among different user groups with different privileges.

The main contributions of our work include:

- We conduct one of the first research works trying to combine the issues of semantic cache for XML query and secure XML query answering together to provide a safe and prompt query answering mechanism.
- We propose a novel framework of semantic cache for secure XML query answering, in which the concepts of *safe path*, *user DTD path set* and *user DTD path sets hierarchy* are applied to ensure the safety of sensitive data.
- We design an efficient semantic cache architecture which takes into consideration cache organization, fast cache lookup, compensation query construction for query evaluation under the secure circumstance.

The remainder of the paper is organized as follows. Section 2 reviews related work. Overview of the framework, preliminaries and definitions are given in Section 3. Section 4 illustrates the algorithms in detail. Section 5 gives experimental analysis of performance of the framework and Section 6 concludes the paper.

2. RELATED WORK

2.1 Secure XML Query Answering

[8] and [15] both consider general access control policy for XML documents, not only XML queries. The decision procedure determines whether to grant access of the required data or not upon submission of a request. If there are sensitive data in the result of a user query, this query is denied. [5, 6] study access policy for XPath queries where a particular view is assigned to each user. Such views are constructed by an algorithm based on tree labeling. [9] answers a query by checking each element and returning the accessible ones. A static optimization algorithm checks the safety of a query, where a safe query is a query which only returns accessible elements. In contrast, unsafe queries still have to go through run-time security check.

Different from these general-purposed methods, [3] mainly concentrates on secure answering of twig queries and proposes a multi-level security model. Annotations of accessibility are attached to elements, and only elements labeled as accessible can appear in the result. But the fine granularity of secure annotation at the element level also introduces some problems. For instance, when adding new nodes to the XML document, it is rather expensive to define and maintain the secure policy. To solve the problem of costly materialized view maintenance, [7] proposes an algorithm for deriving security views for different user groups from security policies at the DTD level. Algorithms for XPath query rewriting and optimization are developed to answer queries without materializing the views.

2.2 Semantic XML Query Caching

A related work of exploiting semantic cache to accelerate XML query answering is query rewriting which in turn depends on query containment estimation between views and queries. XPath query containment has attracted a lot of attention these years [10, 13, 16]. [1, 4, 19, 12] propose some frameworks and prototypes of organizing semantic cache in XML databases. [4] studies caching the result of XQuery queries at the client side. The cached views have smaller results because of the characteristic of the XQuery queries, but optimizing the lookup process is harder accordingly. When there are a large number of views in the cache, lookup becomes the bottleneck. In [19], frequently submitted tree pattern queries are mined and maintained to answer new queries. [1] materializes XPath queries into views containing XML fragments, data values, full paths and node references to assist query processing. However, this work does not address the problem of speeding up the cache lookup process when the cache has a large number of views. Exact containment is used in matching queries and views. [12] gives the state-of-the-art algorithm to achieve fast cache lookup. By regulating XPath queries and views into strings, the lookup process is simplified into efficient string matching to find appropriate views.

3. OVERVIEW OF THE FRAMEWORK AND PRELIMINARIES

In this section we present a whole image of our secure query caching framework and give definitions and theorems which form the foundation of this paper.

3.1 Overview of the Cache Framework

The cache framework runs in four major steps: (1) Accept a user query Q, check the safety of Q according to the user DTD path set D_u to determine if Q needs rewriting, that is, whether the result node set or the predicate condition set of Q contains some inaccessible elements; (2) If rewriting is needed, record inaccessible sub-elements of the output node, which will be used in the final evaluation step; refine predicate conditions, so that the

terminal node of each predicate condition is an accessible element to the current user, this step generates a safe query Q_s; (3) Execute cache lookup, if there is some materialized view which can answer Q_s, construct the compensation query Q_c for Q_s(a **compensation query** is executed on the result of the matching view and returns the exact result of the original query executed on the database); (4) Evaluate Q_c by the cache if there is a matching view or evaluate Q_s by the underlying storage, filter out all inaccessible sub-element of the output node in the result to give the answer of Q.

There are three major structures in our framework. One is the user DTD path set hierarchy, which is organized as a tree and composes of all the user DTD path sets. A user's join and leaving processes map to appending and deleting a user DTD path set node in the hierarchy. When a new user joins the system, the user DTD path set D_u is inserted to an appropriate position in the hierarchy so that its parent node D_p contains all the safe paths of D_u and no child node of D_p satisfies this condition. The leaving of a user is processed in a similar way, with all the child nodes of D_u changed to children of D_p. In this way the user management mechanism is very flexible.

The second structure is the semantic cache containing **materialized views** which are evaluated XPath queries with result node sets. The cache is partitioned into n_u parts, one for each of the totally n_u users. Since the cache has limited space to share among all users, it has to be allocated reasonably. We use weight values to guarantee this. A weight is assigned to a user according to the position of his/her DTD path set node in the hierarchy. Users at higher levels get greater weights and accordingly more cache space. Before user queries are actually answered, the system runs a warm-up process to generate a bunch of potentially frequently submitted queries and load their result node sets into the cache for later query answering. In the cache lookup step, if there is no matching view of a query in its user's caching part, we resort to the hierarchy to find this user's parent user U_p or ancestor user U_a in a bottom-up manner, and check whether their caching parts contain a matching view or not. If there is a matching view, we have a hit and this view is used to answer the query; otherwise, this query is evaluated against the underlying database storage.

The third structure is the blocking set, which records at runtime all the inaccessible sub-elements of the output node of a query. Blocking set is used at the last step of query processing to filter out sensitive data in the result set to ensure the security of the XML data. We will give an example of blocking set later in this section.

3.2 Document Type Definition, Safe Path (Set)

A DTD (Document Type Definition) defines the structure of an XML document with a list of legal building blocks composed of simple blocks such as *element*, *attribute*, *entity*, *PCDATA* and *CDATA*, etc. We model a **DTD graph** as a triple of a root element, a node set and an edge set, *D=(r, V, E)*, which is a variant of finite node-labeled DAG [3] such that: (1) *r* is a special node which defines the root element of XML documents conforming to this DTD; (2) each node in *V* is labeled by an element tag; and (3) each edge in *E* has a label from the set {+, *, ?, 1}.

An element *v* in *V* may contain attributes to which we grant the same accessibility as *v* for simplicity. An edge *e* in *E* connects an element to one of its sub-elements. The number of a certain sub-element can be "one or more" with *e* labeled "+", "zero or more" with *e* labeled "*", "zero or one" with *e* labeled "?" or "exactly one" for the "1" edges whose labels are usually omitted. We give an example DTD in Figure 1. A company document conforming to this DTD consists of a group of *projects* and the *staff* which includes the *manager* and the *employee*. One *project* has a *manager* who can access its three sub-elements, namely, *p-id*, *duration* and *budget* while an *employee* only knows the *p-id* and the *duration* of the *project* he/she is working for.

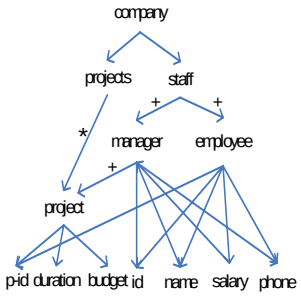


Figure 1. Example DTD

We define user access privilege by the concept of *User DTD path set* on the DTD level to grant access of elements in XML documents to different user groups. There are three kinds of relationship among different users' DTD path sets, namely containing, overlapping and disjoint, which are used for query containment estimation in the cache lookup process to efficiently reuse cached data for different users.

Definition 1. (**User DTD**) A user DTD $D_u=(UID , r, V_u, E_u)$ is a subset of the original DTD graph *D* in which (1) *UID* is the unique identifier for a user; (2) *r* is the same root element as the root of *D*; (3) V_u is a subset of *V* and contains the accessible elements of this user; and (4) E_u is a subset of *E* where edges connect elements in V_u. □

Figure 2 gives an example of user DTD for users of the *employees* type.

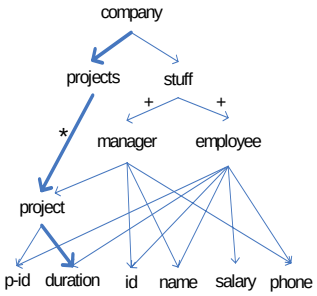


Figure 2. User DTD for employee

User DTD is checked to return accessible data and hide private information in query evaluation step. To simplify the process, we devise a sequential representation of user DTD graph. The sequential representation is composed of a set of *safe paths* which help rewrite user queries into safe queries. The checking and

rewriting algorithms will be presented and discussed in detail in Section 4.

Definition 2. (**Safe Path**) A safe path *sp* in a user DTD D_u is a path from e_1, the root element of D_u, to a deepest accessible element e_k granted to a certain user, *sp* is in the form of $e_1/e_2.../e_k$, where e_i $(1 \le i \le k)$ is the i^{-th} element along *sp*. □
For example, path "company/projects/project/duration" whose edges are marked bold in Figure 2, is a safe path in the user DTD for *employee*. Definition 2 implies the following property:

Property 1. The parent element of an accessible element is also accessible, but the sub-elements of an element may have different accessibility, that is, some sub-elements may be accessible while others may be not. □

Definition 3. (**User DTD Path Set**) A user DTD path set, denoted as $\{p \mid p \in D_u\}$, is a set of all safe paths in a user DTD D_u and *p* is any safe path for a certain user. □
For instance, user DTD path set for *employee* in Figure 2 is {company/projects/project/p-id,company/projects/project/duration, company/staff/manager/id, company/staff/manager/name, company/staff/manager/phone, company/staff/employee/id, company/staff/employee/name, company/staff/employee/phone, company/staff/employee/salary}.

For estimating the query containment relationship in the cache lookup step we propose a hierarchy to organize all the user DTD path sets. First we assign full access to a root user whose user DTD D_r equals to the original unrestricted DTD. User DTD path sets of all the other users are subsets of D_r. The idea of reusing cached data is like this: if user u_1, whose DTD path set DPS_1 contains DTD path set DPS_2 of user u_2, then the cached view results for u_1 have the potentiality to answer queries submitted by u_2. We will discuss this in detail later in Section 4.

Definition 4. (**User DTD Path Sets Hierarchy**) A user DTD path sets hierarchy $H=(H_r, \{(PS_p, PS_c)\})$ is composed of a root element H_r which is the root user's DTD path set and a set of tuples (PS_p, PS_c) declaring the containing relationship between a parent user DTD path set PS_p and a child user DTD path set PS_c. *Containing* implies that for each safe path sp in PS_c, sp has an instance in PS_p. □
Suppose a manager has the root user's privilege, let his user DTD path set be D_m, and there is a human resource specialist user who can access the *name* and *phone* sub-elements of *manager* and *employee*, whose user DTD path set D_s= {company/staff/manager/name, company/staff/manager/phone, company/staff/employee/name, company/staff/employee/phone}. Then the user DTD path sets hierarchy for *manager*, *employee* and *specialist* is $(D_m, \{(D_m, D_s), (D_m, D_e)\})$ as depicted in Figure 3.

3.3 Cache-based XPath Query Containment Estimation and Answering

We model XPath queries to be tree-structured patterns, which is denoted as a tree (V_p, E_p, r_p, o_p) over $\Sigma \cup \{*\}$, where: (1) V_p is the set of vertices, E_p is the set of edges and Σ is the set of all tag names; (2) $r_p \in V_p$, $o_p \in V_p$, are the root and output vertices of the tree pattern respectively; (3) each vertex *v* has a label in $\Sigma \cup \{*\}$; and (4) an edge *e* can be either a child edge "/" or a descendant

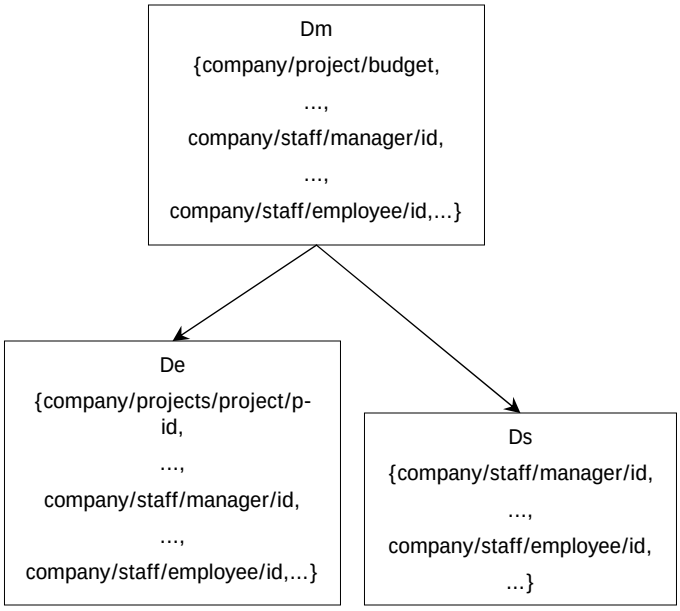
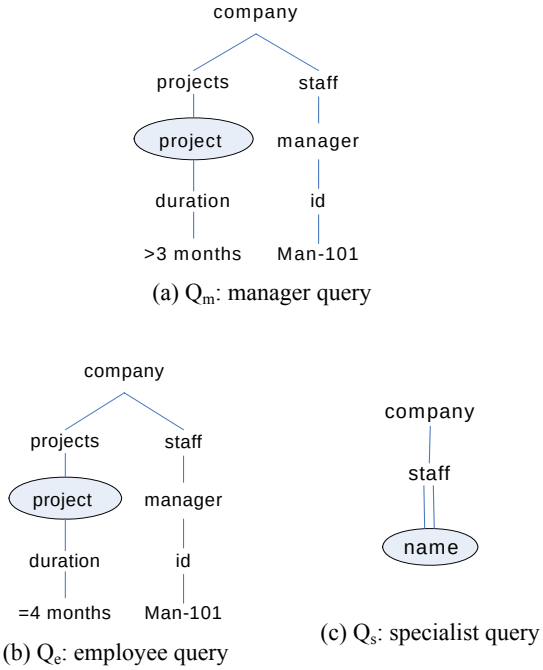


Figure 3. User DTD Path Set



(a) Q_m: manager query

(b) Q_e: employee query

(c) Q_s: specialist query

Figure 4. Example XPath Tree Patterns

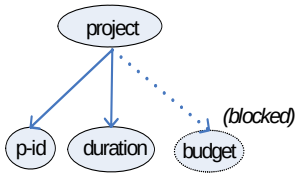


Figure 5. Accessibility for different sub-elements

edge "//", representing the parent-child or the ancestor-descendant relationship between two vertices respectively.

Figure 4 gives examples of XPath queries. Suppose a manger with id "Man-101" asks for all the *project* elements with a duration longer than three months that he/she takes charge of, the corresponding query is: company[staff/manager[id="Man-101"]]/projects/project[duration>3months] as Q_m shown in Figure 4(a), where output nodes are marked by ellipses.

Here we give an example of blocking sub-elements of the output node in Figure 5, where accessible elements are labeled by solid circles while inaccessible ones by dotted circles. Suppose an employee submits a query with element *project* being the output node. From the user DTD of an employee E, E knows that a *project* has *p-id* and *duration* sub-elements, but E has no idea that a *project* has a *budget* sub-element. So the final result to such a query should not contain any *budget* element. To achieve such goals, algorithm 3 is proposed to block inaccessible data to ensure safety of the final result. Compared with the use of dummy elements in [7], refining the result of a query is semantically more clear and straightforward than presenting dummy elements in the query result.

In the cache-based query rewriting and answering process, query pattern containment estimation is an important issue. If a query is contained in some materialized view in the semantic cache, we have a hit and only need to construct the compensation query to answer this query by cached result of the matching view. Since it is expensive to determine the containing relationship between tree-structured patterns while containment test of sequences is less costly and more efficient, we transform tree patterns into an equivalent sequential representation defined in Definition 5. For more detailed studies on XPath containment please refer to [10, 13, 16].

Definition 5. (**Sequential XPath**) A sequential XPath S_{XP} is an equivalent representation of an XPath query Q, $S_{XP}=(MP, \{(n_i, PCS_i) \mid 1 \le i \le l\})$, where (1) *MP* is the Main Path of Q which is the path from the root to the output element; n_i is the i^{-th} node of MP, l is the number of nodes along MP; (2) PCS_i is the Predicate Condition Set of n_i which includes all the sequential predicate conditions of n_i; (3) a Predicate Condition of a path node n_i is a sequence which starts from n_i and ends at one of n_i's non-output leaf nodes. □

For instance, sequential XPath of the query in Figure 4(a) is (company/projects/project, {(company, {company/staff/manger/id[="Man-101"]}), (projects, Φ), (project, {project[duration>3months]})}), where Φ is the empty set. Specifically, company/staff/manger/id[="Man-101"] is a predicate condition for path node *company*, {project[duration>3months]} is the predicate condition set for path node *project* and this set contains only one predicate condition.

Definition 6. (Single Path Containment) A single XPath $S_1=n_1n_2...n_k$ is **contained** in a single XPath $S_2=n_1'n_2'...n_l'$ if: (1) $k \ge l$; and for $1 \le i \le l$: (2) if n_i and n_i' are tags, they have the same tag name or n_i'="*"; (3) if n_i and n_i' are axis symbols, both are "/" or n_i="/" or "//" while n_i'="//"; (4) if n_i and n_i' are value expressions, condition in n_i has equal or stricter limit than in n_i'. □

For example, single path S_1=a/b[d>60]/c//e is contained in S_2=a//b[d>50]/c.

Definition 7. (**Sequential XPath Containment**) A sequential XPath $S_1=(MP_1, \{(n_i, PCS_i)\})$ is contained in another sequential XPath $S_2=(MP_2, \{(n_i', PCS_i')\})$ if (1) MP_1 is contained in MP_2; (2) each predicate condition in PCS_i of n_i is contained in a predicate condition in PCS_i' of n_i'. □

Definition 7 gives the guideline to determine whether materialized views in the cache can answer a query or not. Accordingly, one can prove the following theorem:

Theorem 1. (**View/Query Answerability**) A query Q can be answered by a view V if the sequential XPath of Q is contained in sequential XPath of V. □

4. ALGORITHMS OF THE SECURE CACHE FRAMEWORK

In this section we present the algorithms used to implement the cache framework and illustrate the whole query answering process in detail. Note that the problem of secure XML query answering is different from general-purposed query answering in that the user might ask for inaccessible data. So we take a best-effort strategy, that is, the inaccessible sub-elements of a query's output node are removed from the result set.

4.1 Algorithm 1: Security Test and Safe Query Generation

Security test is carried out firstly to ensure the safety of the query result. In this step main path and predicate condition sets of a sequential XPath query are examined and processed in different manners. Inaccessible sub-elements of the output node are recorded in the blocking set with main path unchanged. Meanwhile, predicate conditions are refined to ensure that each predicate condition's terminal node and all of its sub-elements are accessible. The sequential XPath query is compared with the user DTD path set D_u. These processes are sequence set containment test. This algorithm is shown in Figure 6. It runs within $O(n^3)$ time bound, where n is the total number of nodes in the query.

4.2 Algorithm 2: Cache Lookup and Compensation Query Construction

The view/query answerability ensures that if the sequential XPath of a view V contains a sequential XPath query Q, Q can be answered by the cached result of V. The cache lookup process runs in a bottom up manner: caching part of the user, the parent and ancestor DTD path sets in the hierarchy are checked one by one until a matching view is found or until the root user's DTD path set is reached without a matching view. Figure 7 depicts the algorithm which runs within $O(n^4)$ time bound.

We give an example here. Recall Figure 4, suppose Q_m is evaluated and cached as V_m in the caching part for manager. If a specialist user submits Q_s as shown in Figure 4(c) and there is no view in the caching part for specialist containing Q_s, as the hierarchy in Figure 3 implies, the manager's user DTD path set is the parent set of that of the specialist, so we go on to check if the cached result in the caching part of manger has a matching view of Q_s. If there is still no matching view, a cache miss happens and Q_s has to be evaluated using data in underlying storage.

4.3 Algorithm 3: Secure Query Answering

The running results of the previous two algorithms indicate whether a query can be answered by the cache. We discuss the case of cache hit. Now the known information includes the compensation query, the caching part containing the matching

view and the blocking set for the output node, the evaluating algorithm can be devised accordingly as shown in Figure 8. This algorithm has polynomial time complexity.

Alg. 1. Security Test and Safe Query Generation

Input: UID (user id), S_{XP} (sequential XPath), H (user DTD path set Hierarchy)

Output: Q_s (safe query in the form of sequential XPath), BS (blocking set)

BEGIN

Let n_o be the output node of S_{XP}, D_r be root user's user DTD, S_{XP}.MP be main path of S_{XP};

1. Find the user DTD path set D_u in H, whose user id is UID;
2. If (#(safe paths containing n_o in D_r) > #(safe paths containing n_o in D_u))
   /* there are some inaccessible sub-elements of n_o for this user */
3. Put sub-elements of n_o in D_r which do not appear in D_u as sub-elements of n_o into BS, Q_s.MP = S_{XP}.MP;
4. For predicate condition set of every node n_{MP} on S_{XP}.MP {
5. For each predicate condition $p=t_1t_2...\mathbf{t_p}$ with terminal node t_p {
6. If (#(safe paths containing t_p in D_r) ≠ #(safe paths containing t_p in D_u))
7. For each safe path $sp=n_1n_2...n_i\mathbf{t_p}n_j...n_k$ containing t_p in D_u do {
8. $f=n_j...n_k$, $p_r=p+f=t_1t_2...\mathbf{t_p}n_j...n_k$, put p_r into predicate condition set of n_{MP} in Q_s; } } }
9. Return Q_s and BS.

END

Figure 6. Algorithm 1. Security Test and Safe Query Generation

Alg. 2. Cache Lookup and Compensation Query Construction

Input: UID (user id), Q_s (sequential safe query generated by Alg. 1), H (user DTD path set Hierarchy), SC (semantic cache)

Output: Ans (boolean flag indicating whether Q_s can be answered by SC), CPtr (pointer to the caching part containing the matching view), Q_c (compensation query)

BEGIN

1. Ans = FALSE; let CPtr point to caching part of the user whose id is UID;
2. While (Ans = = FALSE) do {
3. MV = CL(Q_s, CP); /* Call procedure 1 to do cache lookup and find the matching view MV */
4. If MV ≠ NULL /* A matching view is found */
5. Ans = TRUE;
6. Q_c = CQC(Q_s, MV); /* Call procedure 2 to produce the compensation query Q_c*/
7. Else If CPtr does not point to the root user's DTD
8. Let CPtr point to the caching part of current user's parent user;
9. Else /* there is no matching view for Q_s in the whole cache */
10. Q_c = NULL, CP = NULL;}
11. Return Ans, Q_c and CP.

END

Procedure 1. Cache Lookup CL(Q, CP)

Input: Q (a sequential XPath), CP (a caching part to search)

Output: V (matching view of Q, NULL if there is none)

Begin

1. V = NULL;

2. Find a view V whose main path contains the main path of Q;
/* "contains" conforms to Definition 7 */
3. If V ≠ NULL {
4. If the predicate condition set of V contains the predicate condition set of Q
/* "contains" conforms to Definition 7 */
5. Return V;
6. Else If there are views in CP which are not checked
7. Go to 2; }
8. Return V.

End

Procedure 2. Compensation Query Construction CQC(Q, V)

Input: Q (a sequential XPath query), V (a matching view of Q)

Output: Q_c (compensation query of Q with respect to V)

Begin

1. Let MP_c be the main path of Q_c, $MP_c = Q.MP$;
2. Let $V.MP = n_1n_2...n_{vo}$, $MP_c = n_1n_2... n_{vo}...n_m$;
3. For ($n_i = n_{vo}$ to n_m in MP_c) do {
4. Let $PCSN_C(n_i)$ be predicate condition set of n_i in Q_c, $PCSN_C(n_i) = \Phi$;
5. For (each predicate condition p_j in $PCSN_Q(n_i)$ in Q)) do {
6. If NOT ($p_j \in PCSN_C(n_i)$)
7. $PCSN_C(n_i) = PCSN_C(n_i) \cup \{p_j\}$; } }
8. Return Q_c.

End

Figure 7. Algorithm 2. Cache Lookup and Compensation Query Construction

Alg. 3. Secure Query Answering

Input: Q_c (compensation query), V (matching view), BS (blocking set)

Output: RS (result set of Q)

BEGIN

1. Evaluate Q_c on the result of V, get RS;
2. If BS ≠ Φ
/* there are inaccessible data in the current result set*/
3. Delete sub-elements of the output node in RS which appear in BS;
4. Return RS.

END

Figure 8. Algorithm 3. Secure Query Answering

5. EXPERIMENTAL STUDY

We have conducted a performance study of the efficiency of our cache framework on a 300MB XML document generated by the XMark [14] generator using the DTD depicted in Figure 1. The algorithms were implemented in Java and were run on an AMD Athlon 2000+ computer running Windows 2000 with 768MB memory. We used three different user characters, namely, manager, employee, human resource specialist as introduced in Section 3. The testing queries were generated by a carefully designed procedure to ensure that elements appear at a reasonable frequency.

Experiment 1. This experiment compared three cache policies for the manager user group, namely, (1) no cache(NC), (2) simple cache(SC) where only exact sequence equality is considered as cache hit, and (3) our secure semantic cache(SSC), in which an XPath query is composed of a Main Path and several Predicate

Condition Sets. We give the average query processing time of the manager user in Figure 9.

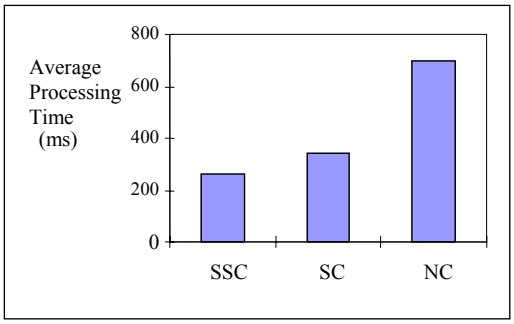


Figure 9. Average Query Processing Time of Three Caching Mechanisms

Experiment 2. This experiment compared two cache policies namely, (1) no hierarchy and exact sequence equality for cache hit(NoH) and (2) SSC for three different user groups. Figure 10 shows the average query evaluation time for different users.

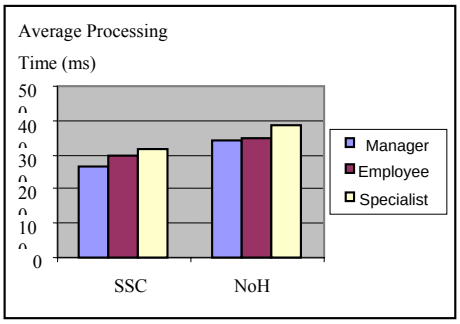


Figure 10. Average Query Processing Time of Two Caching Mechanisms for Different User Groups

6. CONCLUSTION

What we would like to emphasize in this paper is that performance improvement is everywhere in database systems, and a study of semantic cache techniques under secure query answering circumstance provides a good example of accomplishing such goals in a particular application environment. We have proposed a framework of semantic cache for secure XML query answering. Such an attempt is beneficial in search of new research directions and application fields. The framework explores the joint of the semantic caching technique and the secure constrains. Specifically, by combining together the two-folds, requested accessible data can be retrieved quickly while sensitive data are protected from unauthorized access. Theoretical and experimental analysis both prove the efficiency of our algorithms.

7. REFERENCES

[1] A. Balmin, F. Ä Ozcan, K. S. Beyer, R. J. Cochrane and H. Pirahesh. *A Framework for Using Materialized XPath Views in XML Query Processing.* In VLDB, 2004

[2] E. Bertino and E. Ferrari: Secure and Selective. *Dissemination of XML Documents.* TISSEC, 5(3):290–331, 2002

[3] S. Cho, S. Amer-Yahia, L. Lakshmanan, and D. Srivastava. *Optimizing the Secure Evaluation of Twig Queries.* In VLDB, 2002

[4] L. Chen, E.A. Rundensteiner. *ACE-XQ: A Cache-aware XQuery Answering System.* In WebDB, 2002

[5] E. Damiani, S. di Vimercati, S. Paraboschi and P. Samarati. *Securing XML Documents.* In EDBT, 2000

[6] E. Damiani, S. di Vimercati, S. Paraboschi and P. Samarati. *A Fine-grained Access Control System for XML Documents.* TISSEC, 5(2):179–202, 2002

[7] W. Fan, C. Chan and M. Carofalakis. *Secure XML Querying with Security Views.* In SIGMOD, 2004

[8] S. Hada and M. Kudo. *XML Access Control Language: Provisional Authorization for XML Documents.* http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html

[9] M. Murata, A. Tozawa and M. Kudo. *XML Access Control Using Static Analysis.* In CCS, 2003

[10] G. Miklau and D. Suciu. *Containment and Equivalence for an XPath Fragment.* In PODS, 2002

[11] G. Miklau and D. Suciu. *Controlling Access to Published Data Using Cryptography.* In VLDB, 2003

[12] B. Mandhani and D. Suciu. *Query Caching and View Selection for XML Databases.* In VLDB, 2005

[13] F. Neven and T. Schwentick. *XPath Containment in the Presence of Disjunction, Dtds and Variables.* In ICDT, 2003

[14] 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.

[15] Oasis. *eXtensible Access Control Markup Language (XACML).* http://www.oasis-open.org/committees/xacml/repository/

[16] W. Xu and Z. M. Ozsoyoglu. *Rewriting XPath Queries Using Materialized Views.* In VLDB, 2005

[17] XPath2.0: http://www.w3.org/TR/xpath20/

[18] XQuery 1.0: http://www.w3.org/TR/xquery/

[19] L. Yang, M. Lee and W. Hsu. *Efficient Mining of XML Query Patterns for Caching.* In VLDB, 2003