Incremental Adaptation of XPath Access Control Views

Abstract

Materialized XPath access-control views are commonly used for enforcing access control. When access control rules defining a materialized XML access-control view change, the view must be adapted to reflect these changes. The process of updating a materialized view after its definition changes is referred to as *view adaptation*. While XPath security views have been widely reported in literature, the problem of view adaptation for XPath security views has not been addressed. View adaptation results in view downtime during which users are denied access to security views to prevent unauthorized access. Thus, efficient view adaptation is important for making XPath security views pragmatic. In this work, we show how to adapt an XPath access-control view incrementally by re-using the existing view, which reduces computation and communication costs significantly, and results in less downtime for the end-user. Empirical evaluations confirm that the incremental view adaptation algorithms presented in this paper are efficient and scalable.

1 Introduction

Access control is an important component of maintaining security of data and information systems. Access control rules [6, 14, 18, 8] specify the parts of the data that users can access and parts that they cannot. Materialized access-control views are an efficient way of implementing access control. A materialized access-control view contains only the data to which an user or role has access. User queries are evaluated against this access-control view without the need for any further security check. In this paper, we use the terms "access-control views" and "security views" interchangeably.

Increasingly, XML is being used as a format for storing and exchanging data in a large number of applications such as web-based applications, e-commerce, and XML client-server databases. In all these applications, there is a need for access control. Recently, there is also an increasing interest in supporting fine-grained XML access control [6]. Towards this end, techniques such as XPath security views [7, 12, 24] are being used. Client-side XPath security views [5] and semantic caching [27] are popular in decentralised systems. Such remote views reduce communication costs by allowing user queries to be evaluated locally, but add to communication costs incurred while maintaining the view.

Two issues arise in the case of materialized views - one is the **view update** problem [10] or the problem of synchronizing materialized views with updates to the *base data*, and the other is the **view adaptation** problem [11] or the problem of synchronizing materialized views with changes to the view definition. This paper deals with the latter. Due to changes in organizational policies or security requirements, access control rules for a user (or role) can be updated from time to time. When access control rules change, the corresponding access-control view definition also changes, and the view must be *adapted* [11] to conform to the user's new access permissions. When access-control views are being adapted, users are denied access to the view to ensure that no unauthorized access to data occurs. This "downtime" could have important, often commercial, implications for several applications.

XPath access-control views have been widely reported in literature; however the problem of adapting them efficiently has not been addressed. Recently, work has been reported on updating XML views [13]. However, to the best of our knowledge, no work has been done on the problem of adapting XPath views. We believe that to make materialized security views pragmatic, efficient adaptation of such views is vital. This is because, in numerous applications and environments, there might be frequent updates to access control rules necessitating view adaptation. We outline two such scenarios below to motivate the *efficient XPath view adaptation* problem.

Example 1 (Dynamic Environment). In dynamic environments such as distributed, heterogeneous and web-based environments that use XML to store and integrate information from multiple sources, changes in the access rules and policies of the underlying data sources are not uncommon [25]. This could be the result of the dynamic nature of various organizational policies, changing roles, security requirements, etc. Such changes could necessitate changes to access-control views defined on the data. □

Example 2 (Schema Evolution). In applications like GIS, schemas evolve frequently [20], thus the semantics of the XPath views also change, necessitating view adaptation (discussed in Section 3.1). Mergers of organizations, establishment of new business alliances, changes in organizational structure, etc. are examples of scenarios where schema definitions, access control rules and policies undergo several changes before things stabilize [9]. In such cases, views would need to be adapted frequently. □

In both of the scenarios, changes to view definitions are frequent enough to result in downtime that is a nuisance to the end-user. However, there are substantial performance gains from materializing the views. In order to keep the downtime to a minimum, efficient view adaptation is necessary. There are two factors to be considered while adapting views efficiently: (1) *Computation cost* for adapting access-control views, which is critical for interactive applications [11]; and (2) *Communication cost* for transferring data to adapt remote security views, e.g., in client-server systems, and client-based access control [5].

When view definitions change, obvious but naive way is to simply re-compute the views from the base data. An alternative is to"incrementally" adapt the view by fetching only the data that is not already present in the view. We refer to this technique as **incremental view adaptation.** [11]. That is, the incremental adaptation of XPath access-control views is the focus of our work. Incremental view adaptation reduces computation and communication costs by exploiting the following observations:

Observation 1 *If a new view is entirely* contained *in an old view, no changes to the materialized view are necessary.* □

Observation 2 *If a new view is partially* overlapping *with an old view, only the difference needs to be materialized.* □

Considerable work has been done on the view update problem in the relational domain [23]. Gupta et al. [11] have reported work on incrementally adapting relational views. Due to the semi-structured and hierarchical nature of XML, the problem of adapting XPath views is different from the problem of view adaptation in relational databases. Furthermore, XPath access-control views present additional challenges because typically these views are defined using both positive and negative rules [6, 14, 8]. Typically, non-security views do not have negative rules in their definitions. The presence of negative access control rules makes adaptation more complicated for

access-control views. To the best of our knowledge, no prior work has addressed the problem of XPath view adaptation.

Our key contributions in this paper are as follows:

1. We present a set of comprehensive incremental view adaptation techniques for XPath access-control views expressed for the $XP^{\{/,//,*,[]\}}$ (the fragment of XPath with step, descendant, wild-card and predicate operators) that reduce computation cost and optimize communication cost. We also exploit XPath containment [16] and query answering using views [27] for efficient adaptation; these issues have not been addressed by previous work in relational databases [11].

2. We also suggest auxiliary information that can be stored to improve the performance of our incremental view adaptation techniques.

3. We demonstrate the superiority of incremental view adaptation over view re-computation experimentally.

The rest of the paper is organized as follows. In Section 2, we discuss related work. We present background information and details about the setting where our algorithms can be applied in Section 3. We define the XPath view adaptation problem in Section 4. The incremental view adaptation algorithms are presented in Section 5. We present a validation of our algorithms in Section [**?**]. The paper is concluded in Section 7.

2 Related Work

XPath views for implementing security or access-control for XML documents has been widely reported in literature. XPath security views were first proposed by Stoica and Farkas [24]. Fan et al. [7] propose algorithms for deriving security view definitions from security policies, and present techniques for efficient query processing on security views. Kuper et al. [12] further generalize the notion of XML security views by creating a "view DTD" that hides the base data DTD from users. Yu et al. propose a method using compressed XML views to support access control [28] for reducing the storage overhead for materialized views.

Bouganim et al. [5] talk about client-based access control for XML documents where the access-control is monitored at the client side. This model is suitable for environments where clients do not place sufficient trust in their data service provider.

Several models for specifying access control rules for XML access control have been mentioned. The model proposed by Damiani et al. [6] is very popular and Fundulaki et al. [8], Fan et al. [7], Kuper et al. [12] and Luo et al. [14] propose models that are similar to [6]. Lim et al. [13] propose a more sophisticated priority-based access control model. Recently, some new standards for XML access control, such as XACML have also emerged.

Gupta, et al., [11] originally introduced the problem of view adaptation for relational databases. They consider all possible "redefinitions" of SQL SELECT-FROM-WHERE-GROUPBY-HAVING, UNION, EXCEPT views and show how such views can be adapted using old materializations wherever possible. They identify auxiliary information to be stored with the views to assist redefinitions. Bellahsene also proposes a "fragment-based" approach for relational databases to view adaptation in a setting where fragments of a view are materialized instead of the complete view [27]. They propose a method where adaptation is performed using not only the old view but all the materialised views in the system.

A significant amount of work has been done in the area of XML view maintenance problem when the data in the database changes [21, 13]; however, to the best of our knowledge, no work has been done on the view adaptation problem for XPath views.

Chen, et al., [38] have proposed a cache-aware XQuery answering system that checks whether a query can be answered using cached results of previous queries. This problem is similar to ours in that the cached query results are similar to our views. Their algorithm is based on the existence of a single containment mapping from the query to the view and a set of heuristics. Consequently, their algorithm is not complete. Balmin, et al., [4] have also presented sound but incomplete algorithms for using materialized XPath views for query processing using XML values, full paths and node references. Mandhani and Suciu [16] and Xu and Ozsoyoglu [27] have shown (independently) how a query can be answered using a view. We use their technique in our view adaptation algorithm. Note that there exists no algorithm in the literature that can compute the rewriting of an XPath query using multiple views. However, when an algorithm to rewrite an XPath query using multiple views is developed, our algorithms can utilize it seamlessly.

Miklau and Suciu addressed the XPath containment problem and prove that it is co-NP-complete [17]. They provide a sound and complete exponential time algorithm for XPath containment, and some parameterized polynomial time algorithms. Neven and Schwentick [19] have shown the complexity of XPath containment in the presence of disjunctions and DTDs. Schwentick [22] provides algorithms to test for the containment of XPath in the presence of disjunctions using several techniques, like tree automata.

Note that these complexities are typically with respect to the size of the XPath query and the size of the view definitions, *and not in the size of the data in the view or the documents in the database*. Hence, in practice, the algorithms utilizing these techniques complete in reasonable time.

3 Our Setting

3.1 XPath View-Based Access Control

To make our view adaptation technique pragmatic, we adopt a popular, very commonly used XML-access-control language similar to the models proposed by [6, 14, 8, 18] for specifying access control rules.

An XML document is represented as a hierarchy of nested nodes (elements and attributes) and fine-grained access control is established at the node level. In our model, the node-level authorization is specified via 5-tuple *access control rules* [6, 14, 8]:

$$access\ control\ rule := \{subject, object, action, sign, type\}$$

where (1) *subject* is to whom an access is granted or denied (depending upon the sign of the rule: + or -), i.e., user or role; (2) *object* refers to nodes in XML documents specified by an XPath expression (XPath can be used to identify nodes in an XML document); (3) *action* is one of "read", "write", and "update"; (4) *sign*+, - indicates whether access is "granted" (positive rule) or "denied" (negative rule), respectively; and (5) *type* LC, RC refers to either Local Check (i.e., acess is granted or denied to only attributes or textual data of nodes in context, i.e., `self::text() | self:attribute()` in XPath), or Recursive Check (i.e., access is granted or denied to current nodes and propagated to all their descendants `descendant-or-self::node()`, respectively). For a node in a document to be in a view, under the recursive check semantics, an ancestor of the node must be included by a positive rule and none of its ancestors should be excluded by a negative rule.

By default, access is denied to all nodes whose authorizations are not specified, either explicitly (via LC rules) or implicitly (via RC rules). A node can have more than one relevant rule. If a

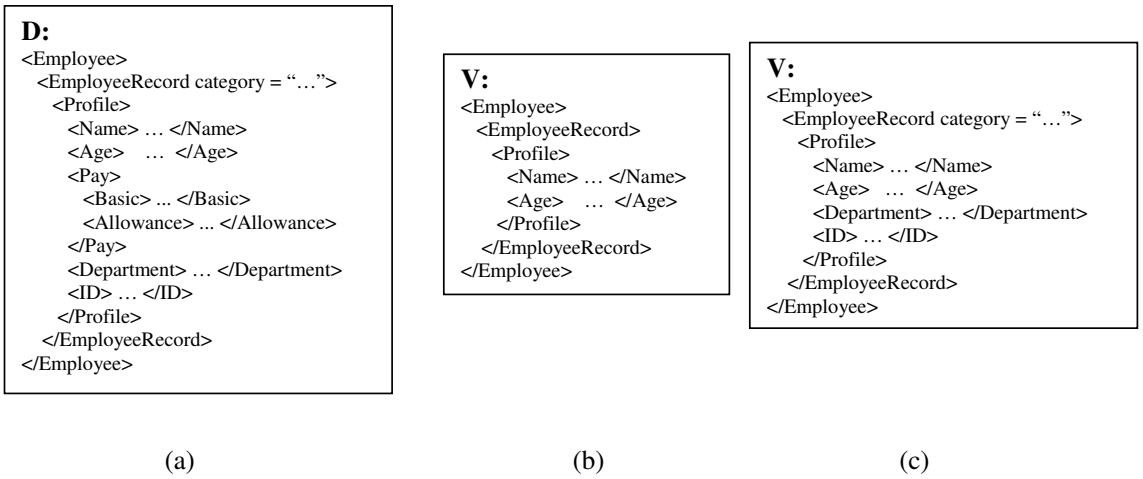


Figure 1: A running example showing: (a) XML document (b) XML security view. (c) Recursive Check.

conflict occurs between a positive (+) and negative (-) rule, the negative rule takes precedence. For our work, we consider only *read-action rules with recursive check*. Consequently, in the rest of the paper, we will refer to only the object and the sign of a rule.

Example 3. Consider the view V shown in Figure 1(c) defined on D from Figure 1(a). If the view is defined using only the positive rule: /Employee/EmployeeRecord and a negative rule */Employee/EmployeeRecord/Pay*, then V would contain *EmployeeRecord* and all its children, but not the *Pay* element, or its *Basic* and *Allowance* children. □

Changes to access control rules are not the only reason why views may need to be adapted. To explain how changes to XML schema induce changes in view definitions, consider the following example. Suppose the IDs for all the employees were updated to include their social security number, then views defined using the positive rule */Employee/EmployeeRecord/profile/** would have access to social security information which was not originally intended . If (some of) the views are not supposed to have access to such confidential information, the system administrator will have to add */site/people/person/profile/ID* as a negative rule for all those views (that do not already have this as a negative rule). This would induce changes in view definitions and would necessitate view adaptation. Such changes would also be common in applications like GIS where schemas are dynamic (dynamic schema evolution) [20].

4 The XPath View Adaptation Problem Definition

An access-control view contains parts of XML documents that are included by the positive (+) access control rules (denoted ACR^+ but **not** included by the negative (-) access control rules (denoted ACR^-).

Given a set of positive access control rules $ACR^+ = p_1, p_2, \ldots, p_n$, and set of negative access control rules, $ACR^- = n_1, n_2, \ldots, n_k$, the corresponding access control view V is defined by the expression:

$$V = (p_1(D) \cup p_2(D) \ldots \cup p_k(D)) -^D (n_1(D) \cup n_2(D) \ldots \cup n_k(D))$$

where p_i and n_i are XPath expressions belonging to the fragment XP{/,//,*,[]}. Updates that cause view definition change include the following events: (1) *Removal of a positive rule*: removing that the view previously had access to. (2) *Addition of a positive rule*: potentially adding data. (3) *Removal of a negative rule*: potentially allowing access to more data. (4) *Addition of a negative rule*: potentially removing access to data. Formally, we study the following problem:

Definition 1 (XPath access control view adaptation problem) *Given an access control view V, whose definition changes due to an update of one of the four types specified above to the corresponding access control rules, adapt the view V such that it satisfies the confidentiality criteria (defined in Section 3.2).*

4.1 Security Criteria

We have two main security criteria:

- *Confidentiality*: At time t, the system's access-control state consists of two important elements: (C1) the current set of access-control rules including all the changes that have been proposed by time t, and, (C2) the set of views that are currently accessible. Note that C2 does not include views that are NOT accessible at time t(because they are undergoing adaptation).

  The confidentiality criteria says that at any point of time t, C2 will not violate C1, i.e., no accessible view in C2 will contain data the user is not allowed to access at time t by C1.

- *Denial of service (or downtime)*: When access control rules defining a view undergo a change threatening confidentiality, the view is usually "shut down" to prevent loss of confidentiality, adapted, and then put back online. During this time, users are denied access to the view.

4.2 XPath Set and Deep-Set Operators

In this section, we outline operators that are used for manipulating XPath documents.

For the XPath *union* operator, we follow the same semantics as defined by the W3C [3]. That is, the semantics of $X1(D) \cup X2(D)$ is defined as the union of two node sequences returned by X1(D) and X2(D). $X1(D) \cup X2(D)$ takes two node sequences as operands and returns a sequence containing all the nodes that occur in either of the operands [3]. The standard XPath *intersect* operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands [3].

However, in the context of XML access control with recursive semantics where the views are defined using not only the nodes but the entire subtrees rooted at the nodes, the formal semantics of regular XPath set operators is not sufficient. Suppose, as shown in Figure 2, there is a positive rule $R1 : (admin, /x/a[e], read, +, RC)$ and a negative rule $R2 : (admin, //d[f], read, -, RC)$. Then, the *admin* can read the data specified by $/x/a[e]$ 'minus' the data specified by $//d[f]$. However, the default except operator of XPath cannot capture the notion of 'minus' correctly because it is based on the node-IDs and ignores subtrees completely. Here, data corresponding to rules $R1$ and $R2$ are indicated by the two rectangular boxes. The *admin* should have access only the nodes with node-IDs: $2, 3, 4, 8$. However, when the 'minus' is expressed using the XPath except operator, $/x/a[e]$ *except* $//d[f]$ is operated in terms of node-IDs as in $\{2\}$ *except* $\{5\} = \{2\}$. Then, the returned answers are the nodes with node-ID with '2' and all of its descendants $3, 4, 5, 6, 7, 8$. However, this is problematic because the answer contains the nodes $5, 6, 7$ violating access control rules.

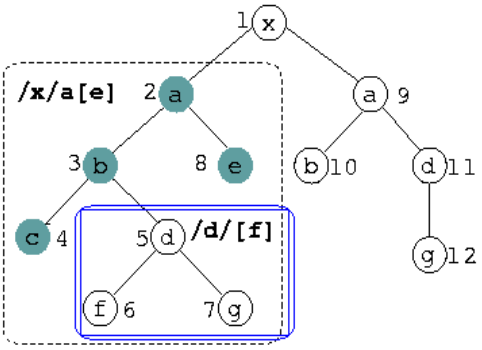


Figure 2: Example XML data D and two rules `/x/a[e]` and `//d[f]`

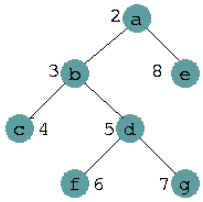
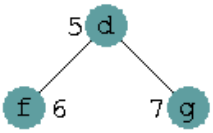
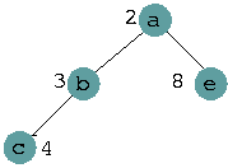
null			
/x/a[e] ***intersect*** **//d[f]**	**/x/a[e]** ***except*** **//d[f]**	**/x/a[e]** ***deep-intersect*** **//d[f]**	**/x/a[e]** ***deep-except*** **//d[f]**

Figure 3: XPath set and deep-set operators.

Similarly, we need to extend the *intersect* operator as well. The deep-set operators with extended semantics are denoted as deep-except ($-^D$), and *deep-intersect* ($-^D$). For instance, the correct semantics for P1 $-^D$ P2 is: (1) when answers to P2 are descendants of those to P1, nodes in the answer to P2 are excluded from the answer to P1 $-^D$ P2 ; and (2) otherwise, it degenerates to the regular *except* operator, $P1$ *except* $P2$. As an example, Figure 3 illustrates different results of the same query when both standard and extended set operators are used for D of Figure 2. Hereafter, we assume that there exist efficient implementations of deep-set operators. In our experimentations, we used the implementation provided by Luo, et al.,[15]. Since they are implemented as user-defined functions of XML databases, they do not require any extra support from underlying XML databases.

4.3 XPath Containment and Rewriting

When a view definition is updated, our algorithm must determine whether the new view can be constructed using the existing views. If the view can be adapted using the old views, then there is no need to fetch data from the database and the incremental adaptation can be performed quickly. The problem of identifying which nodes in the old XPath views satisfy the new rule (added or deleted) is closely related to the problem of answering XPath queries using XPath views [27], which, in turn is closely related to the problem of XPath containment [19]. We define these problems below.

We have adapted the definition of XPath containment provided by Neven and Schwentick [19]:

Definition 2 (XPath Containment) *p is contained in q, if for all XML documents, for all nodes $t \in p$, then $t \in q$. That is, if p is contained in q, then, if t is an answer to p, then t is also an answer to q.* □

Example 4. Given the following queries: p = /Employee/EmployeeRecord/Pay/Basic and q = /Employee/EmployeeRecord/Pay/*, p is contained in q. For the queries: p = /Employee/ EmployeeRecord/Pay/Basic and q = /Employee/EmployeeRecord[/age > 50]/Pay/Basic, p is not contained in q. □

Mandhani and Suciu [16] define the query rewriting/answerability problem as follows:

Definition 3 (Query Rewritability Using a View) *Given a view V and query Q, does V answer Q? If yes, then what should the query C be so that C(V) ≡ Q. That is, C executed on V(D) yields the same answer as Q executed on D, for any set of XML documents D.* □

Example 5. Let D be the XML Document D in Figure 1, V(D) = /Employee/EmployeeRecord/Pay, Q = /Employee/EmployeeRecord/Pay[/Basic], and C = Pay[/Basic]. Executing the *compensation query C* on *V(D)* clearly yields the same results as executing Q on D. □

Suppose Q above was a negative rule being added to the view V(D) defined by a single positive rule /Employee/EmployeeRecord/profile/Pay. Q is re-writable using V(D) and the compensation query C. So Q can be executed on V(D) to determine what parts of V(D) must now be removed due to the addition of the negative rule Q.

Note that although we know that when a negative rule is added, we are removing data from the view that is already contained in the view, it might not always be possible to determine which nodes constitute that data. Consider the following example.

Example 6. Let ACR^+ = {/Employee/EmployeeRecord/profile/Pay}, ACR^- = {}, and, R = /Employee/EmployeeRecord/profile[age>50]/Pay. Although we know that the <Pay> elements for all employees (including the ones aged above 50) are contained in the view, when the negative rule R is added, it is not possible to determine from the view alone which >Pay> elements must be deleted and which must remain. This is because R is not rewritable using the view, although it is contained in the view! □

Example 7. Consider a view that is defined using the following rules:
ACR^+ = {/Employee/EmployeeRecord/profile/Pay, /Employee/EmployeeRecord/profile/age}, and ACR^- = {}. $R = /Employee/EmployeeRecord/profile[age > 50]/Pay$ is a negative rule being added to the view. Then using the positive rule /Employee/EmployeeRecord/profile/Pay or /Employee/EmployeeRecord/profile/age alone, R is not rewritable. To rewrite R using multiple views, one possible approach is to store node IDs for each $< Pay >$ element and $< age >$ element to determine their common $< profile >$ parent and store them together under $< profile >$ element (as shown in Example 9 below). This is because, for each $< Pay >$ element the corresponding $< age >$ element can be determined and checked to see if its value is greater than 50. If it is, then the corresponding $< Pay >$ element would appear in the result set. The path from the root for the profile element must be stored as well to determine the parent of each element. The view contains:
$\langle profile \rangle (path : /Employee/EmployeeRecord/profile) \langle Pay \rangle \ldots \langle /Pay \rangle \langle age \rangle \ldots \langle /age \rangle \langle /profile \rangle$
$\langle profile \rangle (path : /Employee/EmployeeRecord/profile) \langle Pay \rangle \ldots \langle /Pay \rangle \langle age \rangle \ldots \langle /age \rangle \langle /profile \rangle$

If the view stores all the profile elements with their Pay and age children as shown above, then the compensation query for R would be /profile[age¿50]/Pay. □

However, sometimes, although an expression is re-writable using the positive rules in a view, the presence of negative rules might still make not allow the expression to be rewritten using the view.

Example 8. Consider the following example where a negative rule R is added to the view: ACR^+ = /Employee/EmployeeRecord/profile/* ACR^- = /Employee/EmployeeRecord/profile/age R = /Employee/EmployeeRecord/profile[age>50]/name is a negative rule being added to the view. R is rewritable using /Employee/EmployeeRecord/profile/* (ACR^+). However, due to /Employee/EmployeeRecord/profile/age (ACR^-), the view does not contain the $< age >$. element of the employee.s profile. So if R, which is re-writable using ACR^+, is executed on the view, it would not be able to determine which name elements belong to employees whose $< age >$ is above 50, and which elements belong to employees who belong to the IST department. Hence R is contained in the view, is re-writable using ACR^+, but is not answerable using the view. □

Example 9. Consider the view defined using the following rules: ACR^+ = {/Employee /EmployeeRecord/profile/*}, and, ACR^- = {/Employee/EmployeeRecord/profile[age>50]/pay}, and, the query: R = /Employee/EmployeeRecord/profile[age < 25]/name. R is answerable using the view, because R is rewritable using ACR^+ and R is not contained in ACR^- . However, if R was /Employee/EmployeeRecord/profile[age > 75]/name then R would not be answerable using the view. □

Since access control view definition can contain multiple positive and negative rules, XPath query containment and rewriting must be computed for unions of XPath expressions. Neven and Schwentick have shown that by adding disjunction the problem of containment of XP{/,//,[],*} remains in CO-NP [19]. Our algorithms (described in Section 5) use containment of $XP^{\{/,//,[],*,|\}}$ in the presence of unions. This containment can be checked using tree automata as discussed in [22].

The techniques proposed independently by Xu and Ozsoyoglu [27], and Mandhani and Suciu [16], can be used to answer an XPath query using a single view. In their work, a view is defined using only one XPath expression. In the case of access control views, a view is defined using multiple positive (and negative) rules. In the simplest case, where no negative rules are present, an access control view, in our work, can be defined using multiple positive rules, which is equivalent to a *union of multiple views*.

Rewriting XPath queries using multiple XPath views belonging to the XPath fragment $XP^{\{/,//,*,[]\}}$ is still being researched and there is currently no known algorithm to solve this problem. Developing techniques to solve the rewriting problem for multiple views is distinct from the problem of view adaptation.

Our algorithms are currently designed to utilize the algorithms for generating a rewriting using a single XPath view-rule. If an algorithm for rewriting an XPath query using multiple views is developed in the future, then, it can be plugged seamlessly into our framework with minor, straight-forward modifications of our algorithm.

4.4 Cost Models

A view adaptation algorithm has the following costs: (a) *Computation cost:* The time taken to adapt the view. (b) *Communication cost:* The time taken to send the data needed for adaptation from the source database to a remote view.

In applications, where the materialized views are stored at the site of the base data itself, there is no or very little communication cost involved, the computation cost for view adaptation is important.

In systems, such as distributed or peer-to-peer databases, implementing client-based access control [5], views reside at the clients while the source database resides on a server (or remote site). Client-based access control is popular in (1) systems where clients do not trust database service

providers to preserve data confidentiality, protect children from suspicious internet content, etc.; (2) decentralized data sharing systems like peer-to-peer databases. In such cases, the cost for actually computing the view might be insignificant when compared to the communication costs involved. In this case, there are fixed communication costs like latency, etc., and variable communication costs proportional to the size of the data. For large datasets, the latter costs dominate. Our algorithms do not have any effect on the latency of the communication anyway, so we focus on minimizing the data communicated to reduce the communication cost.

Although, the amount of data fetched from the database has an impact on the computation cost, in some cases compromising on the amount of data fetched from the database might lead to improved performance due to fewer operations for computing the adapted view.

5 XPath View Adaptation Algorithms

In this section, we discuss the various view adaptation algorithms. We denote the base data as D.

5.1 Naive View Adaptation

In the naive method, the view is re-computed by executing the XPath expression in Equation 1 on the base data. We call this approach the nave view re-materialization approach. Although simple, this approach is not efficient as it involves many redundant computations (explained further in Section 5.2). The equation for the naive re-computation is given below:

$$(p_1(D) \cup p_2(D) \cup \ldots \cup p_k(D)) -^D (n_1(D) \cup n_2(D) \cup \ldots \cup n_k(D)) \quad (1)$$

where $p_i \in ACR^+$ and $n_i \in ACR^-$.

5.2 Optimized View Adaptation

The expression for optimized view re-materialization is shown in Equation 2 below. This method is more efficient than the nave view re-materialization, because instead of computing all the XML nodes contained in $(p_1(D)) \cup p_2(D) \cup \ldots \cup p_k(D))$ and $(n1(D) \cup n2(D) \cup \ldots \cup nk(D))$ first, and then performing a deep-except operation on them, this scheme ignores all the (parts of) rules in ACR- that do not intersect with ACR+ at all, and hence have no impact on the view. Although some negative rules might not intersect with any positive rules in the view definition, they are still defined to ensure that at a later point, when a positive rule is added, it does not bring in data that is not meant to be viewed by the user (role). The optimized equation for computing the contents of the adapted view is as follows:

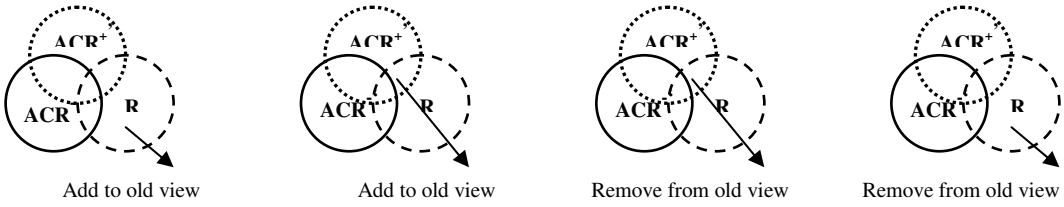
$$(p_1(D) \cup p_2(D) \cup p_k(D)) -^D (ACR^- -^D p_i)(D) \quad (2)$$

where $p_i \in ACR^+$

5.3 Incremental View Adaptation

In this section, we discuss techniques for optimizing the communication cost for XPath security view adaptation. The examples used in this section are based on the schema in Figure 1. Figure 4 explains how we determine the incremental data to be added or removed from the view for all four cases.

In the equations provided below for the different cases, the "Default Rule" is executed at the database. Hence we have provided the equations that minimize the data brought in from the



(a) (b) (c) (d)

Figure 4: View Adaptation: (a) Adding a Positive rule R to ACR+; (b) Deleting a Negative rule R from ACR-; (c) Adding a Negative rule R to ACR-; and (d) Delete a Positive Rule R from ACR+.

database for the "Default Rules". In all the other rules, we have stated the equations that minimize the computation.

5.3.1 Addition of Positive Rules

Addition of a new positive rule R to ACR^+ is processed using the following rules in order:

- [**Containment Rule**]: If R is contained in $ACR^+ \cup ACR^-$, i.e. R is either completely contained in a positive rule in the view, or is disallowed by some negative rule from the view, then do nothing and return.

  Example 10. Let ACR^+ = /Employee/EmployeeRecord/profile/*, ACR^- = /Employee /EmployeeRecord/profile/age, and R = /Employee/EmployeeRecord/profile/name. Then, R is already contained in ACR+, and thus no changes to the view is necessary. □

  Example 11. Let ACR^+ = /Employee/EmployeeRecord/profile/*, ACR^- = /Employee /EmployeeRecord/profile/age, and R = /Employee/EmployeeRecord/profile/age. Then, R is contained in ACR^- and thus no changes to the view is necessary because ACR- prohibits R. □

  If the containment rule is not applicable, then R potentially adds data to the view. Execute the default rule.

- [**Default Rule**]: Execute Equation 3.

$$V'(D) = V(D) \cup (R(D) -^D (R'(D) \cup R"(D))) \tag{3}$$

  Where $R' = R \cap^D ACR^+$ and $R" = R \cap^D ACR^-$

  Example 12. Let ACR^+ = {/Employee/EmployeeRecord/profile/name}, ACR^- = {/Employee /EmployeeRecord/profile/age}, and, R = /Employee/EmployeeRecord/profile/*. Then R' = /Employee/EmployeeRecord/profile/name, and R" = /Employee/EmployeeRecord/profile/age in Equation 3. Upon execution of Equation 3 from the database, the expression adds the children of the node /Employee/EmployeeRecord/profile except the nodes $< name >$ that is already there and $< age >$ that is excluded by the negative rule. □

5.3.2 Deletion of Positive Rules

Deletion of a positive rule R from ACR^+ is handled using the following rules in order:

- [**Containment Rule**]: If R is contained in $ACR^{+'} \cup ACR^-$, which means that removal of R from ACR^+ does not remove any data from the view, do nothing and return.

  Example 13. Let ACR^+ = {/Employee/EmployeeRecord/profile/*, /Employee /EmployeeRecord/profile/age}, $ACR^{+'}$={/Employee/EmployeeRecord/profile/*}, and R = /Employee/EmployeeRecord/profile/age. R is contained in $ACR^{+'}$, and $ACR^{+'}$ is still in the view definition, so the view needs no change. □

- [**Self-maintaining Rule**]: If R does not contain predicates, execute Equation 4:

$$V'(D) = V(D) -^D (R(V(D)) -^D R'(V(D))) \tag{4}$$

  Where R' = R $\cap^D$ $ACR^{+'}$. Deletion of a positive rule may potentially remove data from a view. This data may be identified using the view itself by executing the removed rule on the view.

  Example 14. Let ACR^+ = {/Employee/EmployeeRecord/profile/gender, /Employee /EmployeeRecord/profile/age}, and R = /Employee/EmployeeRecord/profile/gender. Clearly, executing R on the view returns all the $\langle gender \rangle$ elements and they can then be deleted. However, if R contains predicates, we may not be able to determine the answer of R using the view. □

- [**Default Rule**]: Execute Equation 5:

$$V'(D) = V(D) -^D (R(D)) -^D (R'(D) \cup R"(D))) \tag{5}$$

  Where R' = R $\cap^D$ $ACR^{+'}$ and R" = R $\cap^D$ $ACR^{-'}$. The expression $(R(D)) -^D (R'(D))$ is evaluated from the database.

Example 15. Consider the following R that allows access to names of employees whose salary is less than 20,000 and greater than 50,000:
ACR^+ = {/Employee/EmployeeRecord/personal[@$salary\rangle 50,000$]/name,
/Employee/EmployeeRecord/personal[@$salary\langle 20,000$]/name}. Let us say we drop the first rule. Though the names of all the employees whose salaries are greater than 50,000 are in the view, we have no way of determining from the view who has a salary greater than 50,000 and delete those names because the salary values are not saved in the view along with the names. Thus, we have to use the default rule and go to the database. □

5.3.3 Addition of negative rules

Addition of a new negative rule R to ACR^- is processed using the following rules in order:

- [**Intersection Rule**]: If $R \cap^D ACR^+ = \phi$, then do nothing and return. R does not intersect with any positive rule in ACR^+ and hence the addition of R does not affect the view.

  Example 16. Let ACR^+ = {/Employee/EmployeeRecord/profile/name}, ACR^- = {/Employee /EmployeeRecord/profile/gender}, and, R = Employee/EmployeeRecord/profile/age. R does not intersect with any rule in ACR^+ , so the view needs no update. □

- [**Containment Rule**]: If R is contained in ACR^-, then do nothing and return. R is contained in ACR^-. Thus, the view needs no change.

  Example 17. Let $ACR^+ = \{$/Employee/EmployeeRecord/profile/name$\}$, $ACR^- = \{$/Employee /EmployeeRecord/profile/gender, /Employee/EmployeeRecord/profile/Pay/*$\}$, and R= /Employee/EmployeeRecord/profile/Pay/Allowance. R is contained in "/Employee/EmployeeRecord/profile/Pay/*". So the view needs no change. In the absence of predicates, the negative rule R can be simply executed on the view and the data satisfied by R can be deleted from the view. □

- [**Self-maintaining Rule**]: If R has no predicates, then execute Equation 6:

$$V'(D) = V(D) -^D R(V(D)) \tag{6}$$

  Example 18. Let $ACR^+ = \{$/Employee/EmployeeRecord/profile/Pay/*$\}$,
  $ACR^- = \{$/Employee/EmployeeRecord/profile/gender$\}$, and,
  R = /Employee/EmployeeRecord/profile/Pay/Allowance. In this case, all nodes in the view V(D) that satisfy R are deleted from V(D) to obtain the new view. Note that in this case, *the view can be adapted using only the old view*, without needing to access the base data. □

  In the presence of predicates, it may not be possible to adapt the view using the old view and the database must be consulted. Consider the following example:

  Example 19. Let $ACR^+ = \{$/Employee/EmployeeRecord/profile/Name$\}$, and R = /Employee/EmployeeRecord/profile[@*salary*$\rangle 50,000$]/Name. In this case, it is not possible to determine which employees satisfy the salary criterion and whose names should be deleted using the view or the existing rule views corresponding to ACR^+. For another example, see Example 8. □

  Then, the default rule is applied.

- [**Default Rule**]: Execute Equation 7

$$V'(D) = V(D) -^D (R'(D) -^D R"(D)) \tag{7}$$

  where R' = R $\cap^D$ ACR^+ and R" = R' $\cap^D$ $ACR^{-'}$. The expression $R'(D) -^D R"(D)$ is executed at the database, the resultant data shipped to the view and the view is adapted by excluding this data from the old view.

5.3.4 Deletion of Negative Rules

Deletion of a negative rule R from ACR^- is processed using the following rules:

- [**Intersection Rule**]: If $R \cap^D ACR^+ = \phi$, do nothing and return. R does not intersect with any of the positive rules currently in ACR^+ and hence removal of R does not affect the view.

  Example 20. Let $ACR^+ = \{$/Employee/EmployeeRecord/profile/name$\}$, $ACR^- = \{\}$, and, the negative rule being deleted is: R = /Employee/EmployeeRecord/profile/gender. Because the view does not have any gender information, the negative rule does not require the view to be adapted.

- [**Containment Rule**]: If R is contained in $ACR^{-'}$, do nothing and return. The removal of R does not affect the view because there is another negative rule that rules out all the data that R was ruling out.

  Example 21. Let $ACR^{+} = \{$/Employee/EmployeeRecord/profile/*$\}$, $ACR^{-} = \{$/Employee /EmployeeRecord/profile/Pay/*, /Employee/EmployeeRecord/profile/Pay/Allowance$\}$, and R = /Employee/EmployeeRecord/profile/Pay/Allowance. R is contained in $ACR^{-'}$. The other negative rule prevents access to all children of $< Pay >$, so the access to $< Pay/Allowance >$ is also prevented. □

- [**Default Rule**]: Execute Equation 8:

$$V'(D) = V(D) \cup (R'(D) -^{D} R"(D)) \tag{8}$$

  where $R' = R \cap^{D} ACR^{+} and R" = (R' \cap^{D} ACR^{-'})$. When $R -^{D} ACR^{+} \neq \phi$, the removal of R could potentially add data to the view. The expression $(R'(D) -^{D} R"(D))$ is evaluated using the database.

  Example 22. Let $ACR^{+} = \{$/Employee/EmployeeRecord/profile/*$\}$, $ACR^{-} = \{$/Employe /EmployeeRecord/profile/Pay/*, /Employee/EmployeeRecord/profile/Pay/Allowance$\}$, and, R = /Employee/EmployeeRecord/profile/Pay/*. In this case, *R' = /Employee/EmployeeRecord /profile/Pay/** and *R" = /Employee/EmployeeRecord/profile/Pay/Allowance* in Equation 8.□

5.4 Auxiliary Rule Views

Using additional materialized data can enable view adaptation without having to go to the database. An option is to maintain rule views. A rule view is an intermediate view constructed using only a single positive or negative rule and materialized. Rule views are not made available to users, but only kept internally to speed up view adaptation. We construct one rule view for each rule in ACR^{+} and ACR^{-}. Rule views are especially useful when the data necessary to adapt a view cannot be obtained using the view (explained below). We show the equations below only for the reducing computation costs; whenever the view alone cannot be used for adaptation, communication costs will be incurred to querying the rule views at the base data site.

- Addition of a positive rule R: If R can be rewritten using the rule views, then a trip to the database can be avoided by executing the Default Rule on the rule views

$$V'(D) = V(D) \cup (R(D) -^{D} R'_1(\bigcup_i R_i(D))) \tag{9}$$

  where $R' = R \cap^{D} ACR^{+}$, and R'_1 is a rewriting of R',using the union of the rule views $\bigcup_i R_i(D)$.

- Deletion of a positive rule R: If R can be rewritten using the rule views, then execute:

$$V'(D) = V(D) -^{D} (R(\bigcup_i R_i(D)) -^{D} (R'_1(\bigcup_i R_i(D))) \tag{10}$$

  where $R' = R - DACR^{+'}$ and R'_1 is a rewriting of R' using $\bigcup_i R_i(D)$

- Addition of a negative rule R: If a re-writing of R' and R" using the union of all rule-views exists, then execute:

$$V'(D) = V(D) -^D (R1'(\bigcup_i R_i(D))) \tag{11}$$

where $R' = R \cap^D ACR^+$ and R'_1 is a rewriting of R' using $\bigcup_i R_i(D)$

- Deletion of a negative rule R: If a rewriting of R' and R" using the union of all rule-views exists, then execute:

$$V'(D) = V(D) \cup (R'_1(\bigcup_i R_i(D)) -^D R1'(\bigcup_i R_i(D))) \tag{12}$$

where $R' = R \cap^D ACR^+$ and $R' = (R \cap^D ACR^+ \cap^D ACR^{-'})$ and R'_1 and R_1" are rewritings of R' and R" using the union of the rule views $\bigcup_i R_i(D)$.

5.5 Reducing Computation cost for Incremental View Adaptation

In this section, we try to optimize some of the expressions we derived in section 5.3 for computation costs. We transform some of the expressions mentioned in section 5.3 into equivalent expressions that reduce computation costs:

1. *Removal of a positive rule R*: can be implemented as $V'(D) = ACR^{+'}(V(D))$, i.e. the new set of positive Access control rules ($ACR^{+'}$) can be computed on the view itself to generate the new view (note that constraints due to ACR^- are already taken care of in V(D)). This is computationally less expensive as it eliminates the need for the union and deep-except operations as in Equation. This can be done only if $ACR^{+'}$ can be re-written using ACR^+.

2. *Addition of a positive rule R*: in Equation 3, R' is essentially used to prune out parts of R that are already in the view. In cases where view adaptation time takes precedence, the expression $V'(D) = V(D) \cup (R(D) -^D R"(D)) where R' = R \cap^D ACR^-$ can be used instead. This eliminates the need for two operations - running R' on the database and a deep-except operation.

For adding/deleting a negative rule R, equations defined in the previous sub-section, are optimized for minimal data transfer and reduce computing cost as well.

5.6 Auxiliary Data

It is essential to maintain auxiliary data for the following reasons (a) answer end user queries on access control views, (b) speed up the view adaptation process, (c) resolve query re-writing and answering queries using views issues during adaptation.

5.6.1 Storing Path information for answering end user queries and adaptation queries

The path from the root node to the element must be stored in the view in order to allow users to query this data. Without this information, the view would practically be useless for answering user queries. For example, consider two XPath expressions, X1 = /Employee/EmployeeRecord/Name and X2 = Employee/EmployeeRecord/Age. For simplicity, let the view be defined as $X1(D) \cup X2(D)$ (only positive rules). If the path information is not stored, the view would contain:

$\langle Name\rangle \ldots \langle /Name\rangle\langle Age\langle \ldots \langle /Age\rangle$

In this case, a user query *Employee/EmployeeRecord/** cannot be answered using this view! However, if we store the following, then the query can be answered.:

$\langle Employee\rangle\langle EmployeeRecord\rangle\langle Name\rangle \ldots \langle /Name\rangle\langle /EmployeeRecord\rangle\langle /Employee\rangle$
$\langle Employee\rangle\langle EmployeeRecord\rangle\langle Age\rangle \ldots \langle /Age\rangle\langle /EmployeeRecord\rangle\langle /Employee\rangle$

5.6.2 Storing Node IDs to enable joins and query-answering using views

The original node ID of each element in the original XML document can be stored along with the element in the view. This can serve several purposes. For one, this information can enable structural join between views [26]. Secondly, it can also help in cases where we know that a query is contained within the view but cannot be answered using the view (re-writability). Consider a case where a view is defined as:

Example 23. Let ACR^+ = {Employee/EmployeeRecord/profile[/department="IST"]/name}, ACR^- = (for simplicity). Now if we want to add the negative rule R, where

$$R = Employee/EmployeeRecord[@cat. = "senior"]/profile/name$$

. Now, $R \cap^D ACR^+$ is *Employee/EmployeeRecord[@cat.="senior"]/profile/name* cannot be re-written using ACR^+. If in the view, we store the node ID of every Employee Record, then we could use this node ID to obtain the value of the category attribute for only those "EmployeeRecord" elements that are present in the view (that belong to the IST department) to determine which nodes need to be removed from the view. The node ID of every element in the path from the root to the element can be stored along with the element in the view. □

5.6.3 Storing Attributes and Predicate Information for answering predicate queries

Since storing attributes (if any) with every element in the path would result in very little overhead, this information can be stored in the view to overcome re-writability issues in cases where predicates are defined on attributes (see example above). [4] suggests storing pointers to nodes in the original document in views. Storing node IDs can be considered as a variation of this method.

Augmented rule views containing node IDs can also be stored to overcome re-writability issues. As has been illustrated in many of the examples above, rewritability problems commonly arise due to the presence of predicates in access control rules. In cases, where data needs to be removed from the view, storing the predicate information in rules could help reduce rewritability issues and elminiate the cost of accessing and querying base data. Consider this example:

Example 24. Let ACR^+ = {Employee/EmployeeRecord/profile[/dept.="IST"]/name, Employee /EmployeeRecord/profile[/dept.="CSE"]/name} ACR^- = (for simplicity). Now, to remove the positive rule R, where *R = Employee/EmployeeRecord/profile[/dept.="CSE"]/name* from this view, we would need to rewrite R using ACR^+, which is not possible, since the dept. information is missing in the view. However, if for every rule we also store the value of the element or attribute on which the predicate is defined, i.e., if we stored the dept. value with every name element in the view, it would be possible to re-write R using the view. [11] suggests a similar approach. □

6 Validation

We implemented our incremental XPath view adaptation algorithms to validate our proposal experimentally. In this section, we provide details of our empirical evaluation.

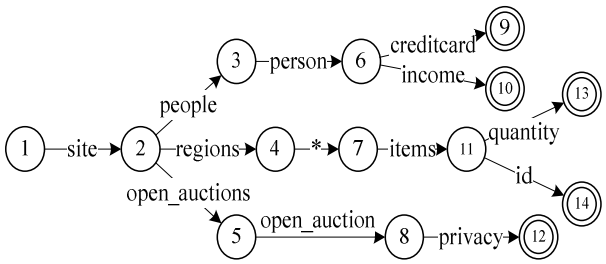


Figure 5: QFilter.

6.1 QFilter:NFA-based query filter

In the incremental view adaptation techniques described so far, in many cases, we need to obtain the intersection between a rule to be added or deleted and ACR^+ or ACR^-. To quickly compute this (deep) intersection, we use an NFA-based XML query filtering framework, QFilter [14]. We choose QFilter for our work because it is independent of the XML query processing engine and provides good performance [14] for XP{/,//,*,[]}. To construct a QFilter from a set of XPath expressions, an NFA for each expression is constructed and all such NFAs are merged to form a QFilter.

Example 25. Consider $ACR^+ = \{\}$ $ACR^- = \{$/site/people/person/creditcard,/site/people/person /income, /site/regions/*/item/quantity,/site/regions/*/item/id, /site /open_auctions/open_auction /privacy$\}$ □

The QFilter constructed from ACR^- is shown in Figure 5. To identify parts of a negative rule R= */site/people/person/** that conflict with $ACR^-(i.e. R -^D ACR^-), R$ is filtered through the QFilter in Figure 4. Filtering R would yield $\{/site/people/person/creditcard \cup /site/people/person/income\}$, i.e.$R' = R -^D ACR^-$ If predicates are involved, QFilter returns an XPath expression with conjunctions. For more on how QFilter handles various predicates, please refer to [14].

6.2 Implementation Set-Up

We have used the benchmark XML schema, XMark [2], and the XMLGen document generator to generate the database of XML Documents. We used Galax [1] XQuery processor on a machine running on Linux 2.4.18, with 1 GB RAM and Intel Xeon 2.80GHz processor. The data size used for the experiments varied from 20 to 100 MB. While recording the time for our experiments, we ignored the time taken for internal data structure initializations in the Galax Package. We record the running times for each view adaptation algorithm from the start of the evaluation of the algorithm till the view has been adapted. We did not implement XPath containment and re-writing. We used the QFilter implementation and observed the performance of Equation 2-Equation 12. The view adaptation algorithms suggested in Section 5.3 are optimal algorithms that minimize the amount of incremental data added or deleted. Therefore, we provide empirical results only for an implementation of incremental view adaptation for reduced computation costs.

6.3 Experimental Results

We collectively refer to the complete set of access control rules defining the view i.e. $ACR^+ \cup ACR^-$ as ACR.

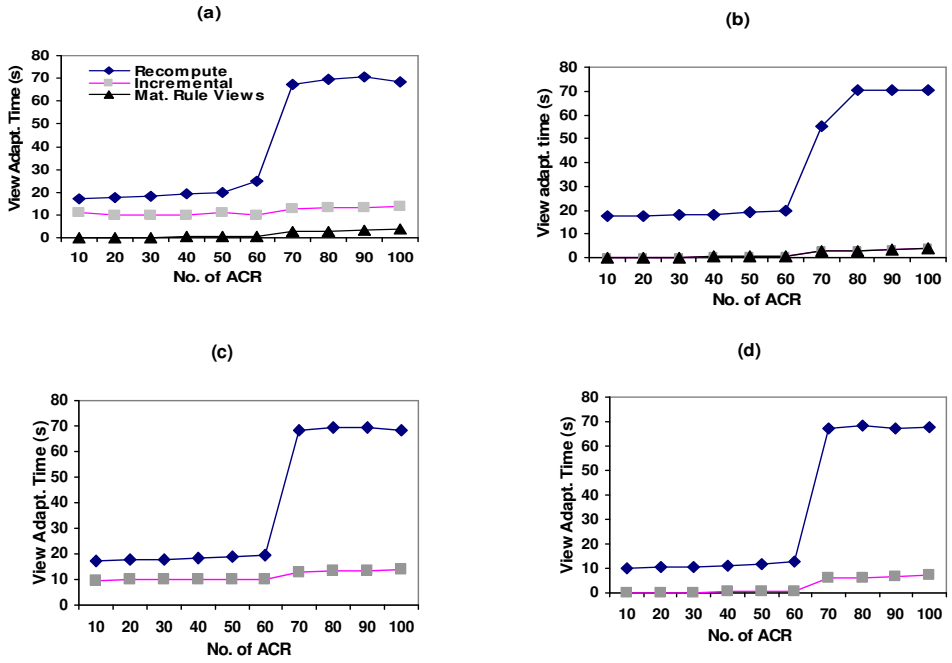


Figure 6: Effect of varying number of access control rules: (a) Remove negative rule; (b) Remove positive rule; (c) Add positive rule; and (d) Add negative rule.

I. The effect of increasing the size of ACR. We studied the effect of increasing size of ACR (10 to 100) on the performance of the different view adaptation schemes. The size of the data file used was 20 MB, and the number of positive and negative rules defining the views was equal.

As shown in Figure 6, the incremental adaptation schemes performed significantly better than the re-computation scheme in all the four cases. Re-computation time increases with increase in size of ACR^+/ACR^- because the number of XPath expressions to be evaluated during view adaptation also increases. Incremental adaptation on the other hand only needs to compute that part of the data that needs to be added/removed from the view by the new rule. As can be seen:

(a) Incremental adaptation performs best when adding a negative rule and removing a positive rule, because, for both these scenarios, in most cases, *only the view* is used for adaptation, which is typically much smaller compared to the database.

(b) While removing a negative rule, incremental adaptation using auxiliary rule views performs the best. This is because, while normal incremental adaptation uses the existing view as well as the database, the auxiliary rule-view-based incremental adaptation uses the existing view and a *small auxiliary rule view* instead to adapt the view.

In Figures 6 (a-d), the sudden peak when the size of ACR increases from 60 to 70 because the new access control rules that are added bring additional data that is disproportionately large for these additional 10 new rules (which increases the time to compute all XML nodes contained by these rules from the base data).

(c) As can be seen in Figure 6, compared to the re-computation algorithm, the incremental algorithms scale much better to this change.

II. The effect of varying the base data size. Figures 7(b) and 7(c) show that when the newly added positive rule (or (part of) the removed negative rule) needs to be evaluated on the base data; as the base data size increases, the time taken for evaluating an XPath expression on it also increases. In all four cases, the incremental methods perform better than re-computation.

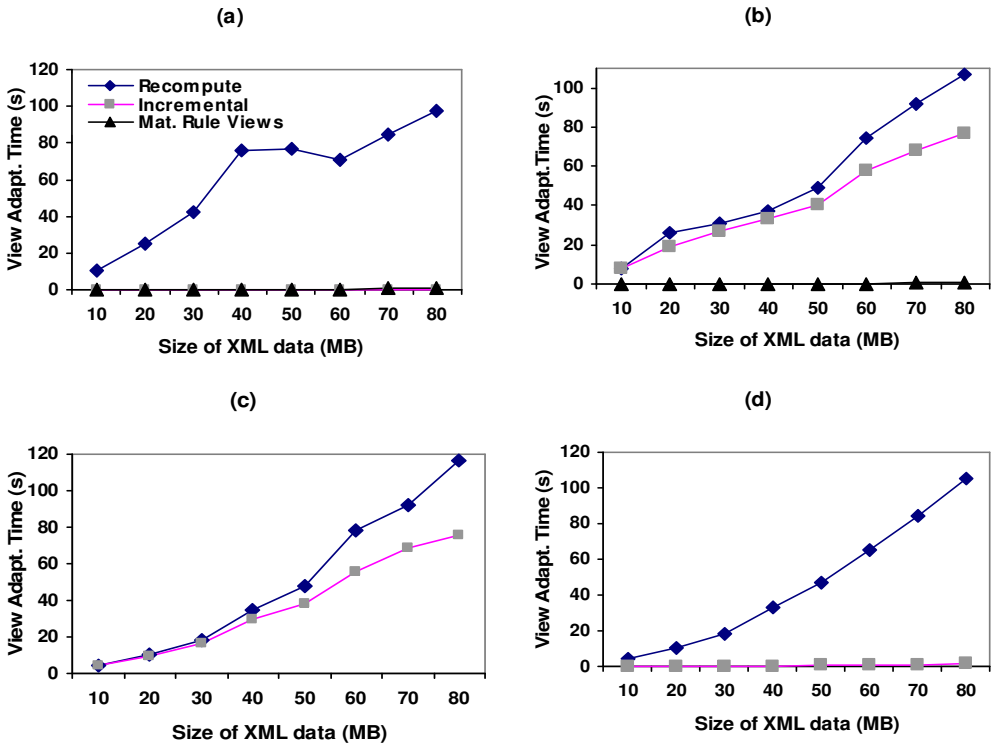


Figure 7: Scalability with increasing data size: (a) Remove positive rule; (b) Remove a negative rule; (c) Add a positive rule; and (d) Add a negative rule.

Increasing the base data size does not have any significant effect on the performance of incremental view adaptation in the cases where only the view is used for adaptation (Figures 7(a) and 7(d)). As a result: (a) In cases where only the view is used for adaptation, incremental methods perform significantly better than re-computation (sometimes by a factor of 170). (b) Where a negative rule is being removed, we see (again) that the auxiliary rule views provide a very significant performance improvement over other two methods.

III. The effect of increasing the view size. We observed the effect in the two representative cases, where a negative rule is added (data is removed from the view) and a positive rule is added (data is added to the view). In both cases, the incremental adaptation method scaled well to increasing view sizes whereas the time taken to re-compute the view increased with increasing size of the view (Figure 8). This is because, as the rules in ACR bring in more data into the view, more computations are needed on the base data while recomputing the view. In the incremental method, when the view size increases by small fractions, it makes very little difference to view adaptation time.

6.3.1 Summary of Observations

From the observations above, it is clear that incremental view adaptation outperforms view re-computation, in the event of all the four different kinds of updates: adding a positive rule, deleting a positive rule, adding a negative rule, and deleting a negative rule. Our incremental view adaptation approach also scales well to increasing number of access control rules, increasing sizes of base data and security views, as demonstrated by the experiments. Rule views have led to significant

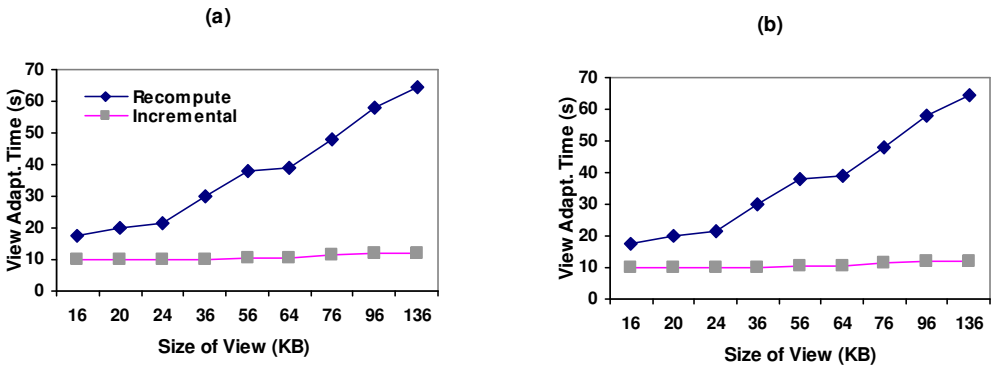


Figure 8: Effect of increasing view size re-computation and incremental approaches: (a) Add negative rule; and (b) Add positive rule.

performance improvements for deleting positive and negative rules, wherever the rule views can be used without any rewritability or answerability issues.

7 Conclusions and Future Work

Efficient view adaptation for reducing view-downtime is important for the use of access-control views in a wide range of applications. In this work, we have proposed and implemented a novel and efficient approach to incrementally adapt XPath access-control views expressed defined using XP{/,//,*,[]}. We also suggest techniques for more efficient view adaptation using auxiliary data, such as rule views. We show empirically that incremental view adaptation performs better than view re-computation, and that auxiliary rule views lead to significant performance improvement when positive rules are added and negative rules removed.

The solution of the XPath view adaptation problem is dependent on efficient solutions to the XPath query rewriting using multiple views problem. Therefore, solving the problem of rewriting XPath queries using XPath views especially in the presence of negative rules and plugging it into our infrastructure, will improve the efficiency and completeness of our view adaptation algorithms. The Galax XQuery processor is limited in the size of the data that it can process. A more robust XQuery processing engine will help us verify the scalability of our algorithms for even larger sizes of XML documents. Given the trends observed, the potential for gains over the recomputation-based methods is even more for larger documents. The investigation of incremental XPath view adaptation under different models of XML access control can also be undertaken.

References

[1] Galax xquery processor, available at http://db.bell-labs.com/galax.

[2] Xmark benchmark schema, available at http://www.xml-benchmark.org/.

[3] Xpath, available at http://www.w3.org/tr/xpath20.

[4] A. Balmin, F. Ozcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB Conference*, 2004.

[5] L. Bouganim, F. Ngoc, and P. Pucheral. Client-based access control management for xml documents. Technical report, INRIA, June 2004.

[6] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):59–75, 2000.

[7] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure xml querying with security views. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 587–598, New York, NY, USA, 2004. ACM Press.

[8] I. Fundulaki and M. Marx. Specifying access control policies for xml documents with xpath. In *The ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 61–69. ACM Press, 2004.

[9] G. Guerrini, M. Mesiti, and D. Rossi. Impact of xml schema evolution on valid documents. In *WIDM '05: Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 39–44, New York, NY, USA, 2005. ACM Press.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[11] A. Gupta, I. S. Mumick, and K. A. Ross. Adapting materialized views after redefinitions. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 211–222. ACM Press, 1995.

[12] G. Kuper, F. Massacci, and N. Rassadko. Generalized xml security views. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 77–84, New York, NY, USA, 2005. ACM Press.

[13] C.-H. Lim, S. Park, and S. H. Son. Access control of xml documents considering update operations. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 49–59, New York, NY, USA, 2003. ACM Press.

[14] B. Luo, D. Lee, W.-C. Lee, and P. Liu. Qfilter: fine-grained run-time xml access control via nfa-based query rewriting. In *The thirteenth ACM international conference on Information and knowledge management (CIKM)*, pages 543–552, New York, NY, USA, 2004. ACM Press.

[15] B. Luo, D. Lee, W.-C. Lee, and P. Liu. Deep set operators for xquery. In *XIME-P*, 2005.

[16] B. Mandhani and D. Suciu. Query caching and view selection for xml databases. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 469–480. VLDB Endowment, 2005.

[17] G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.

[18] M. Murata, A. Tozawa, M. Kudo, and S. Hada. Xml access control using static analysis. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 73–84, New York, NY, USA, 2003. ACM Press.

[19] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *International Conference on Database Theory*, 2003.

[20] R. J. Peters and M. T. Ozsu. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Trans. Database Syst.*, 22(1):75–114, 1997.

[21] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path-expression views. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 443–454, New York, NY, USA, 2005. ACM Press.

[22] T. Schwentick. Xpath query containment. *SIGMOD Rec.*, 33(1):101–109, 2004.

[23] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *The VLDB Journal*, pages 75–86, 1996.

[24] A. Stoica and C. Farkas. Secure xml views. In *DBSec*, pages 133–146, 2002.

[25] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.

[26] Y. Wu and H. Jagadish. Structural join order selection for xml query optimization. In *ICDE Conf., Bangalore, India*, Mar. 2003.

[27] W. Xu and Z. M. Ozsoyoglu. Rewriting xpath queries using materialized views. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 121–132. VLDB Endowment, 2005.

[28] T. Yu, D. Srivastava, L. Lakshmanan, and H. Jagadish. Compressed accessibility map: Efficient access control for xml. In *VLDB Conference*, 2002.