SQS: A SECURE XML QUERYING SYSTEM

Wei Li and Cindy Chen
Department of Computer Science, University of Massachusetts Lowell, One University Ave, Lowell, MA, U.S.A.

Keywords: XML, access control, cache, query.

Abstract: Contemporary XML database querying systems have to deal with a rapidly growing amount of data and a large number of users. As a consequence, if access control is used to protect sensitive XML data at a fine-grained level, it is inefficient when it comes to query evaluation, since it is difficult to enforce access control on each node in an XML document when the user's view needs to be computed. We design and develop a secure XML querying system, namely SQS, where caching is used to store query results and security information. Depending on whether there is a cache hit, user queries are rewritten into secure system queries that are executed either on the cached query results or on the original XML document. We propose a new cache replacement policy LSL, which updates the cache based on the security level of each entry. We also demonstrate the performance of the system.

1 INTRODUCTION

XML has been widely used for data sharing and exchange over the internet. As a consequence, XML data security has become an important concern. In general, there are three major issues of data security: authentication, authorization and access control. For access control, there are two major approaches: discretionary access control and mandatory access control. Under discretionary access control policy, a user may "grant" *read* or *modify* privilege to another user; or "revoke" another user's privilege. The mandatory access control mechanism uses system wide policies, and thus cannot be changed by any user.

Since native XML data do not have a normalized structure, unlike retrieving relational data and publishing them on the web, querying XML databases is intrinsically complicated. Despite many efforts by researchers, it remains a question how to evaluate XML queries efficiently while enforcing access control. We describe and implement a secure XML querying system called SQS. The system uses a cache to improve query execution. Unlike the traditional caching policy that caches only the query results, the SQS system also caches the security information. To the best of our knowledge, the SQS system is the first XML querying system that uses caching for security information. When a query Q is executed, the cache is looked up first to check whether there are cached query results that answer Q, while the system's access control rules are also complied with. If a cache hit occurs, a composing query C is computed and applied to the cached query results to render the result for Q. Otherwise, Q is rewritten into a secure query Q′ and evaluated over the original XML document.

2 RELATED WORK

A number of XML access control models have been proposed. Based on the components of XML documents, Bertino et al. (Bertino et al., 2000) suggested using different protection object granularities such as document/DTD, set of documents, attribute, (sub)element and link. Damiani et al. (Damiani et al., 2002) defined *authorization strength* in their access control model. If there is a conflict between the authorizations specified in an XML document and those specified in its corresponding schema, those with a stronger authorization strength will have a higher priority so to override the others. In Gabillon's model (Gabillon et al., 2002), access control list is used to specify authorization. However, since it is associated with each node, this method suffers when dealing with large XML documents.

Researchers have developed many techniques for XML access control. In (Cho et al., 2002), Cho et al. proposed a method to rewrite a user query with specified authorizations into an authorized query, therefore avoid unnecessary authorization check-ups. In (Yu et al., 2002), Yu et al. introduced a method called CAM to enforce XML access control. The compression by CAM is achieved by taking advantage of the tree structure of XML database and grouping together the nodes that have similar accessibility, on a per-user basis. In (Fan et al., 2004), Fan et al. developed a view-based XML security model. Access specifications are enforced during the process of deriving the *security view*, where a security view is based on the user view DTD and a function defined via XPath queries. In (Mandhani and Suciu, 2005), Mandhani and Suciu proposed a novel technique to cache XPath views for XML. An algorithm was designed to determine whether a view can answer a new query. When it does, a composing query will be computed and applied to the result of the view to answer the new query.

3 SYSTEM ARCHITECTURE

The SQS system uses a multi-level access control model. The security levels are specified using non-negative integers. The access control rule is simple, yet robust, which requires that the security level of the query should be no less than that of the data in order to be granted access. The access control specifications are defined at the document level. In the XML documents, attribute `SecurityLevel` is added to each element to indicate its security level. In an XML tree, the security level is only allowed to increase monotonically from the root to a leaf, which means that no element has a higher security level than any of its descendants. The system requires that the security level of each node should be annotated. Although this will bring the needs for additional space and extra work by the document owner, it would be acceptable if secure query evaluations can be improved.

We choose to use XPath query language to navigate down the XML tree and retrieve data. The system recognizes the XPath queries defined by the following grammar:

$$p \ ::= \ e \mid p/p \mid p//p \mid p[q]^* \qquad (1)$$

e is an element name, and predicate *q* can be equalities, comparisons, or paths.

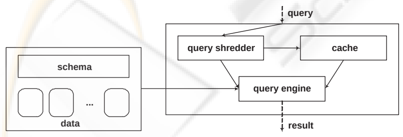

Figure 1: The SQS system architecture.

Figure 1 illustrates the architecture of the system. At first, a new query Q is parsed by the query shredder. Then, the cache will be looked up to check whether there is a cached view V that answers the query. If a cache hit, a composing query C is computed based on Q and V, then C is evaluated on the result of V by the query engine; if a cache miss, Q is rewritten into a secure query Q′. Then Q′ is sent to the query engine and evaluated on the XML document.

4 METHODS

4.1 Query Rewriting

For a non-secure XPath query Q issued by a user at security level *n*, we rewrite it into a secure one by adding a predicate `[@SecurityLevel <= n]` to each node in Q. However, since in our access control model, the security level increases monotonically along a path, we may only need to add the above predicate to each node appearing after the last axis in the XPath query.

In the system, if the original user query cannot be answered using the cached views, it will be rewritten as above and sent to the query engine; if a cached view can answer the query, the composing query is then rewritten and evaluated on the cached query result.

4.2 Cache Structure

In the system, a cache is used to store materialized views and their security information. The cache is maintained using RDBMS. We selectively adopt the cache tables in (Mandhani and Suciu, 2005). We also add columns to indicate query security level, result information, as well as auxiliary query information that is used for cache replacement. Therefore, the cache of the SQS system consists of the following two tables:

- *View(viewID, prefix, predicate, all-predicates, comparison-tags, query-security-level, result-flag, usage-count, timestamp)*
- *XMLData(viewID, data)*

The columns *viewID, prefix, predicate, all-predicates, comparison-tags* provide the same information as in (Mandhani and Suciu, 2005). *Query-security-level* stores an integer which is the security level of the query issuer. *Result-flag* is *TRUE* or *FALSE*, which is an indicator of whether there is non-empty result associated with the view. *Usage-count* records the number of times that this view has been used to answer new queries. *Timestamp* indicates the time when the view is last used.

4.3 Cache Lookup

To process a query, we need to find a view in the cache that not only answers the query, but also complies with the access control rules. Note that there might be a few views in the cache that satisfy the two constraints. To select the best cached view to answer query Q at security level S, we develop the following algorithm based on (Mandhani and Suciu, 2005):

```
**cache-lookup**(Q, S)
for $k = n, \ldots, 1$
  EXEC SQL
    SELECT      View.*
    FROM        View
    WHERE       View.prefix = Prefix(Q, k)
    AND         (View.predicate = null
                 OR
                 View.predicate in ConPreds(Q, k))
    AND         View.query-security-level >= S
    ORDER BY    View.query-security-level ASC
  examine the returned views in its order;
  if some view answers Q, return it and exit;
return null
```

Prefix(Q, *k*) is the query left after removing from Q the predicates of the *k*th axis node and all the axis nodes after. ConPreds(Q, *k*) is the concatenation of all the normalized trees that can map to one of the predicates of the *k*th axis node of Q.

In our system, it implies that if user security level $S_1 \geq$ user security level S_2, then the result of executing the same query at S_1 will contain all that at S_2. Thus, the predicate *View.query-security-level* $>=$ S filters out those views that would not answer Q even though by their semantics they could. The returned views are also sorted since the best view that answers Q is the one whose *query-security-level* is greater than that of Q and is also the closest to that of Q.

4.4 Cache Warmup

In our system, we do not use any particular warmup policy to populate the cache. The system automatically inserts the newly processed query with its result into the cache until the cache's upper limit is reached.

4.5 Cache Update

Insertion happens when a query is processed and there is still room in the cache. The query result will be inserted into table *XMLData* and the system will automatically assign a unique key for the field *viewID*. This key is also returned and used in table *View*. The fields *prefix, predicate, all-predicates* and *comparison-tags* in table *View* are computed according to (Mandhani and Suciu, 2005). *Query-security-level* will be filled with the security level value of the query issuer. *Result-flag* will be *TRUE* if the query result is non-empty or *FALSE* otherwise. *Usage-count* is set to "1". Finally, *timestamp* is assigned the current system time.

Practically the cache has limited storage. When it is full, it has to eject a certain entry to make room. Our system supports three cache replacement policies:

- **LFU (Least Frequently Used)**:
  The view that has the smallest *usage-count* will be ejected from the cache. In case there is a tie, the system will randomly choose one view to discard.
- **LRU (Least Recently Used)**:
  The system selects from the cached views the one that is least recently used to eject. The auxiliary column *timestamp* serves here to indicate which view has the oldest timestamp value.
- **LSL (Lowest Security Level)**:
  The view that has the lowest security level value will be chosen to eject.

Unlike LFU and LRU, LSL is special for our system. Intuitively, the view at a high query security level may potentially be able to answer many queries that have a lower security level, as long as semantically this view answers those queries.

Besides insertion and replacement, update happens to the view chosen to answer the new query. It includes increasing the *usage-count* field by "1" and assigning *timestamp* the newest system time value.

5 EXPERIMENTS

5.1 Experimental Setup

This section assesses the performance of the SQS system. The system is implemented in Java. All experiments are performed on a 2.80GHz Intel Pentium IV machine with 512 MB of RAM running Linux. All experimental XML documents are generated using XMark. XPath queries are also generated based on the DTD of the documents. The cache of the SQS is maintained by Apache Derby DB. Galax XQuery processor is used to process the XPath queries.

5.2 Results and Analysis

Effect of Cache Size on Cache Hit Rate

As shown in Figure 2, when there are relatively fewer cache entries, LSL gives the best cache hit rate, while LFU the worst. As the cache size increases, the cache hit rate increases as well, but the pace of the increase is getting slower and slower. When the cache size is very large, all three replacement policies have virtually the same cache hit rate. Further increase of the cache size becomes undesirable.

The reason why LSL gives a better cache hit rate than the other two is because the views with higher *query-security-level* tend to be more general thus "better" than those with lower *query-security-level*. The more "good" views in the cache, the higher the cache hit rate. Since LSL can select those views with higher *query-security-level* to keep, it will increase the chance of a cache hit.

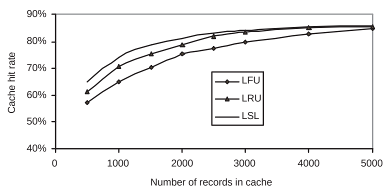

Figure 2: Cache size vs. cache hit rate.

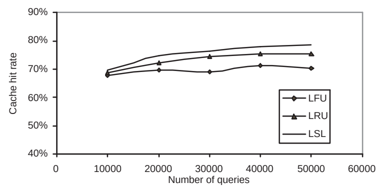

Figure 3: Number of queries vs. cache hit rate.

Effect of Number of Queries on Cache Hit Rate

As in Figure 3, the cache hit rate does increase as the number of queries processed increases. This is due to the reason that repeated query evaluation and cache update statistically select those "bad" views to eject and leave the cache with more "good" views. As the number of "good" views becomes larger and larger, the cache hit rate increases. Among the three cache replacement policies, LSL gives the highest cache hit rate while LFU the worst.

Effect of XML File Size on Average Query Processing Time

For comparison, the same query file is also processed under similar conditions but without using the cache. Figure 4 shows an obvious fact that as the XML file size increases, the curve without the cache increases at a much faster pace than those with the cache. The separation between the one without the cache and those with the cache is expected to be much greater as the XML file size gets larger. The time saving ratio is about 15% and 25% when the document size is around 25 MB and 30 MB, respectively.

Note that when the XML file size is small, the average query processing time is even shorter without the cache. This is because processing a query using then cache introduces the overheads of cache lookup and cache update. When the XML file is small, processing queries without the cache is quick enough to overcome the overheads caused by using the cache, so it takes shorter time.

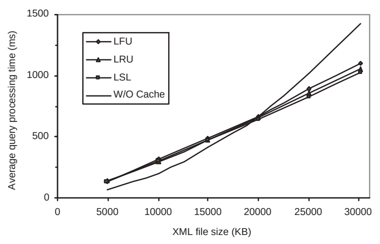

Figure 4: XML file size vs. average query processing time.

6 CONCLUSIONS

We designed and developed the SQS system, where the cache stores both query results and query security information. Its content can be efficiently looked up and repeatedly used to answer new queries. We discussed the factors affecting the cache hit rate and the average query processing time. Our study showed that LSL is currently the best cache replacement policy, and the cache can significantly improve the system performance on secure query evaluation.

REFERENCES

Bertino, E., Castano, S., Ferrari, E., and Mesiti, M. (2000). Specifying and enforcing access control policies for XML document sources. *World Wide Web Journal*.

Cho, S., Amer-Yahia, S., Lakshmanan, L., and Srivastava, D. (2002). Optimizing the secure evaluation of twig queries. In *Proceedings of the 28th VLDB Conference*.

Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. (2002). A fine-grained access control system for XML documents. *ACM TISSEC*.

Fan, W., Chan, C., and Garofalakis, M. (2004). Secure XML quering with security views. In *Proceedings of ACM SIGMOD*.

Gabillon, A., Munier, M., Bascou, J. J., Gallon, L., and Bruno, E. (2002). An access control model for tree data structures. In *5th Information Security Conference*.

Mandhani, B. and Suciu, D. (2005). Query caching and view selection for XML databases. In *Proceedings of the 31th VLDB Conference*.

Yu, T., Srivastava, D., Lakshmanan, L., and Jagadish, H. (2002). Compressed accessibility map: Efficient access control for XML. In *Proceedings of the 28th VLDB Conference*.