Controlling Access to XML Documents over XML Native and Relational Databases

Lazaros Koromilas, George Chinis,
Irini Fundulaki, and Sotiris Ioannidis

FORTH-ICS, Greece
{koromil,gchinis,fundul,sotiris}@ics.forth.gr

Abstract. In this paper we investigate the feasibility and efficiency of mapping XML data and access control policies onto relational and native XML databases for storage and querying. We developed a *re-annotation* algorithm that computes the XPath query which designates the XML nodes to be re-annotated when an update operation occurs. The algorithm uses XPath static analysis and our experimental results show that our re-annotation solution is on the average 7 times faster than annotating the entire document.

Keywords: XML, access control, XML to relational mapping

1 Introduction

XML has become an extremely popular format for publishing, exchanging and sharing data by users on the Web. Often this data is sensitive in nature and therefore it is necessary to ensure selective access, based on access control policies. For this purpose flexible access control frameworks must be built that permit secure XML querying while at the same time respecting the access control policies. Furthermore such a framework must be efficient enough to scale with the number of documents, users, and queries.

In this paper we study how to control access to XML documents stored in a relational database and in a native XML store. Prior work proposed the use of RDBMS for storing and querying XML documents [1], to combine the flexibility and the usability of XML with the efficiency and the robustness of a relational schema. In this paper we examine the feasibility and efficiency of using the above approach to *enforce access control policies*. In particular, we study how to control access on XML documents following the *materialized* approach, in which the XML document is stored in a database along with *annotations* attached to the nodes; these specify whether a node is accessible or not. We evaluate our approach using *(i)* a native XML storage system and *(ii)* a relational database where the XML documents are shredded à la ShreX [8]. Specifically we:

– propose a method to *annotate* XML documents stored in a relational database and in an XML database;

– discuss an *optimization* procedure based on *XPath containment* that removes *redundant* access control rules from a policy;
– develop a *re-annotation* technique that allows us to re-compute the annotations of a portion of the nodes in an XML document if a document update occurs; and finally
– we discuss results of extensive experiments that compare annotation and re-annotation techniques for the relational and the XML cases.

This is the first attempt to compare the use of relational and XML databases to store annotated (with accessibility information) XML documents. Annotation-based enforcement techniques have been considered in [3, 7] for rule-based policies. More sophisticated techniques for storing and querying annotations have been investigated [26, 27]. The related problem of optimizing security checks during query evaluation with respect to an annotated document was investigated in [5]. XML access control over relational databases has been also studied in [23]. Our work is different in that we use annotations (materialized approach), whereas Lee et al. check the accessibility of the document on-the-fly. [20] discusses a "function-based" model that translates policy rules to functions (e.g. Java methods) which are subsequently called to check the policy whenever a part of the document is accessed. Security views [10, 16] address the problem of information leaks in the presence of *read-only* security policies and queries. Security views contain just the information a user is allowed to read; queries to the view can be translated efficiently to queries on the underlying data, foregoing expensive view materialization and maintenance. However, previous work on annotation-based security policies, such as compressed accessibility maps, does not address the problem of keeping the annotations consistent with the policy when the document or policy changes. These techniques have not yet been used directly to provide access control in an XML database system; it appears that doing so would require modifying the database system internals.

1.1 Motivating Example

Before we formally discuss our approach, we present an example from the medical domain. Consider the XML DTD of Figure 1 that is used to represent information for hospitals, their departments, staff and patients.

We choose a node and edge labeled graph representation for the XML DTD where nodes in the graph are the element types of the XML DTD, and edges represent the content models of an element type (sequence, choice). Dashed arrows connecting a node with its children nodes capture the choice, whereas straight lines capture the sequence content model. Edges are labeled with *, + and ? to denote the occurrence indicators in the XML DTD ("zero or more", "one or more" and "optional" respectively).

In the graph, a valid hospital instance of this schema contains one or more departments (dept+). Each department holds information about patients (patients) and its staff (staffinfo). There may be zero or more patients (patient*) and zero or more staff members (staff*). A patient has an identifier (psn), a registered

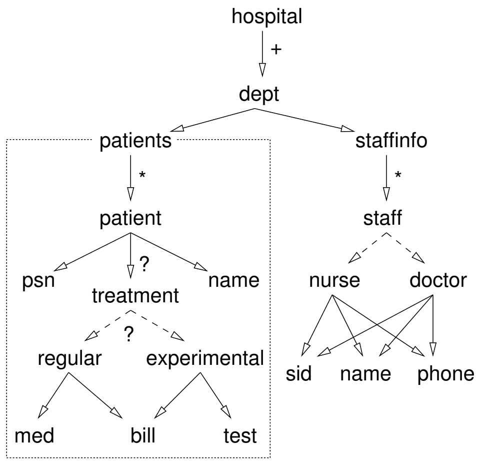


Fig. 1. Hospital schema.

Rule	Resource	Effect
R_1	//patient	$+$
R_2	//patient/name	$+$
R_3	//patient[treatment]	$-$
R_4	//patient[treatment]/name	$+$
R_5	//patient[.//experimental]	$-$
R_6	//regular	$+$
R_7	//regular[med= "celecoxib"]	$+$
R_8	//regular[bill > 1000]	$+$

Table 1. Hospital policy rules.

name (name) and an optional treatment (treatment?). The treatment may be either conventional (regular?) or experimental (experimental?); it can also be unspecified (an empty element). Regular treatments have a medication (med) and a bill (bill), whereas experimental treatments are associated with a medical exam (test) and a bill (bill). Staff members are doctors (doctor) or nurses (nurse). In either case they have an identifier, a name and a phone number (sid, name and phone respectively).

A sample partial instance of the hospital schema is presented in Figure 2. For the sake of simplicity we focus on the patients element of a department and show three different patients. We will be using this document together with the access control rules of Table 1 in the examples in the remainder of this paper.

Table 1 shows the access control rules specified for the hospital XML DTD. Each rule has the form *(resource, effect)* where *resource* is an XPath expression that designates the nodes in the XML document concerned by the rule and *effect* specifies whether the node is accessible (sign "+") or inaccessible (sign "−"). Rule R_1 says that all patient nodes are accessible whereas rule R_3 specifies that patient nodes that have a treatment are not. Rules R_4 and R_2 specify that the names of patients that have a treatment and patients in general are accessible. Patients under experimental treatment are not accessible according to rule R_5. Rule R_6 gives access to all regular treatment nodes; in addition rules R_7 and R_8 are more specific and specify that regular treatment nodes that have a medication (med) with value "celecoxib" or a bill (bill) with a value greater than 1000 respectively are accessible. This set of rules is associated with a *conflict resolution* policy and *default semantics* [14,15,17]. The former specify the accessibility of a node in the case in which it is in the scope of access control rules with opposite signs. The later determines the default accessibility of a node. In our example we consider that the *conflict resolution* policy is *deny overrides*

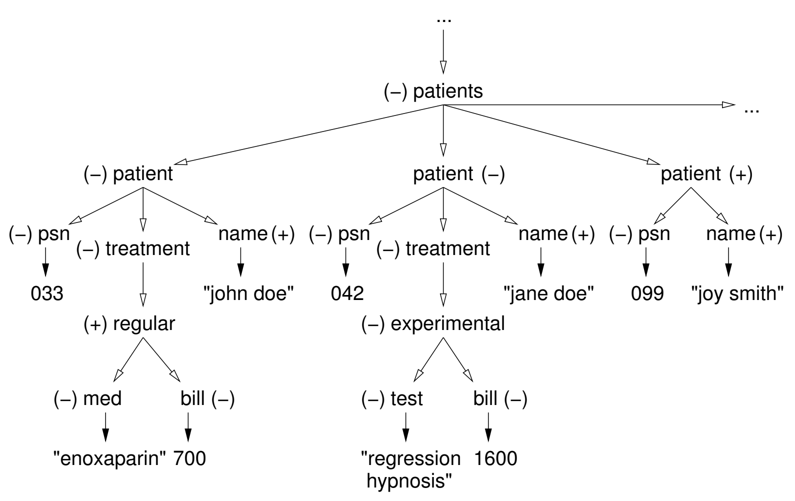


Fig. 2. Partial hospital document.

(the rule that denies access to a node overrides the one that grants access to it) and the default semantics is *deny* (nodes are inaccessible by default). We say that an XML node is in the scope of an access control rule, if it is in the result of the evaluation of the resource (i.e., XPath expression) part of the rule on the XML document.

Figure 2 shows the *annotated* XML document where annotations designate whether a node is accessible (label "+") or not (label "−"). Note that the elements for which no access control rule is specified are annotated with "−" (denied access by default). The first and second patient elements are not accessible: both elements have a treatment subelement and according to rule R_3 are not accessible (note that R_3 overrides R_1 due to the conflict resolution policy). On the other hand, the third patient is accessible, since it is in the scope of rule R_1 and not in the scope of either R_3 or R_5.

2 Preliminaries

2.1 XML Trees

We model XML documents as *rooted unordered* trees with labels from the set $\Sigma \cup D \cup \{*\}$. Σ is a finite set of *element names*, D a data domain and $*$ is the wildcard (matches any label). We represent an XML document as a tree $T = (V_T, E_T, R_T, \lambda_T)$, where *(i)* V_T is the set of nodes in T, *(ii)* $E_T \subseteq V_T \times V_T$ is the set of edges, *(iii)* $\lambda_T : V_T \rightarrow \Sigma \cup D$ maps nodes to element names from Σ and values in D and *iv)* R_T is a distinguished node in V_T, called the root node.

2.2 XPath

The fragment of XPath that we will be using in queries and access control rules is defined as follows:

$$\begin{array}{lrl} \text{Paths} & p & ::= axis :: ntst \mid p[q] \mid p/p \\ \text{Qualifiers} & q & ::= p \mid q \ \mathsf{and}\ q \mid p = d \\ \text{Axes} & axis & ::= \mathsf{child} \mid \mathsf{descendant} \\ \text{Node Test} & ntst & ::= l \mid * \end{array}$$

where l is an element label from Σ, and d a value in D.

The expressions are built using only the *child* and *descendant* axes of XPath and conditions which test for the existence of sub-elements or constants in the subtree of an element. We use the standard abbreviated form of XPath expressions. For example, /a//b[*] is an abbreviation of

$$\mathsf{/child::a/descendant\text{-}or\text{-}self::node()/child::b[child::*]} \ .$$

For p an absolute XPath expression (a path expression starting with "/"), and T an XML tree, we write $[\![p]\!](T)$ to denote the set of nodes of T obtained from evaluating expression p on the root node of T. The semantics of XPath expressions are defined in [2, 12, 25].

We say that an XPath expression p is *contained in* another expression q (denoted by $p \sqsubseteq q$), if for every XML tree T, $[\![p]\!](T) \subseteq [\![q]\!](T)$. We say that two XPath expressions are *disjoint* (denoted by $p \not\varnothing q$) if their intersection is empty. That is, for every T, $[\![p]\!](T) \cap [\![q]\!](T) = \emptyset$. Otherwise, we say p and q *overlap* (denoted by $p \varnothing q$).

3 Access Control Framework

An *XML access control policy* is defined by a set of rules that specify *who* has *access* to which *data*. We undertake the more or less agreed definition of an access control rule which is a tuple of the form (*requester*, *resource*, *action*, *effect*, *propagation*) where:

- *requester* refers to the user or a set of users concerned by the authorization;
- *resource* refers to the data that the requester is (or not) authorized to access;
- *action* refers to the action (read, write, modify, delete etc.) that the requestor is (or not) allowed to perform on the resource;
- *effect* specifies whether the rule grants ("+" sign) or denies ("−" sign) access to the resource and finally
- *scope* which defines whether the rule applies to the node only, or to its subtree [11].

In this paper we assume that the *requester* and *action* parameters are fixed and concentrate on the *resource* and *effect* components. We define the *scope* of a rule to be the *XML node* itself (explicit rules). Implicit rules are not considered here (no accessibility inheritance). We will refer to the access control rules that grant access to a node (*effect*='+') as *positive* and those that deny access to it (*effect*='$-$') as *negative*.

For simplicity we define an access control rule R to be a tuple of the form R = (*resource*, *effect*) with *resource* an XPath expression in the fragment discussed in Section 2 and *effect* $\in \{+, -\}$. We say that a node n in the XML tree T is in the scope of an access control rule r = (*resource*, *effect*) if $n \in [\![resource]\!](T)$.

We define an *access control policy* P to be a tuple of the form $P = (ds, cr, A, D)$ where:

- ds is the *default semantics* $ds \in \{+, -\}$,
- cr the *conflict resolution* policy $cr \in \{+, -\}$,
- A is the set of positive access control rules and
- D is the set negative rules.

As previously discussed, *conflict resolution* specifies the accessibility of a node in the case in which it is in the scope of access control rules with opposite signs. Default semantics indicate that a node in the XML tree is accessible/inaccessible by default.

Intuitively, an access control policy P restricts the set of nodes of an XML tree T returned as the answer to the user query. The *semantics* of an access control policy P for an XML tree T are the set of *accessible* nodes of T. We denote with $[\![P]\!](T)$ the semantics of a policy P for an XML tree T. Table 2 defines the semantics of a policy $P = (ds, cr, A, D)$ where $\mathcal{U}(T)$, $[\![D]\!](T)$ and $[\![A]\!](T)$ are the nodes of tree T, the nodes that are in the scope of some negative, and the nodes in the scope of some positive rule of policy P respectively.

$$\begin{aligned}
[\![(+,+,A,D)]\!](T) &= \mathcal{U}(T) - ([\![D]\!](T) - [\![A]\!](T))\\
[\![(-,+,A,D)]\!](T) &= [\![A]\!](T)\\
[\![(+,-,A,D)]\!](T) &= \mathcal{U}(T) - [\![D]\!](T)\\
[\![(-,-,A,D)]\!](T) &= [\![A]\!](T) - [\![D]\!](T)
\end{aligned}$$

Table 2. Semantics of an access control policy $P = (ds, cr, A, D)$.

In the case in which the default semantics is allow and the conflict resolution is allow overrides, then the accessible nodes are all the nodes in T *except* those that are in the scope of a negative rule and not those in the scope of a positive rule ($\mathcal{U}(T) - ([\![D]\!](T) - [\![A]\!](T))$). In the case in which the default semantics is deny and the conflict resolution policy is allow, the accessible nodes are exactly those that are in the scope of some positive acces control rule ($[\![A]\!](T)$). On the other hand, if the default semantics is allow and the conflict resolution is deny, the accessible nodes are all the XML nodes in T *except* those that are in the scope of some negative rule ($\mathcal{U}(T) - [\![D]\!](T)$). Finally, in the case that occurs most often in practice, if the conflict resolution policy is deny overrides and the default semantics is deny, the accessible nodes are those that are in the scope of some positive rule except those that are in the scope of some negative rule ($[\![A]\!](T) - [\![D]\!](T)$).

4 System Architecture

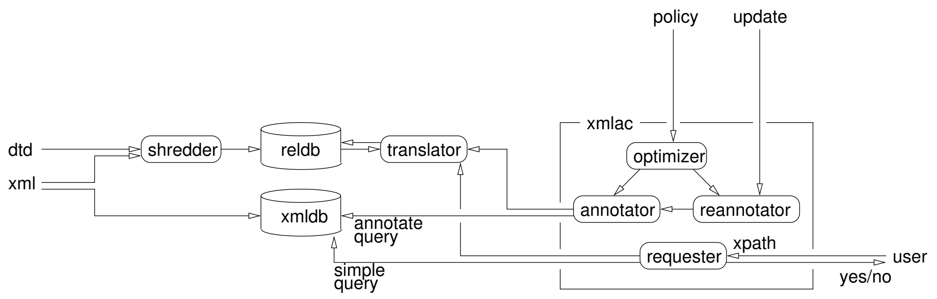


Fig. 3. System components.

We will now present the architecture of the access control system and describe the functionality of its key components. The core of the system is comprised of the optimizer, annotator, reannotator and requester modules.

Module optimizer shown in Figure 3 is responsible for detecting and removing the redundant access control rules from the access control policy. We discuss this idea in detail in Section 5.1.

The annotator module is responsible for computing the queries used to annotate the XML document with accessibility information. More specifically, it takes as input an XML access control policy P and computes the SQL and the XQuery queries that will be used to annotate with accessibility information the relational representation of the XML document and the XML document stored in the native XML store resp.. These queries implement the semantics of an access control policy as presented in Section 3 and will be discussed in detail in Section 5.2.

The reannotator module is responsible for computing the SQL and XQuery queries to *re-annotate* the already annotated XML document when a document update has occured. The idea is that when an update occurs (i.e., a node is deleted or inserted in the document), the accessibility of a node might change: for instance, if the treatment element of a patient element is deleted, then the latter becomes accessible (Table 1, rules R_1 and R_3). In this case, we need to consider re-annotating only the patient elements in the XML document. The re-annotation algorithm is discussed in detail in Section 5.3.

The requester module is the front-end of our system. A user request is sent by the requester to the relational and native XML stores for evaluation and depending on the result of the evaluation, it either returns the requested data or denies access to the user request.

To store an XML document in a relational database, we first need to create the relational tables used to store the XML document, and in a second phase, produce a relational representation of the XML document using these tables. We employ ShreX [1, 8] to obtain the relational representation of XML documents. ShreX is a system that handles the translation of XML data into relational tables. This includes relational schema creation, document loading and database querying. It takes as input an XML Schema [4, 9, 24] and produces a mapping to create relational tables so that XML documents which are valid instances of the given XML Schema, can be stored. ShreX is also responsible for translating XPath [6] queries to SQL queries which are then evaluated on the relational representation of the XML document.

In our system we control access to *read-only* queries expressed as XPath expressions. We follow an *all-or-nothing* semantics for query answering: if all the nodes requested by the XPath expression are accessible (i.e., annotated with "+" sign), then we return the requested nodes. Otherwise, we deny access to the user request.

5 Controlling Access to XML Documents

In this section we discuss in detail our approach on controlling access to XML documents stored in a relational and an XML store.

5.1 Access Control Policy Optimization

The first step is to remove redundant rules from the access control policy. This redundancy elimination is performed by the optimizer module presented in Section 4. Given a policy P, we say that an access control rule R is *redundant*, if there exist some access control rule R' such that

1. R, R' are both either *positive* (in A) or *negative* (in D) and
2. R is *contained in* R'.

We say that an access control rule R=(*effect*, *resource*) is *contained* in a rule R'=(*effect*, *resource'*) iff *resource* is *contained in resource'*. An XPath expression p is *contained in* an XPath expression p' ($p \sqsubseteq p'$) iff the set of nodes obtained by evaluating p on any XML tree T is a subset of the set of nodes obtained by evaluating p' on T [18,19,22].

Algorithm Redundancy-Elimination shown in Figure 4 takes as input a set of access control rules S and returns a subset S' of S which is free of redundant rules. The idea is the following: redundancy elimination is performed for both sets of positive and negative rules (A and D). The resulting redundancy-free sets of rules are combined to obtain a revised policy. We employ the containment algorithm of [13,18]. Containment for fragments of XPath such as $XP(/, //, *, [])$ has been studied in [18] and for larger fragments in [19] (see [22] for a survey).

```
REDUNDANCY-ELIMINATION($rules$)

Ensure: $\forall r_1 \forall r_2 \in rules \Rightarrow \nexists r_1 \sqsubset r_2$
 1: for all $r \in rules$ do
 2:     for all $r' \in rules$ where $r' \neq r$
        do
 3:         if $r \sqsubset r'$ then
 4:             $rules \leftarrow rules - \{r\}$
 5:         else if $r \sqsupseteq r'$ then
 6:             $rules \leftarrow rules - \{r'\}$
 7:         else
 8:             {neither disjoint nor over-
                lap, do nothing}
 9:         end if
10:     end for
11: end for
12: return $rules$
```

Fig. 4. Eliminating *redundant* access control rules.

```
ANNOTATION-QUERIES
Require: Policy $\mathcal{P} = (A, D, ds, cr)$
Ensure: Annotation Query
 1: for all $r \in A$ do
 2:     $grants \leftarrow grants$ UNION $r$
 3: end for
 4: for all $r \in D$ do
 5:     $denys \leftarrow denys$ UNION $r$
 6: end for
 7: if $ds$ = '−' then
 8:     if $cr$ = '−' then
 9:         $toupdate \leftarrow$ query($grants$ EXCEPT
            $denys$)
10:     else {$cr$ = '+'}
11:         $toupdate \leftarrow$ query($grants$)
12:     end if
13: else {$ds$ = '+'}
14:     if $cr$ = '−' then
15:         $toupdate \leftarrow$ query($denys$)
16:     else {$cr$ = '+'}
17:         $toupdate \leftarrow$ query($denys$ EXCEPT
            $grants$)
18:     end if
19: end if
20: return $toupdate$
```

Fig. 5. Computing *annotation* queries

For our motivating example, the redundancy-free access control policy is shown in Table 3. Rule R_4 is removed because //patient[treatment]-/name $\sqsubseteq$ //patient/name (it is contained in R_2). Similarly, rules R_7, R_8 are contained in R_6. Rule R_3 is contained in R_1, however it is not eliminated because the two have different effects.

Rule	Resource	Effect
R_1	//patient	+
R_2	//patient/name	+
R_3	//patient[treatment]	−
R_5	//patient[.//experimental]	−
R_6	//regular	+

Table 3. Redundancy-free policy.

5.2 Annotating XML Documents with Accessibility Information

To annotate an XML document independently of where it is stored, we must first compute the *annotation queries* that implement the semantics of the XML access control policy P. Algorithm ANOTATION-QUERIES takes as input an XML access control policy P and computes the SQL and the XQuery queries that will be used to annotate the relational and XML databases with accessibility information, implementing the policy semantics as described in Table 2. In the relational case, the resource part of an access control rule (XPath expression) is translated into an equivalent SQL query q using the ShreX [1,8] translation.

The resource part of the rules that grant (resp. deny) access are unioned using the relational UNION (XQuery union) operator. Depending on the default semantics and conflict resolution policy the relational EXCEPT (XQuery except) operator is used to express the annotation query that implements the semantics of an access control policy.

In the relational case, the components in the UNION query, are the *translated* SQL queries from the XPath expressions using the chosen XML-to-relational mapping.

Relational Approach To store an XML document in a relational database, the tree (specified in our case with an XML DTD), must first be mapped into an equivalent, relational schema. Using this mapping the XML document is then shredded and loaded into the relational tables.

In the context of XML, we need to capture access control information at the *XML node* level. To satisfy this requirement, we map *each element type* in the XML DTD to a relational table. More specifically, each element type E with attributes $A_1, A_2, \ldots A_n$ in the XML DTD, is mapped to a table $E_T(id, pid, A_1, A_2, \ldots A_n, s)$ where id is the primary key for E_T, pid is a foreign key that refers to a relational table E'_T to which the parent element type E' of E is mapped to. Finally s is an additional column that stores the access permission for the tuple (i.e., node in the XML document).

The value of an A_i column is the value of the A_i attribute of the XML node. For nodes whose type is a base type such as string, integer we define tables of the form $E_T(id, pid, A_1, \ldots, A_n, v, s)$ where v is the value of the XML node. The id key is unique not only through the table but throughout the entire database; we will call this key 'universal identifier'.

patients

id	pid	s
1	null	−

patient

id	pid	s
2	1	−
9	1	−
16	1	+

psn

id	pid	val	s
3	2	*033*	−
10	9	*042*	−
17	16	*099*	−

treatment

id	pid	s
4	2	−
11	9	−

name

id	pid	val	s
8	2	*john doe*	+
15	9	*jane doe*	+
18	16	*joy smith*	+

regular

id	pid	s
5	4	+

experimental

id	pid	s
12	11	−

med

id	pid	val	s
6	5	*enoxaparin*	−

bill

id	pid	val	s
7	5	*700*	+
14	12	*1600*	+

test

id	pid	val	s
13	12	*regression hypnosis*	+

Table 4. Relational representation of the XML document in motivative example.

For our motivating example we will define one table of the form $E_T(id, pid, s)$ per element type in the XML DTD shown in Figure 1. Table 4 shows the output of the XML to relational mapping for the XML document shown in Figure 2 where for each node in the XML document whose element type is E we create one tuple in table E_T. The accessibility of each tuple (i.e., corresponding XML node) is initialized to the default semantics of the policy.

To annotate the tuples in the relational store given a policy P, we must first find the tuples that are in the semantics of P (i.e., are accessible according to policy P) and then perform the necessary update operation. To obtain the tuples to be annotated, we run the SQL query obtained by executing algorithm Annotation-Queries. Our annotation algorithm is a two phase algorithm: in the first phase we find the id's of all the tuples that need to be annotated, and in the second phase, for each such tuple we run the update query that changes the value of the s column.

For example, consider the policy shown in Table 1. The translated SQL queries for the rules of the policy are given below: For rule R_1 the produced query Q_1 is:

```
SELECT pat1.id
FROM patients pats1, patient pat1
WHERE pats1.id = pat1.pid;
```

For rule R_3 the corresponding query Q_3 is

```
SELECT pat1.id
FROM patients pats1, patient pat1, treatment treat1,
WHERE pats1.id = pat1.pid AND pat1.id = treat1.pid
```

For rule R_7 the corresponding query Q_7 is

```
SELECT med1.id
FROM patients pats1, patient pat1, treatment treat1,
     regular regular1, med med1
WHERE pats1.id = pat1.pid AND pat1.id = treat1.pid
  AND treat1.id = regular1.pid AND regular1.id = med1.pid
  AND med1.v = ‘celecoxib’
```

Finally, given that default semantics is *deny* and conflict resolution is *deny overrides*, the SQL query that implements the semantics of the redundancy-free policy in Table 3 is:

$(Q_1$ UNION Q_2 UNION Q_6 EXCEPT $(Q_3$ UNION $Q_5))$ where a query Q_i is defined for access control rule R_i.

Note that the result of the SQL query is a set of tuple identifiers that are in the semantics of the access control policy, i.e., are accessible. In the relational context, to update a relational tuple, we need to know the name of the table that the tuple belongs to. The universal identifier does not provide us with that kind of information. Consequently, to identify the table that a tuple (i.e., the identifier of a tuple belongs to), we iterate over all tables of the database. For each table the algorithm computes the intersection between the universal identifiers of the tuples included in the table and those computed by applying the SQL query that implements the semantics of policy P. The tuples with primary key in the computed intersection are updated to reflect the accessibility of a node. The annotation process from the creation of the annotation queries to the addition of accessibility information to the relational tuples is shown in Algorithm Annotate.

```
ANNOTATE(policy)
Require: Policy $P$, Relational DB $D$
Ensure: Annotated $D$ according to $P$
    {first, produce the SQL query}
 1: $sqlquery \leftarrow$ ANNOTATION-QUERIES($P$)
    {execute the SQL query to compute set $S$ of tuple ids}
 2: $S \leftarrow$ query($sqlquery, D$)
 3: for all $table \in schema$ do
 4:    $ids \leftarrow$ query(SELECT $id$ FROM $table$)
 5:    $upids \leftarrow ids \cap S$
       {produce the SQL update queries to update the permissions}
 6:    for all $upid \in upids$ do
 7:       query(UPDATE $table$ SET $s =$ '+' WHERE $id = upid$)
 8:    end for
 9: end for
```

Fig. 6. Annotation Algorithm for Relational DB

Native XML In the case of a native XML store, the annotation process is straightforward. We choose to store accessibility annotations for XML elements in the form of the XML *attribute sign* that takes value "+" (if the node is accessible) or "−" otherwise. The idea is the following: we employ algorithm ANNOTATION-QUERIES to obtain the XQuery expression that implements the semantics of the access control policy (i.e., determines the accessible nodes). To minimize the amount of information stored, we choose to annotate the accessible (inaccessible) nodes for policies with deny (grant) default semantics respectively.

The modification of the *sign* attribute of the nodes is performed with function `xmlac:annotate()` shown below. The function takes as input the XML node to be annotated (n), and the annotation label (val). If the node does not have a *sign* attribute, then the attribute is inserted along with its value, otherwise, the current value is updated.

```
function xmlac:annotate($n as element(), $val as xs:string)
{
        if (count($n/@sign) = 0)
        then do insert attribute sign { $val } into $n
        else do replace value of $n/@sign with $val
};
```

For instance, for the motivative example (policy of Table 3) we run the following query to annotate the XML nodes.

```
for $n := doc("xmlgen")((R1 union R2 union R6) except (R3 union R5))
return xmlac:annotate($n, "+")
```

5.3 Re-annotation

When a database is frequently updated, the cost of keeping the annotations consistent with the access control policy becomes considerably large. The simple approach to tackle this problem is to delete all annotations and annotate from scratch, a process that induces large processing cost. In this section we discuss how we can identify the access control rules that should be triggered to re-annotate the nodes whose access permission changed due to the update.

Intuitively, the nodes that must be re-annotated are all the nodes that are in the scope of an access control rule that specifies a condition (filter) on a node that is modified (inserted or deleted) by the update operation. For instance, consider the following example: suppose that the `treatment` subelement of a `patient` element is deleted. Recall that access control rule R_3 of our motivating example states that `patient`s with `treatment` are inaccessible. In this case, we should consider for re-annotation all patient elements, since rule R_3 that was used in their annotation is no longer applicable. To determine this set of rules we employ XPath containment tests between the rules and the update query. We discuss this in more detail in the following.

As a tool to discover the access control rules that must be considered for re-annotating the access permissions of a node, we compute their *dependency graph*. The graph captures interdependencies between the access control rules: for every rule R in a policy P that has in its scope a node n, the dependency graph stores all the rules R' of opposite sign that also have in their scope node n. The graph allows us to get in constant time all the rules that should be considered for re-annotating an XML node.

```
Depend($P$)
1: for all $r \in P$ do
2:     $r.visited \leftarrow false$
3: end for
4: for all $r \in P$ do
5:     Depend-Resolve($r, dlist$)
6:     $r.depends \leftarrow dlist$
7: end for
```

```
Depend-Resolve($r, dlist$)
1: $r.visited \leftarrow true$
2: for all $n \in r.neighbours$ do {explore $(r, n)$}
3:     if $n.visited = false$ then
4:         $dlist \leftarrow dlist \bigcup \{n\}$
5:         Depend-Resolve($n, dlist$)
6:     end if
7: end for
```

Fig. 7. Dependency resolution algorithm.

The dependency graph is represented as a *list* of *adjacency lists*, where each member of the list corresponds to an access control rule in a policy P. We associate with each rule r, attributes *neighbours* that stores the adjacency list (i.e., dependency graph) for r and *visited* to note that the rule has been visited during the execution of the algorithm. Algorithm Depend computes the dependency graph as follows: we iterate over all rules in a policy P to discover the dependencies that arise. Each call to Depend-Resolve for rule r initiates a DFS-like recursive traversal that finds all dependent rules for r. In line 2 of algorithm Depend-Resolve the $r.neighbours$ variable denotes the *adjacency list* for r. In these adjacency lists, each entry n is neighbor of another entry r iff r has a containment relation with n: $r \sqsubseteq n \vee n \sqsupseteq r \vee r = n$.

For instance, consider the rules R_1 and R_3 of the motivative example (Table 3). We can see that R_3 is contained in R_1 (//patient[treatment] $\sqsubseteq$ //patient) as the former returns patients with treatment subelement whereas the latter all patients. Consequently, after this process rule R_3 will be included in the dependency list of rule R_1 and vice versa. We should clarify that we are interested in dependencies between rules that have opposite effect, in contrast to the offline policy optimization where we eliminated rules of the same effect.

```
TRIGGER($P, u$)
 1: $rules \leftarrow \emptyset$
 2: for all $p \in P$ do
 3:     $X \leftarrow$ EXPAND($p$)
 4:     for all $x \in X$ do
 5:         if $x \sqsubset u \vee x \sqsupset u \vee x = u$ then
 6:             $rules \leftarrow rules \bigcup \{p\}$
 7:         end if
 8:     end for
 9: end for
10: for all $r \in rules$ do
11:     $rules \leftarrow rules \bigcup r.depends$
12: end for
13: return $rules$
```

Fig. 8. Trigger algorithm.

We consider that the updates are XPath expressions that specify the location of the nodes to be inserted or deleted. When an update u occurs we must determine the XML nodes that must be re-annotated. The idea is that the nodes that must be re-annotated are in the scope of the access control rules that are "related to" the update u. To discover this set of rules we run the TRIGGER algorithm which tests the containment between the query and the *expansion of policy* rules, and then adds the dependent rules based on the previously constructed dependency graph.

The complexity of this algorithm is $O(n \cdot h)$, where n is the number of rules and h the height of the XML document tree.

The need for *rule expansion* and *dependency resolution* can be supported with a simple example. Consider the XML tree in Figure 2 and the accompanying policy of Table 1. The rules R_1 and R_3 say that all patients are accessible except those that have a treatment as child element. Also consider that the incoming update query specifies the deletion of //patient/treatment nodes. After this operation one would expect that all patient elements are now accessible. To make this happen we should consider triggering the positive rule //patient (R_1) for the re-annotation process. This is accomplished in two steps: *(i)* rule R_3 expands to

$$\text{//patient[treatment]} \longrightarrow \begin{array}{l}\text{//patient}\\ \text{//patient/treatment}\end{array}$$

the latter part of which matches the query, *(ii)* the dependency resolution finds that positive rule R_1 is a dependent of R_3 (by means of containment) and consequently is included in the set of rules to consider. If the expansion had not taken place, the positive rule R_1 would not have been triggered and thus the previous annotations would have incorrectly been preserved.

This rule expansion does not cover the case in which the XPath expressions of the access control rules contain predicates with descendant axes. Consider the hospital document and rules R_1 and R_5. Consider an update that deletes all treatment elements (//treatment) and their subtrees. The query will not trigger any rules that do not contain the treatment tag. This is not right because patient elements should be accessible now, as there is no descendant experimental element under patient anymore. To deal with this problem, we need to replace all descendant axes that occur inside a predicate of an access control rule with relative paths using only the child axis. With the schema information these replacements are finite. Rule R_5 now expands to

$$\begin{array}{lll} \text{//patient[.//experimental]} & \longrightarrow & \begin{array}{l}\text{//patient}\\ \text{//patient//experimental}\end{array} \\ & \longrightarrow & \begin{array}{l}\text{//patient}\\ \text{//patient/treatment/experimental}\end{array} \end{array}$$

After the expansion, rule R_5 is triggered by the query. This triggers also rule R_1 because of containment, and accessibility of nodes is updated correctly.

The full picture of the re-annotation process can be perceived as a sequence of the steps described previously. The idea is the following: we first obtain the set of triggered rules by calling TRIGGER. We then produce an annotation query Q for this set of rules (using algorithm ANNOTATION-QUERIES). We then re-annotate the nodes that are in the semantics of Q as accessible.

6 Implementation

Our system transforms XML data and stores it in a relational database by shredding them using ShreX[1]. We modified ShreX to better interface with external code modules. For uniformity of evaluations we decided to use the MonetDB[2] database. MonetDB offers the advantage of providing both XML and supporting and SQL module. This permitted us to directly compare the two methods using the same engine. We also chose to evaluate our system using PostgreSQL[3]. We used the PL/Python feature, which enables PostgreSQL functions to be written in Python.

The core of the application that handles all the input and transactions was also written in Python. We used the `py-psycopg2` module for PostgreSQL and the `MonetSQLdb` module distributed with the MonetDB project.

In some cases we used object serialization and disk storage, to keep an algorithm's computation or a procedure's output for future use. For example, document shredding which is a very time consuming process, or containment comparisons, which are an issue mostly because our current implementation is in Java, and we must pay the cost of JVM initialization.

[1] `http://shrex.sourceforge.net/`
[2] `http://monetdb.cwi.nl/`
[3] `http://www.postgresql.org/`

7 Evaluation

7.1 Setup

factor	size (bytes) XML	SQL
0.0001	19K	33K
0.001	85K	149K
0.01	804K	1.6M
0.1	7.9M	17M
1.0	79M	78M
2.0	158M	140M
10.0	793M	310M

Table 5. Documents generated with `xmlgen` and their sizes.

To evaluate our system and runtime environment we used the following parameters: *(i)* size of the XML document, *(ii)* size of the policy, *(iii)* the coverage of the policy, and we designed our experiments to measure: *(i)* loading time, *(ii)* annotation time, *(iii)* response time, *(iv)* re-annotation time.

The XML data were generated with `xmlgen` from the XMark project [21]. We should also note that we modified `xmlgen`'s code that generates XML —and as a consequence the conforming schema— in an effort to eliminate all recursive paths. This is crucial for the specific shredding procedure to work properly. With `xmlgen` we generated a set of documents of variable sizei (see Table 5). The sizes of the respective SQL files are also displayed.

We manually designed policies with variable coverage, that is, we crafted several policy files to force our system to annotate increasingly larger portions of the data.[4] In this fashion we obtain information about the system's behavior when managing small or large number of annotations. We refer to these policies as the *coverage* policy dataset.

We shred the XML files to text files containing SQL INSERT statements representing the data. *Loading time* is the time needed to run these SQL files on a relational database. Similarly, with respect to native XML storage, loading time refers to the time needed to load the document from the XML file to the XQuery database. The annotation process for the relational store consists of evaluating the query obtained by algorithm Annotation-Queries, performing set operations on their results to determine the tuples that need updating, and finally execute UPDATE queries if needed as discussed in Section 5.2. *Annotation time* is the time required for these actions to complete. In a similar manner, for the XML store, we measure the time needed to evaluate the Annotation-Queries with MonetDB XQuery. *Response time* is the time needed to check whether a user has access to the data they request. Finally, *re-annotation time* is the time spent to get the database to a consistent state, after an update occurs.

We run our experiments on a Dell OptiPlex 755 Desktop with an Intel® Core™ 2 Duo CPU E8400 @ 3.00GHz with 3*GB* of memory, running FreeBSD 7.0.

7.2 Experimental Results

We run a series of experiments to evaluate the efficiency and feasibility of the system. Loading the documents in native XML form is fairly quick to complete,

[4] We evaluated the actual coverage percents with XQuery after each document annotation.

whereas running all the equivalent INSERTs is over one order of magnitude slower. Among the two relational databases, we found that PostgreSQL performs about twice faster than MonetDB/SQL when inserting data (see Figure 9).

In Figure 10 we show the performance on client requests. We run 55 different queries (of the same complexity as the *coverage* policy dataset) and calculated their average response time for each document. The required time is roughly analogous to the document size. MonetDB/SQL performs better than PostgreSQL on large documents, but compared to XQuery they both perform 34 times slower on average (and growing).

Figure 11 presents the results for variable coverage policies of the actual annotating process on all the database systems we used. There is a small performance gain on small documents when using relational databases, but in the long run the MonetDB/XQuery database performs best when annotating actions. There is little difference between MonetDB/SQL and PostgreSQL.

Optimization results in the case of re-annotation are shown in Figure 12. We run the same 55 queries (derived from the *coverage* dataset) as delete updates and calculated the average re-annotation time per document. The re-annotation time is not actually dependent to the document size. On XQuery, for documents of tens of MBs or larger it becomes efficient as a technique, and is 5 times faster than full annotation. On relational databases it is almost always more efficient to perform partial re-annotation, and on average it's 9 and 7 times faster on MonetDB/SQL and PostgreSQL respectively. Among the relational and native XML re-annotation, the latter is about twice as fast on average.

8 Conclusions

In this paper we have studied the problem of enforcing access control on XML documents stored in relational and native XML databases. We have presented a novel *re-annotation* algorithm that computes the XPath query which designates the XML nodes to be re-annotated when an update operation occurs. We performed exhaustive experiments to evaluate the effectiveness and efficiency of the proposed solutions. We concluded that performing access control on XML

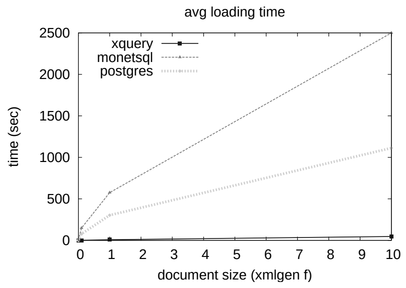


Fig. 9. Loading time comparison.

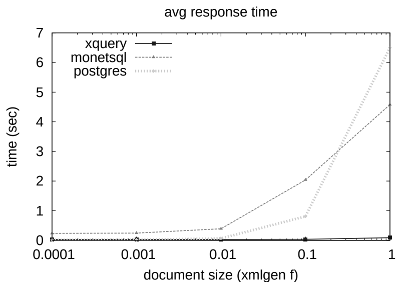
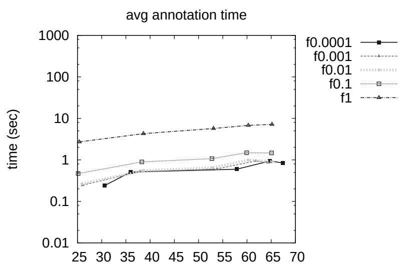
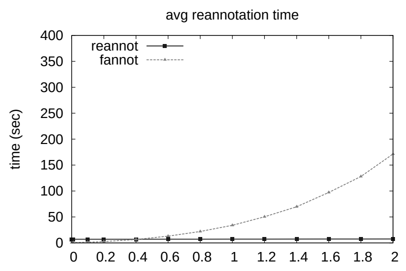


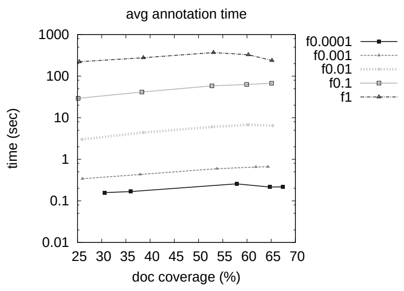
Fig. 10. Response time comparison.



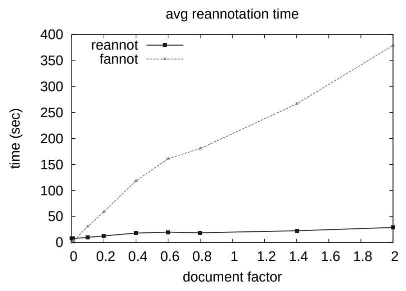
(a) MonetDB/XQuery



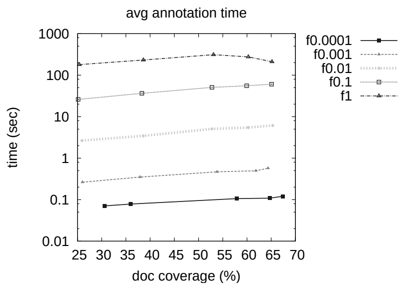
(a) MonetDB/XQuery



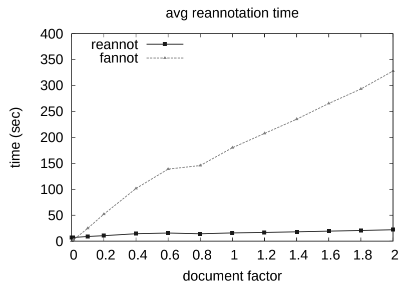
(b) MonetDB/SQL



(b) MonetDB/SQL



(c) PostgreSQL



(c) PostgreSQL

Fig. 11. Annotation time comparison.

Fig. 12. Reannotation vs full annotation.

documents stored in native XML databases outperforms the relational-based solution.

Schema-aware optimizations should be further studied, as they can extend our mechanism to support larger XPath fragments and produce more accurate results. As a future work, we also plan to extend our framework to handle access control for update operations (inserts and deletes).

9 Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS. We would like to thank James Cheney for interesting discussions during the first steps of this work and Loreto Bravo for insightful comments.

References

1. S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the XML-to-relational mapping problem. In *Proc. of the 6th Annual ACM Int'l workshop on Web Information and Data Management*, pages 31–38. ACM New York, NY, USA, 2004.
2. M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. *Theoretical Computer Science*, 336(1):3–31, 2005.
3. E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, 5(3):290–331, 2002.
4. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. `http://www.w3.org/TR/xmlschema-2/`, October 2004. W3C Recommendation.
5. S.R. Cho, S. Amer-Yahia, L.V.S. Lakshmanan, and D. Srivastava. Optimizing the secure evaluation of twig queries. In *Proc. of the 28th Int'l Conf. on Very Large Data Bases*, pages 490–501. VLDB Endowment, 2002.
6. J. Clark, S. DeRose, et al. XML path language (XPath) version 1.0. W3C recommendation, 1999. `http://www.w3c.org/TR/xpath`.
7. E. Damiani, S.C. Di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):169–202, 2002.
8. F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML documents in relational databases. In *Proc. of the 30th Int'l Conf. on Very large data bases-Volume 30*, pages 1297–1300. VLDB Endowment, 2004.
9. David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. `http://www.w3.org/TR/xmlschema-0/`, October 2004. W3C Recommendation.
10. W. Fan, C. Chee-Yong, and M. Garofalakis. Secure XML querying with security views. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD), Paris, France*, pages 587–598, 2004.
11. I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In *Proc. of the 9th ACM symposium on Access control models and technologies*, pages 61–69. ACM New York, NY, USA, 2004.
12. G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *Journal of the ACM*, 52(2):284–335, 2005.
13. K. Haj-Yahya. XPath-Containment Checker. Version: 2005. `http://www.ifis.uni-luebeck.de/projects/XPathContainment`.
14. S. Ioannidis. *Security policy consistency and distributed evaluation in heterogeneous environments.* PhD thesis, Philadelphia, PA, USA, 2005.
15. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proc. IEEE Computer Society Symposium on Security and Privacy*, pages 31–42, 1997.

16. G. Kuper, F. Massacci, and N. Rassadko. Generalized XML security views. *Int'l Journal of Information Security*, 8(3):173–203, 2009.
17. E. C. Lupu and M. S. Sloman. Conflict Analysis for Management Policies. In *Proc. of the 5th IFIP/IEEE Int'l Symposium on Integrated Network Management IM, San Diego, CA*, 1997.
18. G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.
19. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. *Lecture notes in Computer Science*, pages 315–329, 2003.
20. N. Qi, M. Kudo, J. Myllymaki, and H. Pirahesh. A function-based access control model for XML databases. In *Proc. of the 14th ACM Int'l Conf. on Information and Knowledge Management*, pages 115–122. ACM New York, NY, USA, 2005.
21. A. Schmidt, F. Waas, M. Kersten, M.J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. of the 28th Int'l Conf. on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.
22. T. Schwentick. XPath query containment. *SIGMOD RECORD*, 33(1):101, 2004.
23. K.L. Tan, M.L. Lee, and Y. Wang. Access control of XML documents in relational database systems. In *Int'l Conf. on Internet Computing*, pages 185–191. Citeseer, 2001.
24. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition. `http://www.w3.org/TR/xmlschema-1/`, October 2004. W3C Recommendation.
25. P. Wadler. Two semantics for XPath. Technical report, 2000.
26. T. Yu, D. Srivastava, L.V.S. Lakshmanan, and H.V. Jagadish. A compressed accessibility map for XML. *ACM Transactions on Database Systems (TODS)*, 29(2):363–402, 2004.
27. H. Zhang, N. Zhang, K. Salem, and D. Zhuo. Compact access control labeling for efficient secure XML query evaluation. *Data & Knowledge Engineering*, 60(2):326–344, 2007.