A Rewrite Based Approach for Enforcing Access Constraints for XML

Sriram Mohan[1], Arijit Sengupta[2], and Yuqing Wu[1]

[1] School of Informatics, Indiana University
[2] Raj Soin College of Business, Wright State University

Abstract. Access control for semi-structured data is nontrivial, as witnessed by the number of access control approaches in literature. Recently, a case has been made for expressing access constraints at finer levels of granularity on data nodes and extending constraints to structural relationships. In this paper, we introduce a rewrite-based approach for access constraint enforcement, based on the ACXESS framework we developed at Indiana University. The ACXESS framework utilizes virtual security views and introduces a set of rewrite rules that takes advantage of the Security Annotated Schema (SAS) - an internal representation for virtual views. It is capable of rewriting user queries against security views into queries against the source data, while honoring the access constraints.

Keywords: XML, Access Control, Query Rewriting.

1 Introduction

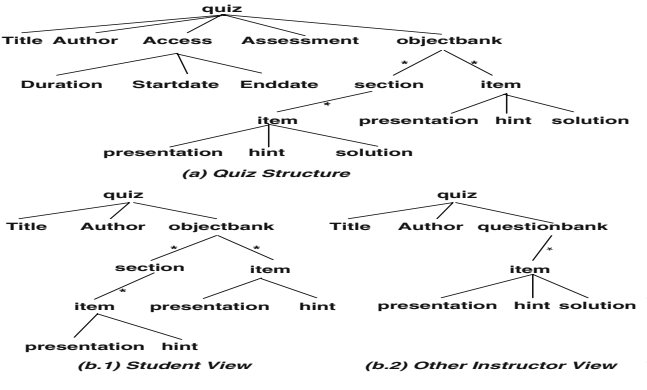

Fig. 1. Simplified tree structure an example security views for a course management system

XML has become one of the most extensively used data representation and exchange formats. The problem of access control in XML has many similarities to access control in relational and object-oriented databases. However, the semi-structured nature of XML increases the complexity of access constraint specification for XML. The tree pattern matching nature of XML queries further complicates the enforcement of the access constraints on XML data.

B. Apolloni et al. (Eds.): KES 2007/ WIRN 2007, Part III, LNAI 4694, pp. 1081–1089, 2007.

Example 1. *Consider a course management system used in open universities that share exam resources using the IMS-QTI schema*[1]*. A highly simplified version of this structure is shown in Figure 1(a). Users belonging to different access profiles should have different access privileges: an author (instructor) should have access to all elements of quizzes he/she writes ; a student should only have access to current quizzes in courses that he/she is registered for, but not have access to the solutions; if quizzes are shared among instructors of different universities, instructors other than the authors should have access to the questions as well as to the solutions, but potentially without the course-specific structuring.*

The access constraints listed in Example 1 are not uncommon. Different access constraint enforcement techniques with variable flexibility and efficiency can be used to enforce the constraints [1,2,3,4]. If materialized views are used to implement the above role-based access constraints, the view schemas for the 'student' and 'other-instructor' access levels would look like those shown in Figure 1 (b). However, it is well received that the storage and computational cost of view construction and maintenance hinders their utility and scalability. It is critical to develop a methodology that allows dynamic execution of queries on different access levels without the necessity of view materialization. In [5,6] we proposed a generalized access constraint specification and representation framework that enabled system users to specify access constraints on semi-structured data. In this paper, we introduce a query rewrite based access constraint enforcement technique that provides efficient access constraint enforcement. We will briefly review the access constraint specification language, and the security annotated schema in section 2, then, focus on the algorithms and rules for rewrite-based access constraint enforcement in section 3, followed by a discussion in section 4.

2 ACXESS

The infrastructure of the ACXESS framework - an XML security infrastructure developed at Indiana University, is as shown in Figure 2. The system can be divided into two components: Security View Construction (on the left) and Secure Query Rewrite (on the right). Taking the original XML schema and security view specification (in the specification language SSX) as input, the Security View Construction process constructs a Security Annotated Schema(SAS). SAS is an internal representation in our system and it is straightforward to derive the schema of the security

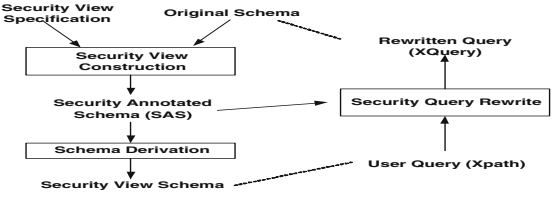

Fig. 2. The Infrastructure of the Security View Based Query Answering System

[1] IMS Global Consortium at http://imsglobal.org

view from an SAS. Rather than creating and maintaining materialized security views, we choose to rewrite the user queries (in XPath) to queries (in XQuery) that reflect the security constraints. The Secure Query Rewrite process (SQR) is rule-based, and translates an user XPath query on the security view to an XQuery expression against the original data.

We adopted the core of a graph editing language and introduced a security view specification language (SSX) [5] in the form of a set of graph editing primitives, that the system user can use to specify access constraints. The parameters and functions of the primitives are defined as follows (parameters within square brackets are optional):

- **create(destSPE** [2] **, newName)** creates a new element with tag 'newName', as a child of each element that matches the destSPE in the input schema.
- **delete(destXPath)** removes the sub-trees rooted at the elements that matches the destXPath in the input schema.
- **copy(sourceXPath, destSPE, [newName], [scope], [preserve])** For each element that matches the scope, the **copy** primitive creates an identical copy of the sub-trees rooted at the nodes that match the sourceXPath in the original schema with respect to the scope, and makes them the children of the elements that match the destSPE in the input schema.
- **rename(destSPE, newName)** assigns a new name to the elements that matches the destSPE in the input schema.

A security view specification is then written in the form of a sequence of these primitives. Each primitive takes the result of the subsequence in front of it as input. The final result is the Security Annotated Schema(SAS) for the SSX sequence.

To facilitate query answering and rewrites, we proposed an internal representation - Security Annotated Schema (SAS)[5] in the form of annotations to represent the schema transformation specified by an SSX sequence. The annotations include "New Node, Delete, Scope Stamp, Dirty Stamp, and Chronological Operation Sequence" and are associated with the element node that was modified. The annotations reflect the actual changes performed on the original schema structure.

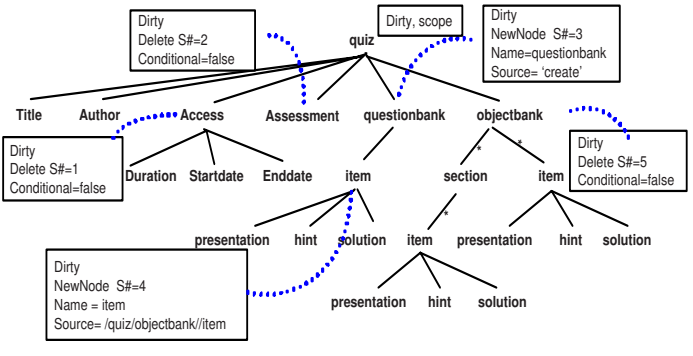

Fig. 3. SAS of the 'other-instructor' Security View

[2] SPE - *S*imple *P*ath *E*xpression is an XPath expression without branching predicates.

An example SAS representation for the 'other-instructor' security view is shown in Figure 3.

3 Enforcing Access Constraints Via Query Rewriting

XPath is the core language for all XML query languages, therefore we choose to rewrite user queries in XPath. Security view construction can create new structural relationships and to accomodate this, the rewrite process translates an input XPath expression into an XQuery expression.

3.1 Secure Query Rewrite Algorithm

The Secure Query Rewrite process (SQR) is rule-based. To facilitate the discussion of the rewrite rules and the algorithm, we define a recursive procedure as follows: Given an XML database D containing instances conforming to a schema S_0, a security view V (represented in the form of an SAS (S_v)), and an XPath expression $p = t_1[c_1]/t_2[c_2]\ldots /t_n[c_n]$ against V, where t_is are tokens and c_is are XPath expressions serving as branching predicates, the function $q = SQR_{S_v}(p, vb, vb')$ translates p to an XQuery expression q, referencing S_v and under a specific environment, represented by a set of variable-bindings vb (vb' represents the new variable-bindings in q).

We develop an algorithm to implement $q = SQR_{S_v}(p)$. The skeleton of the rewrite algorithm is presented in Figure 4. The algorithm accepts as input an *XPath query* q that needs to be rewritten. It iterates through q and walks the SAS tree based on the tokens found in q. Annotations, if found during the tree walk, are appropriately handled to generate the corresponding XQuery expression. In the case of a predicate in the input XPath expression, the predicate expression is treated as an XPath expression and the procedure is recursively called to generate an XQuery expression for the predicate. Operators, if encountered in a predicate, are substituted with equivalent XQuery operators while literals are carried over to the XQuery expression. The rules and the algorithm only deals with elements, however, they can be easily adapted to handle attributes. A conceptual presentation of the rules used by SQR is provided below.

```
SQR(q:Parsed XPath) {
  nexttoken = Obtain next token from input XPath
  if (nexttoken = ''//'') Apply RULE 3
  else if (currtoken is XPath condition c Apply RULE 1
  // the nexttoken is a string now.
  if (nexttoken is an element) {
    if (env.curnode has no children on the desired path)
      output ''return ()''
    if (env.curnode is not dirty) Apply RULE 2 for HCE
    else if (annotation A in curnode)
      switch A {
        case A = 'uncond delete': Apply RULE 6
        case A = 'copy': Apply RULE 7
        case A = 'create': Apply RULE 8
        case A = 'rename': Apply RULE 9
        }
    else if (Dirty node with no annotation) Apply RULE 4
    if (current element is last Token) Apply RULE 5
  }
} // end SQR
```

Fig. 4. The SQR Algorithm

3.2 Path Rules

The *Path Rules* deal with the XPath expression to be rewritten, and simplify the XPath expression by breaking it into parts, or keeping it as is, or rewriting it into XQuery expressions, based on the access constraints.

Rule 1. *Given an SAS S_v and an XPath expression $p = p_1/t_1[c_1]/p_2$ under environment vb, where p_1 and p_2 are sub-XPath expressions, t_1 is an element tag and c_1 is a branching predicate, taking the form of p_3[op p_4] (please note that 'op p_4' is optional) where p_3 and p_4 are XPath expressions and op is a XPath operator, the XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

```
for $i in SQR_{S_v}(p_1, vb, vb_1)
for $j in SQR_{S_v}(t_1, vb_1 ∪ {i}, vb_5)
where SQR_{S_v}(p_3, vb_5 ∪ {i, j}, vb_3) op
    SQR_{S_v}(p_4, vb_5 ∪ {i, j}, vb_4)
return SQR_{S_v}(p_2, vb_5 ∪ {i, j}, vb_2)
```

The premise behind this rule is that every branching predicate in an XPath expression is a valid XPath expression, and can be rewritten into an XQuery expression and be a part of the WHERE clause in the final rewritten XQuery expression. Therefore, we focus our discussion on rewriting a XPath expression without predicates- Simple Path Expression(SPE), in the rest of the paper.

A 'dirty' stamp associated with a node in the SAS identifies that either the node itself or its descendant(s) have been modified. Before presenting the rewrite rule for 'dirty' stamps, we first define the following important notion:

Definition 1. *Given an SAS S_v and an SPE expression $p = t_1/t_2 \ldots /t_n$, if there exists t_k, such that t_{k-1} has a 'dirty' stamp and none of t_k, t_{k+1}, ..., t_n has a 'dirty' stamp, we call t_k the* highest clean element (HCE) *of p on S_v, and $p_1 = t_1/t_2 \ldots /t_{k-1}$ the* prefix path *w.r.t. S_v and $p_2 = t_k/t_{k+1} \ldots /t_n$ the* suffix path *w.r.t. S_v.*

Rule 2. *Given an SAS S_v, and an SPE p under environment vb, if there exists an HCE on S_v such that $p = p_1/p_2$ (p_1 is the prefix path w.r.t. S_v, and p_2 is the suffix path w.r.t. S_v), we define the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ of p as follows:*

```
for $i in SQR_{S_v}(p_1, vb, vb_0)
return $i/p_2
```

The notion of *HCE* enables us to leave the suffix path "*as is*" in the rewritten expression. Since the nodes below the HCE are not modified they can be evaluated as a sub-expression in the resultant XQuery expression without rewrites.

Rule 3. *Given an SAS S_v, and an SPE $p = p_1//t/p_2$ under environment vb, where p_1 and p_2 are SPEs and t is an element tag, the XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

```
for $i in SQR_{S_v}(p_1, vb, vb_0)
return
    {SQR_{S_v}(t_1/p_2, vb_0 ∪ {i}, vb_1)} union
        ⋮
    {SQR_{S_v}(t_n/p_2, vb_0 ∪ {i}, vb_n)}
```

where $\{t_1, \ldots, t_n\}$ are the paths that lead to t from the node that matches p_1.

Uncertainty due to '//' can be handled by walking the SAS tree and searching all possible paths for the node of interest. Given that as a first step, we have assumed that the schema is acyclic, we have optimized the rewrite algorithm by ensuring that given a pair of nodes 'A' and 'B', all possible paths from 'A' to 'B' are pre-computed and stored in the SAS. Similarly, Rule 3 can be used to handle the wild card character '$\star$'.

Rule 4. *Given an SAS S_v, and an SPE $p = t_1/t_2/\ldots/t_n$ under environment vb, if node $t_y (y < n)$ is 'dirty' without any annotations then we can directly proceed to the next token in the path. The XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

$$\texttt{for \$i in } SQR_{S_v}(t_1/t_2\ldots/t_{y-1}, vb, vb_0)/t_y \\ \texttt{return } SQR_{S_v}(t_{y+1}/../t_n, vb_0 \cup \{i\}, vb_1)$$

When the tail of the XPath has a 'dirty' stamp, rather than stopping at the node and returning the whole subtree rooted at the node, the children of the tail node have to be reconstructed individually as the 'dirty' stamp indicates that at least one descendant node has been modified. This is achieved by recursively rewriting each and every child node of the node being constructed.

Rule 5. *Given an SAS S_v and an SPE $p = t_1/t_2\ldots/t_n$ under environment vb, if t_n is 'dirty' and has children $cld_1,\ldots cld_m$ in S_v, the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

$$\begin{array}{l} \texttt{for \$i in } SQR_{S_v}(t_1/t_2\ldots/t_{n-1}, vb, vb_0) \\ \texttt{return } < t_n > \\ \qquad\qquad SQR_{S_v}(cld_1, vb_0 \cup \{i\}, vb_1) \\ \qquad\qquad \vdots \\ \qquad\qquad SQR_{S_v}(cld_m, vb_0 \cup \{i\}, vb_m) \\ \qquad < /t_n > \end{array}$$

3.3 Structural Modification Rules

Structural modifications due to access constraints can be captured by Delete and New Node annotations. These annotations need to be treated differently in the rewrite procedure.

The SSX and SAS support both conditional and unconditional delete of substructures while constructing security views. If a node has an unconditional delete annotation, the node and its descendants are no longer accessible. Any query that attempts to retrieve them should evaluate to an empty set. The condition (if any) used to delete nodes is a valid XPath expression and is rewritten in the resultant XQuery expression, by computing its complement.

Rule 6. (Delete Rule) *Given an SAS S_v and an SPE $p = t_1/t_2\ldots/t_n$, if t_k $(0 \leq k \leq n)$ matches to a node that has an unconditional delete annotation in S_v, the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

```
return ()
```

If the nodes that match to t_k ($k < n$) are annotated with a conditional delete, with condition cond (an XPath expression), the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:

```
for $i in SQR_{S_v}(t_1/t_2 .../t_k, vb, vb_0)
where count(SQR_{S'_v}(cond, vb, vb_1)) = 0
return SQR_{S_v}(t_{k+1}/.../t_n, vb_0 ∪ {i}, vb_2)
```

Here, S'_v is the SAS generated by the prefix of the access constraint specification sequence (up to the operation before the conditional delete in question).

The New Node annotation resulting from the copy operator indicates that new subtrees have been constructed by cloning nodes. When a query retrieves information from a copied node or its descendants, data is retrieved from the source, along the source path. The retrieved data is compliant to any annotations on the descendant subtree.

Rule 7. (Copy Rule) *Given an SAS S_v, and an SPE $p = t_1/t_2 .../t_n$ under environment vb, if node t_y ($y < n$) matches to a node that has a newNode annotation generated by the copy operator with sourcePath $t_1/t_2/.../t_k/.../t_x$ and scope $t_1/t_2/.../t_k$ the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

```
for $i in SQR_{S_v}(t_1/t_2 .../t_k, vb, vb_0)
for $j in SQR_{S_v}(t_{k+1} .../t_{y-1}, vb_0 ∪ {i}, vb_1)
for $k in $i/t_{k+1}/t_{k+2}/.../t_x
return SQR_{S_v}(t_{y+1}/.../t_n, vb_0 ∪ {i, j, k}, vb_2)
```

New Node annotations resulting from rename and create operations indicate new tags that did not exist in the original data. If a newly created/renamed node appears in an XPath query, the rewrite process simply matches it and moves on to the subtrees rooted at the node in question.

Rule 8. (Create Rule) *Given an SAS S_v, and an SPE $p = t_1/t_2 .../t_n$ under environment vb, if node t_y ($y < n$) matches to a newNode annotation, generated as a result of a newly created node, the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

```
for $i in SQR_{S_v}(t_1/t_2 .../t_{y-1}, vb, vb_0)
return SQR_{S_v}(t_{y+1}/.../t_n, vb_0 ∪ {i}, vb_1)
```

Rule 9. (Rename Rule) *Given an SAS S_v, and an SPE $p = t_1/t_2 .../t_n$ under environment vb, if node t_y ($y < n$) matches to a newNode annotation generated by renaming nodes (ancestor depersonalization), the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

for $i in $SQR_{S_v}(t_1/t_2 \ldots /t_{y-1}/t_x, vb, vb_0)$
return $SQR_{S_v}(t_{y+1}/ \ldots /t_n, vb_0 \cup \{i\}, vb_1)$

Example 2. *With respect to the rewrite rules described above, consider the query '/quiz//item[hint]', which finds all items in the quizzes with a hint. When users in different user groups issue this query, it will be rewritten differently, based on the security constraints specified for the user group. The rewritten queries for the user groups are as shown in Table 1. For the 'student' group, we find that there are two differnt paths to 'item' nodes and both are reconstructed. The results are reconstructed to hide the 'solutions', and the conditional delete operation in the access control definition is taken care of by a "where" clause. The rewrite for the 'other-instructor' group ensures that all the different paths to the 'item' nodes are reconstructed.*

Table 1. Rewritten queries for query */quiz//item*[*hint*] for different user groups

other instructor group	student group

for $q in doc("data/quizzes.xml")/quiz
return
{
 for $q1 in $q/objectbank
 for $q2 in $q1/item
  where count(for $q3 in $q2/hint return $q3)>0
 return $q2
}
union
{
 for $q4 in $q/objectbank
 for $q5 in $q4/section/item
  where count(for $q6 in $q5/hint return $q6)>0
 return $q5
}
	for $q in doc("data/quizzes.xml")/quiz
	where $q/Access/Startdate <= currdate
	and $q/Access/Enddate >= currdate
	return
	{
	  for $q1 in $q/objectbank/item
	  where count(for $q2 in $q1/hint return $q2) > 0
	  return <item> {$q1/text} {$q1/hint} </item>
	}
	union
	{
	  for $q3 in $q/objectbank/section/item
	  where count(for $q4 in $q3/hint return $q4) > 0
	  return <item> {$q3/text} {$q3/hint} </item>
	}

4 Discussion and Conclusion

The complexity of the query rewrite is bound by two factors (i) the size of the input query (ii) the size of the schema fractions that has been marked as 'dirty'. This ensures that SQR has a favorable evaluation time when compared to that of post-query filtering and view materialization. Experimental analysis[3] reveals that the rewrite algorithm and the rules are effective and efficient, especially for queries with // and wildcards and the queries targeting nodes whose access has been denied. The rewrite rules can be shown to be sound and complete with respect to the SAS [3].

In this paper, we proposed a rewrite-based access constraint enforcement algorithm and associated rules for the ACXESS XML security infrastructure. The proposed technique is capable of handling not only the access constraints specified by SSX, but with careful generalization of SAS, it is also capable of handling access constraints specified by other existing access control approaches in literature. The techniques proposed in this paper assume that every XML database has a schema and that it is acyclic. We are looking forward to extending our work to handling recursions in schema and to tackle the more generic scenario where partial or no schema is available.

[3] Details of theorems, proofs, test cases and results are not included due to space limitations and are available at http://www.cs.indiana.edu/∼acxess.

References

1. Fan, W., Chan, C.Y., Garofalakis, M.: Secure XML querying with security views. In: SIGMOD (2004)
2. Finance, B., Medjdoub, S., Pucheral, P.: The case for access control on XML relationships. In: CIKM (2005)
3. Gabillon, A.: A formal access control model for XML databases. In: SDM (2005)
4. Yu, T., Srivastava, D., Lakshmanan, L.V.S., Jagadish, H.V.: Compressed accessibility map: Efficient access control for XML. In: Bressan, S., Chaudhri, A.B., Lee, M.L., Yu, J.X., Lacroix, Z. (eds.) CAiSE 2002 and VLDB 2002. LNCS, vol. 2590, Springer, Heidelberg (2003)
5. Mohan, S., Sengupta, A., Wu, Y.: Access control for XML - a dynamic query rewriting approach. In: CIKM (2005)
6. Mohan, S., Klinginsmith, J., Sengupta, A., Wu, Y.: Access control for XML with enhanced security specifications. In: ICDE (2006)