Access-Condition-Table-Driven Access Control for XML Databases

Naizhen Qi (Naishin Seki) and Michiharu Kudo

IBM Research, Tokyo Research Laboratory,
1623-14, Shimo-tsuruma, Yamato-shi,
Kanagawa 242-8502, Japan
{naishin,kudo}@jp.ibm.com

Abstract. Access control represented by XPath expressions allows for access restrictions on elements, attributes, and text nodes according to their locations and values in an XML document. Many XML database applications call for such node-level access control on concerned nodes at any depth. To perform such node-level access control, current approaches create heavy loads on XML database applications since these approaches incur massive costs either at runtime or for data optimization. In order to solve these problems, we introduce an access condition table (ACT), a table equivalent to an access control policy, where Boolean access conditions for accessibility checks are stored. The ACT is generated as a means of shifting the extra runtime computations to a pre-processing step. Experimental results show that the proposed ACT can handle accesses to arbitrary paths at a nearly constant speed.

1 Introduction

The Extensible Markup Language (XML [6]) is widely used for data presentation, integration, and management because of its rich data structure. Since data with different security levels may be intermingled in a single XML document, such as business transactions and medical records, access control is required on both the element- and attribute-level to ensure that sensitive data will be accessible only to the authorized users. In most of the current research, such node-level access control is specified with XPath [10] to identify the sensitive portion.

Since the utilization of global elements and global attributes may scatter data throughout an XML document, the node-level access control specification is required to cover affected nodes at any depth. For instance, in XML-formatted medical records, doctors' comments may appear anywhere as the occasion demands. Therefore, to hide comments from unauthorized users, it is required to select all of the comments regardless of their locations.

Many approaches (e.g. [3, 11, 21, 24]) fulfill access control on arbitrary nodes with the `descendant-or-self` axis of XPath (`//`) and the propagation mechanism. Since `//` selects a specific descendant in the subtree and propagation mechanism brings access down to the subtree, both of them require node traversal of the entire subtree. Especially for deeply layered XML documents, the enforcement of // and propagation imposes heavy computational costs. Therefore, some of the approaches(e.g. [3, 11, 17, 21]) trade efficiency off against *expressiveness*[1].

Ideas to efficiently perform an arbitrary XML node selection are also proposed by [16, 19, 20, 22, 28], in which the result of access control is cached for each affected node. Since such optimization is generally on a per-document basis, the databases with few document updates and document insertions may profit from these approaches. However, in real XML database applications, new XML documents may frequently be inserted into the database. For instance, as E. Cecchet[7] shows, the interaction of EJB-based business applications may happen 10 to 200 times per second. If each interaction is recorded into an XML document, document-based optimization will be performed frequently and potentially lead to unacceptable performance. Therefore, we think efficiency should be achieved independent of the data in the XML documents. We call this requirement *document-independency.*

In this paper, we introduce a novel access-condition-table-driven mechanism which achieves high *expressiveness* with good *document-independent* efficiency. The access condition table (ACT) is a table storing the paths and the two types of access conditions, which are Boolean expressions. Given a path, the ACT can provide a proper access condition according to the path location. Then the provided access condition is evaluated to decide the accessibility. As far as we know, our ACT-driven approach is the first one capable of processing the access to an arbitrary path at a nearly constant speed irrespective of the XML document. The ACT is free from re-computation as long as the access control policy is not updated. In addition, the ACT can provide applicable access control for various query languages including XQuery, SQL XML, and XPath.

1.1 Related Work

Many approaches to enforcing XML access control have been proposed. Some of them support full XPath expressions but perform with naive implementations which may incur massive computations. For instance, when the access control policy is large or the XML document is deep layered, Kudo et al. [21] and Gabillon [17] may suffer from high runtime costs, since they create the projection of the access control policy on a DOM [18] tree and then evaluate the accessibility at each node. The mechanisms proposed in [2, 3, 11, 12] also encounter the same problem at runtime since the node-level access control on a DOM-based view can be expensive especially for a large XML document.

Substantial performance costs can be a fatal weakness for a secure XML database, access control with a pre-processing optimization to improve runtime efficiency has been explored in many research projects [1, 8, 14, 24, 28]. For example, the static access optimization algorithm in Murata et al. [24] minimizes the runtime checks by determining the accessibility of nodes in a query with pre-computed automata which is *document-* and *schema-independent.* However,

[1] We use *expressiveness* to describe the support to //, predicates, and propagation.

more than access control policy, the queries, which can be complicatedly expressed in XQuery, are required by the system. Our approach is complement to their method that we supports node-level access control for more query languages but also with more limitations on policy specifications. Yu et al. [28] enforces access control by obtaining accessibility from an accessibility map which is optimized on the basis of the XML document and therefore document updates and insertions may trigger re-computations. In addition, there are XPath-based documents filtering systems [1, 8, 14] satisfying the requirements of *expressiveness* and *document-independency* through pre-computed special data structures. However, these approaches do not support to eliminate a denial node from a grant tree or sub-tree; therefore, they may be unable to afford appropriate access control for some XML database applications.

Optimization is also done in a number of research efforts on XML query languages (e.g., XPath and XQuery [5]). The methods include query optimization based on (i) the tree pattern of queries [9, 13, 23, 25–27], (ii) XML data and XML schema [16, 19, 20, 22] and (iii) the consistency between integrity constraints and schemas [15]. However, these data selection mechanisms are not applicable to other query languages and primitive APIs such as DOM.

Outline. The rest of this paper is organized as follows. After reviewing some preliminaries in Section 2, we introduce the features of the ACT in Section 3 and the construction of the ACT in Section 4. Experimental results are reported in Section 5 and the conclusions and future work are summarized in Section 6.

2 Preliminaries

XPath. An XPath expression can select nodes based on the document structure and the data values in an XML document. A structure-based selection relies on the structural relationships, which is expressed by `/` and `//`. For example, `/record//name` selects all `name` in the subtree of `record`. In addition, value-based selection is done by attaching a value-based condition on a specific node: if the condition is satisfied, the node is selected. For instance, the XPath expression `/disclosure[@status='published']` selects `disclosure` whose `status` attribute equals 'published'. In this XPath expression, `@status='published'` is a value-based condition which is called a *predicate*.

Access Condition Policy. Various access control policy models have been proposed, but we use the one proposed by Murata et al. [24] in which an access control policy contains a set of 3-tuples rules with the syntax: *(Subject, Permission Action, Object)* as shown in Table 1. The subject has a prefix indicating the type such as *uid* and *group*. '+' stands for a grant rule while '-' for a denial one. The action value can be either *read*, *update*, *create*, or *delete*. Due to the lack of space, we focus on the read action in this paper though the others can be implemented with the same mechanisms. The rule with `+R` or `-R` is propagation permitted that the access can be propagated downward on the entire subtree, while `+r` is propagation denied. As an example, `(uid:Seki, +r, /a)` specifies

Table 1. Access Control Rule Syntax

Field	Description
Subject	A human user or a user process with a *uid* or *role* prefix
Permission	Grant access (`+`) or denial access(`-`)
Action	`r`: without propagation; `R`: with propagation
Object	An XPath expression selects affected nodes

user Seki's access to `/a` is allowed but to `/a/b` is implicit specified since *grant* is not propagated down to the descending paths of `/a` owing to `r`. Moreover, according to the *denial downward consistency* in [24] that the descendants of an inaccessible node are either inaccessible, there is an accessibility dependency between the ancestors and the descendants. Therefore, it is obvious that `-r` is equivalent to `-R`; and thus, we specify denial rules only with `-R` in this paper. In addition, in order to maximize the security of the data items, we (i) resolve access conflicts with the *denial-takes-precedence* [4], and (ii) apply the default denial permission on the paths if no explicit access control is specified.

3 Features of the Access Condition Table (ACT)

Before entering the details, we list up some requirements considered to be important to an access control system for XML databases.

– *Document-independency* and *schema-independency* should be satisfied.
– Document updates and document insertions should not trigger any re-computation.
– Access control should be applicable to query languages including XQuery, SQL XML, and XPath.

However, we consider these requirements in a system where the frequency of the policy updating is far lower than the frequencies of the actions performed.

3.1 Structure of the ACT

The ACT is directly generated from an access control policy. Different from the access control policy, the ACT separates the affected paths and the related conditions from the object. We call the affected paths the *target paths*. The values of each target path are two types of access conditions: *access condition* and *subtree access condition*. The access conditions are for the accessibility checks on the target paths, and the subtree access conditions are for the descending paths. Featured with two types of access conditions, the ACT enables access control on a path of any length. The key idea of this structure is based on the following two observations.

- Each `//` in an XPath expression can be regarded as a divider. The part on the right of `//` is the condition that should be satisfied by the descending paths in the subtree of the part on the left of `//`. For instance, granting access on `/a//c` specifies *if the node is* `c` *in the subtree of* `a`*, the access to the node is granted.* As a result, the accesses to `/a/c`, `/a/b/c`, and `/a/b/c/d` are granted since they are `c`s in the subtree of `a`. Therefore, an subtree access condition can be prepared at `/a`, which is responsible for the access to any path in the subtree of `/a`.
- Propagation is the mechanism controlling the access of the entire subtree. For instance, granting access to `/a` with propagation specifies *the accesses to* `a` *itself and its descendants are granted.* However, granting access to `/a` without propagation specifies *the access to* `a` *is granted but to the descendant is denied.* Since the access conditions to `/a` and to its subtree can be separately specified, we can prepare the access conditions separately as well.

As a result, we generate two types of access conditions: one is the access condition for a path itself like `a` in the above two examples, and the other is the access condition for the entire subtree such as the subtree of `a` in the examples. Example 1 shows a simple access control policy and its equivalent ACT.

Example 1. Given an access control policy P consisting of rules R1, R2, R3, and R4 as follows, the equivalent ACT is as Table 2 shows.

R1: `(uid:Seki, +r, /a)`, R2: `(uid:Seki, +R, /a/b)`
R3: `(uid:Seki, +r, /a/c[g>1])`, R4: `(uid:Seki, -R, /a/b//e)`

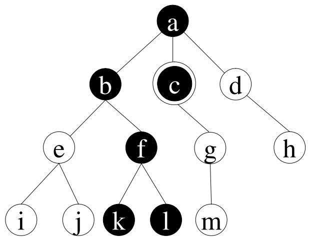
Table 2. Structure of the ACT

Target Path	Access Condition	Subtree Access Condition
/a	true	false
/a/b	true	not (ancestor-or-self::e)
/a/c	g>1	false

We use an accessibility marked tree, the abstract representation of an XML document, to reflect the access control result of P with accessible and inaccessible nodes individually distinguished. It also reflects the ACT shown in Table 2 since the ACT leads to the same accessibility marked tree. In Fig. 1, grant access is propagated from `b` to `e`, `i`, `j`, `f`, `k` and `l` owing to R2 while denial access is propagated from `e` to `i` and `j` owing to R4. Therefore, access conflicts occur on `e`, `i` and `j`. Finally, `e`, `i` and `j` are inaccessible based on the *denial-takes-precedence* principle. On the other hand, the nodes without access specification, `d`, `g`, `h`, and `m` are ultimately inaccessible owing to the default denial permission. In addition, the accessibility of `c` is decided by the value of `g` : if g>1 then accessible, otherwise inaccessible.

3.2 Pre-computation

The ACT generation can be considered as a pre-computation stage in which some extra computations are shifted from the runtime. Generally, the extra



- Black nodes: accessible nodes
- Circled black nodes: value-based accessible nodes
- White nodes: inaccessible nodes

Fig. 1. The accessibility marked tree of P

computations are caused by the processes of propagation and conflict resolution. Conflict arises when a node is granted access and denied access at the same time. For instance, (i) `(Sbj, +R, /a)`, and (ii) `(Sbj, -R, /a/b)` cause conflict on `b` since `b` is granted owing to the propagation from `a` and at the same time denied because of the denial specification. Moreover, if multiple `b`s are subordinate to the same `a`, the propagation and the conflict identically occur on each `b`. Obviously, such repeated processes should be optimized to conserve resources. The ACT-driven mechanism moves the processes of propagation and conflict resolution into the ACT generation to achieve high runtime performance.

3.3 Access Conditions and Subtree Access Conditions

Access conditions and subtree access conditions are Boolean expressions evaluated for accessibility. They are *true* or *false* when the objects do not contain either a `//` or a predicate. Only for the objects containing a `//` or a predicate, is the XPath expression representing the structure-based or value-based condition involved in the condition expressions.

For instance, for R2 in Example 1, the access condition on `/a/b` is *true* and the subtree access condition of `/a/b` is *true* too since 'true' is permitted to spread down to the entire subtree of `/a/b` owing to 'R'.

3.4 ACT-Driven Access Control Enforcement

Since access conditions are path-specific, each node in an XML document can obtain a proper access condition through its path expression on the basis of its location: one of the target paths or the descending path in someone's subtree.

Access control is enforced via an ACT handler that, upon receiving a user request, returns accessible data to the user. The user request contains the user information and the access target, which can be either a path or an XML document. For a path, the accessibility is decided by evaluating the access condition according to its location: if the path is a target path in the ACT, the access condition is evaluated; otherwise, the closest ancestor is searched for and its subtree access condition is evaluated. For an XML document, the ACT handler traverses the document to check the accessibility of each path: if the path is found to be accessible, it is added to the user view. In order to improve the performance in processing XML documents, the ACT handler adopts a *check-skip* mechanism to skip accessibility checks on the nodes with an inaccessible ancestor. According to the denial downward consistency, the nodes with an inaccessible ancestor are definitely inaccessible, so the checks on them can be skipped.

Table 3. Accessibility checks for Seki

Requested Path	Target	Access Condition	Accessibility Result
/a	/a	true	Accessible.
/a/c	/a/c	g>1	Accessible when g is bigger than 1, otherwise inaccessible.
/a/b/h	/a	false	Inaccessible.
/a/b/e/i	/a/b	not(ancestor-or-self::e)	Inaccessible.

Using the ACT in Table 2, we show four examples in Table 3. For instance, Seki requests to access `a`, the ACT handler finds `/a` in the ACT and therefore the access condition, *true*, is evaluated. Consequently, the access to `a` is allowed. When Seki accesses `i` whose path is `/a/b/e/i`, the ACT handler finds it absent from the ACT. However, the closest ancestor target path, `/a/b`, is found and the subtree access condition, `not(ancestor-or-self::e)`, is provided. Since `not(ancestor-or-self::e)` is evaluated *false* for `/a/b/e/i`, the access is denied.

3.5 Limitations on the ACT

In order to keep the ACT simple, the following four limitations on XPath expressions are specified.

– `//` is limited to appear only once. (i.e. `/a//c//e` is not allowed.)
– A wildcard `*` should always be accompanied with a `//` in the form of `//*`. (i.e. `/a//b/*/@d` is not allowed.)
– `//` and `*` never appear in predicates. (i.e. `/a[*>1]/b` is not allowed.)
– Only one node can be identified after `//`. (i.e. `/a//c/d` is not allowed.)

The limitations restrict the expressiveness of the access control policies. Nonetheless, the specification of a node at any depth, e.g. `//*[@level='classified']` and `//comment`, are supported. In addition, some of these limitations can be resolved by XPath extension functions.

4 Construction of the ACT

The ACT is generated from the access control policy by fowllowing 5 steps:

– Step 1: Generate the target paths
– Step 2: Generate the local access conditions

– Step 3: Generate the local subtree access conditions
– Step 4: Perform propagation
– Step 5: Perform conflict resolution and expression combination

Note that the access conditions and the subtree access conditions generated in Step 2 and Step 3 are local since propagation has not yet occurred. In Step 4, local subtree access control with propagation permission is spread down to the descending target paths in the ACT. In Step 5, access conditions and subtree access conditions residing on each target path are individually combined. At the same time, conflicts are also resolved with the conflict resolution mechanism.

4.1 Generating the Target Paths

The target path is generated from the object by removing the additional access conditions. If the object contains `//`, the prefix to `//` is the target path. If the object contains predicates, then the path after removing predicates is the target path. However, if the object contains both `//` and predicates, the path after removing the predicates in the prefix is the target path.

For instance, since R3's object in Sect. 3.1 contains a predicate, the target path is `/a/c` after removing the predicate from `/a/c[g>1]`. Moreover, for R4's object `/a/b//e`, the prefix to `//`, which is `/a/b`, is the target path.

4.2 Generating the Local Access Conditions

Local access conditions on each target path are generated from the action and the object. If the object contains predicates or `//`, the additional access conditions should be included.

When the object does not contain either a `//` or a predicate, the action decides the local access condition: `true` for `+` and `false` for `-`. When the object contains a predicate, the predicate relative to the target path is generated as the local access condition. If the object contains multiple predicates, the local access condition is generated by connecting the predicates relative to the target path with 'and'. However, for the object containing `//`, the local access condition is not available since it specifies access for the descendants rather than for the target path itself.

In the example in Sect. 3.1, the access conditions of `/a` and `/a/b` are 'true' since the actions of R1 and R2 are both `r`. However, the access condition of `/a/c` is the predicate `g>1` which is the value-based condition imposed on `c`.

4.3 Generating the Local Subtree Access Conditions

The intuition for subtree access conditions is to combine both information on (i) the accessibility dependency from the root to the target path, and (ii) the ancestor-descendant relationship from the target path to an arbitrary descending path. Only when both (i) and (ii) are satisfied can the nodes in the subtree be accessible. We use `ancestor-or-self` axis and `descendant-or-self` axis to describe the structural relationship between ancestors and descendants. Algorithm 1 shows the subtree access condition generation when given the action A and the object T. In addition, in this section we address the case that predicates and // do not appear in the same T.

Algorithm 1. Subtree access condition generation.
If T contains //, then T can be represented as $/e_1//e_n$.

```
if (T contains '//') then
    if (A is '+R') then
        subtree access condition ← ancestor-or-self::$e_n$ or descendant-or-self::$e_n$
    else if (A is '+r') then
        subtree access condition ← descendant-or-self::$e_n$
    else if (A is '-R') then
        subtree access condition ← not(ancestor-or-self::$e_n$)
    end if
else
    if (A is '+R') then
        subtree access condition ← true
    else if (A is '-R') then
        subtree access condition ← false
    else
        return        // local subtree access condition is N/A
    end if
end if
```

According to the above algorithm, R4: `(uid:Seki, -R, /a/b//e)` in Sect. 3.1 leads to a local subtree access condition `not(ancestor-or-self::e)` stating that *if the requested node is* `e` *below* `/a/b` *or has such an ancestor* `e`*, it is inaccessible.* Therefore, subtree access conditions enable access control on a path of any depth.

4.4 Performing Propagation

The propagation mechanism brings the local subtree access conditions to a wide descendant area. In our approach, propagation is performed by adding the local subtree access condition to both the access conditions and the subtree access conditions of the descending target paths. To reduce the runtime computation costs, at this step the propagated subtree access conditions are evaluated as to whether or not they are satisfied by the current descending target path. If so, the access conditions are rewritten as `true` or `false` based on the propagated access.

4.5 Performing Expression Combination and Conflict Resolution

Since the propagation mechanism transmits ancestral access to the descending target paths, multiple access conditions and subtree access conditions may apply to each target path. Rather than simply connect all of the conditions with the Boolean operators *and* or *or*, we logically combine the access conditions into a Boolean expression. In addition, when access conflict happens, it is resolved with the *denial-takes-precedence* policy.

In Example 1 of Sect. 3.1, after performing the four steps, two different subtree access conditions are imposed on `/a/b`. One is *true* specified by R2, the other is `ancestor-or-self::e` specified by R4. According to the expression combination, the subtree access condition is `true and not(ancestor-or-self::e)`, which can be rewritten to `not(ancestor-or-self::e)`.

4.6 Enhanced Subtree Access Conditions

In this section, we introduce a mechanism to handle the case where a predicate and the permitted propagation appear at the same time, which is not handled by the algorithm in Sect. 4.3.

Suppose there is a P' containing a propagation permitted third rule that R3':`(uid:Seki, +R, /a/c[g>1])`. Owing to propagation, R3' implies the access to `g` and `m` should be decided by evaluating the predicate relative to `c`. Therefore, it is required to convert the predicate relative to `c` into the ones relative to `c` from the accessed descendants, such as `..[g>1]` from `g` and `../..[g>1]` from `m`. It is obvious that the descendant at a different depth leads to a different relative predicate, and it is necessary to describe this non-deterministic relative relationship in a static way.

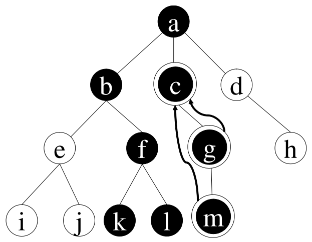

a
b
c
d
e
f
g
h
i
j
k
l
m
- Black nodes: accessible nodes
- Circled black nodes: value-based accessible nodes
- White nodes: inaccessible nodes

Fig. 2. The accessibility marked tree of P'

However, according to the denial downward consistency, we can say when `c` is accessible its descendants are all accessible if no other denial access is imposed. Therefore, the non-deterministic relative relationship can be regarded as a value-based accessibility dependency on `c` in this example. Moreover, regardless of the locations, all descendants below `c` can share the same value-based accessibility dependency. A system variable `ref(target path)` is used to represent this dependency: if the value-based `target path` is evaluated accessible, then the descendant is accessible.

Fig. 2 shows the accessibility marked tree for P' in which arrows starting with `g` and `m` ending with ancestor `c` represent the accessibility dependency. The local subtree access condition of R3' can be simply notated as `ref(/a/c)`. Furthermore, the accessibility reference is also required when the object contains `//` with a predicate before, like `(uid:Seki, +r, /a/b[f>1]//e)`. Using the same mechanism, the local subtree access condition is therefore `ref(/a/b) and descendant-or-self::e`.

5 Experiments

To validate the efficiency of the ACT, we ran a variety of experiments to see how our techniques perform. We measured *view generation time*, the time cost to generate the view of an XML document based on the policy, to compare the performance between our ACT-driven enforcement and a simple implementation without pre-computing access conditions. The purposes of the experiments are to see (i) how the policy size works on the view generation time, (ii) how the ACT performs comparing with the simple implementation, and (iii) how ACT performs when `//` appears.

5.1 Experimental Data

Experimental Environment. All experiments were run on a 3.06 GHz Xeon processor with 2.50 GB RAM. In this paper, we report on an XML document obtained from http://www.w3c.org, which was generated on the basis of xmlspec-v20.dtd, with a data size of almost 200 KB, 768 types of paths, and 4,727 nodes when represented as an XML tree.

For simplicity in the experiments, we created policies for a single user. Nonetheless, we can use multiple ACTs for multi-user scenario. Moreover, we ran comparison experiments for structure-based access control, but did not for value-based since the performance partly depends on the efficiency of the predicate evaluation. However, we can infer the performance from the results of our experiments.

Access Control Policy Patterns: *pattern-a* **and** *pattern-b***.** The access control policies were not randomly generated. In particular, we guaranteed that no duplicated access was imposed on the inaccessible descendants, since 'denial' is already contained in the inaccessible ancestor according to the denial downward consistency.

We generated the policy with accessible nodes as the object and `+r` as the action, and named it *pattern-a* access control policy. We also generated a corresponding reverse version, *pattern-b*, that contained denial rules on the most ancestral node of each denial subtree. In addition, *pattern-b* also contains a grant rule specified on the root node with `+R`. Conceptually, *pattern-a* selects the accessible nodes from the inaccessible region while *pattern-b* selects the inaccessible nodes from an accessible region. Fig. 3 shows an example for the accessibility marked tree *Tree*, with the corresponding access control policies in *pattern-a* and *pattern-b* as shown in the figure.

Access control policy in *pattern-a*:
(uid:Seki, +r, /a)
(uid:Seki, +r, /a/b)
(uid:Seki, +r, /a/c)

Access control policy in *pattern-b*:
(uid:Seki, +R, /a)
(uid:Seki, -r, /a/d)

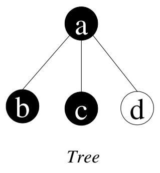


Fig. 3. Access control policies for *Tree*

Access Control Policies. We generated 11 paired sets of the access control policies in *pattern-a* and *pattern-b* with an access ratio varying from 0.03 to 0.95, where the access ratio is the fraction of the accessible nodes in the XML structure tree. The XML structure tree is the one built by the set of the paths appearing in the XML document. *Policy size* is the number of rules in an access control policy. The policy sizes of these access control policies are different, as Fig. 4 shows. Since in *pattern-a* the greater the access ratio is, the more nodes are explicitly

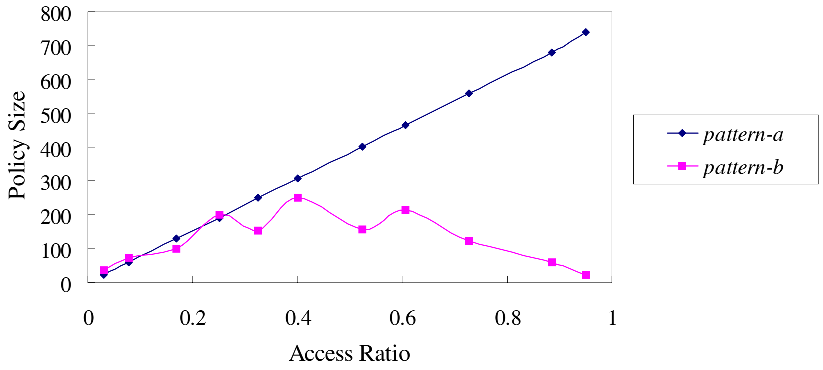


Fig. 4. Policy Size versus Access Ratio in *pattern-a* and *pattern-b*

specified, the size of the policy increases proportionally. On the other hand, in *pattern-b* rather than listing up each inaccessible node, only those inaccessible ones with an accessible parent are specified denial access. Consequently, the size of *pattern-b* is much smaller than *pattern-a* and without a regular form.

For the access control policies in both patterns, the size of the corresponding ACT is almost the same for each policy size, and thus Fig. 4 can be regarded as the chart for the ACT size.

5.2 Experimental Results

We show the advantages of the ACT through performance comparisons between the ACT-driven and a simple enforcement, which has a hash table structure storing the *Subject*, *Object*, and *Permission Action* introduced in Table 1. In the simple enforcement, each requested path is checked by evaluating the access control policies with *Objects* matched by the requested path. The check-skip mechanism introduced in Sect. 3.4 is also used by the simple enforcement.

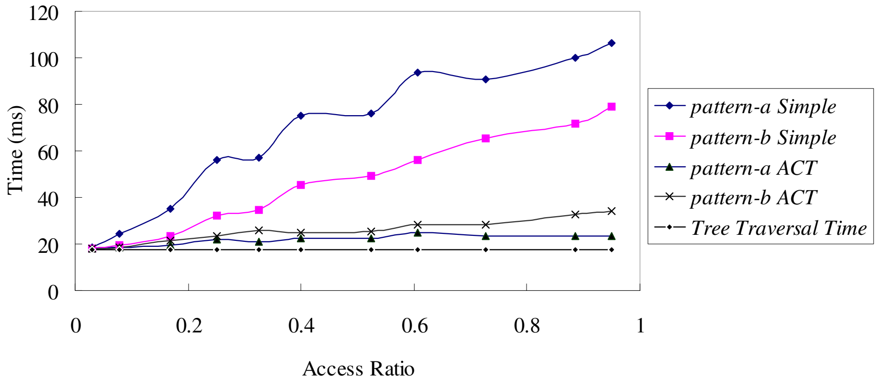


Fig. 5. View Generation Time versus Access Ratio

Relationship Between the Policy Size and the View Generation Time. Fig. 5 respectively shows the view generation times of the simple enforcement (Simple) and the ACT-driven enforcement (ACT) for both *pattern-a* and *pattern-b*, when the access ratio is varied from 0.03 to 0.95. In addition, in the figure we also show the *tree traversal time*, which is the time required by traversing the entire DOM structure. Time costs on accessibility checks for both Simple and ACT are the results by subtracting the *tree traversal times* from the *view generation times*.

From the figure, we can see that the chart of *pattern-b* is very similar to that of *pattern-a* in the trend: the larger the access ratio, the more time is required by Simple but ACT is almost constant. In particular, for *pattern-a*, when the access ratio is higher than 60%, ACT performs more than 4 times faster than Simple and when the access ratio is 95%, it reaches 5 times; while for *pattern-b*, Simple takes more than 2 times as long compared to ACT, while ACT shows quite slow growth. Furthermore, for the time costs on accessibility checks, which ignores the time of a DOM-based traversal, ACT performs much better than Simple that ACT is 11 $\sim$ 15 times faster than Simple for *pattern-a* when the access ratio is higher than 60%, and approximately 4 times faster for *pattern-b*.

In our experiments, a DOM-based traversal is used to retrieve paths from the XML document. However, other means of path retrievals may lead to less view generation times. Nonetheless, time costs on accessibility checks will not change for both ACT and Simple, which we have shown above.

How ACT Performs for //. To illustrate the advantages of the ACT-driven access control specified with //, we also ran experiments on the third set of access control policy data. We converted 20%, 25%, and 33% percentage of access control rules in *pattern-b* into rules with objects containing a //. The results are quite similar to each other, and therefore we show only the 20% case in Fig. 6.

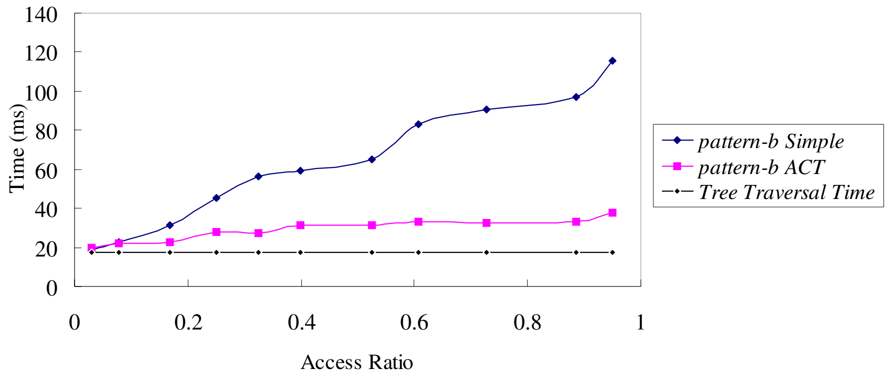


Fig. 6. Simple versus ACT on //

From the figure, we can see that ACT takes less time on view generation and the time increases slightly as the access ratio grows. In particular, ACT is faster than Simple by more than 3 times when the access ratio is 60% and by 5 times when 95%. Moreover, for the time costs on accessibility checks, ACT performs 6 times faster than Simple at most.

Our comparison experiment results show the ACT can perform almost constantly no matter with the policy size. And it is clear that the ACT-driven access control enforcement also shows advantages for //. The efficiency of the ACT is achieved with minimum computation costs by eliminating the processing of propagation and conflict resolution from the runtime.

6 Conclusion and Future Work

In this paper, we proposed an ACT-driven access control mechanism to provide efficient node-level access control. Using document- and schema-independent optimization features, the ACT is capable of handling various XML database applications without re-computation triggered by data and schema updates. In addition, our approach shifts most of the extra computations to the pre-processing stage to save on the runtime costs, which yields efficient performance that it can provide structure-based access control nearly in constant time.

On the other hand, some further research work is required. We have placed four limitations on access control policy, some of them can be resolved by XPath extension functions. Another important effort is to integrate the ACT approach with the static analysis method [24] in a well-balanced manner to improve expressiveness and performance.

Acknowledgments

We would like to thank Makoto Murata for discussions and comments on this work.

References

1. M. Altinel and M. Franklin: Efficient filtering of XML documents for selective dissemination of information. VLDB (2000) pp.53-64.
2. E. Bertino, S. Castano, E. Ferrari, and M. Mesiti: Controlled access and dissemination of XML documents. ACM WIDM (1999) pp.22-27.
3. E. Bertino and E. Ferrari: Secure and selective dissemination of XML documents. ACM TISSEC (2002) pp.290-331.
4. E. Bertino, P. Samarati, and S. Jajodia: An extended authorization model for relational database. IEEE trans. on Knowledge and Data Engineering (1997).
5. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon: XQuery 1.0: An XML query language, W3C Working Draft 12 November 2003. Available at http://www.w3.org/TR/xquery/.
6. T. Bray, J. Paoli, and C. M. Sperberg-McQueen: Extensible Markup Language (XML) 1.0. W3C Recommendation. Available at http://www.w3g.org/TR/REC-xml (Feb. 1998).
7. E. Cecchet, J. Marguerite, and W. Zwaenepoel: Performance and scalability of EJB applications. OOPSLA (2002) pp.246-261.
8. C. -Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi: Efficient filtering of XML documents with XPath expressions. ICDE (2002) pp.235-244.
9. S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, and D. Srivastava: Optimizing the secure evaluation of twig queries. VLDB (2000) pp.490-501.
10. J. Clarkand S. DeRose: XML Path Language (XPath) version 1.0. W3C Recommendation. Available at http://www.w3g.org/TR/xpath (1999).
11. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati: Design and Implementation of an Access Control Processor for XML documents. WWW9 (2000).
12. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati: A Fine-Grained Access Control System for XML Documents. ACM TISSEC (2002) pp.169-202.
13. A. Deutsch and V. Tannen: Containment of regular path expressions under integrity constraints. KRDB (2001).
14. Y. Diao, P. Fischer, M. Franklin, and R. To.: YFilter: Efficient and scalable filtering of XML documents. Demo at ICDE (2002) pp.341.
15. W. Fan and L. Libkin: On XML integrity constraints in the presence of DTDs. Symposium on Principles of Database Systems (2001) pp.114-125.
16. M. F. Fernandez and D. Suciu: Optimizing regular path expressions using graph schemas. ICDE (1998) pp.14-23.
17. A. Gabillon and E. Bruno: Regulating Access to XML Documents. Working Conference on Database and Application Security (2001) pp.219-314.
18. A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne: Document Object Model (DOM) Level 3 Core Specification. Available at http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205 (2004).

19. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth: Covering indexes for branching path queries. ACM SIGMOD (2002) pp.133-144.
20. D. D. Kha, M. Yoshikawa, and S. Uemura: An XML Indexing Structure with Relative Region Coordinate. ICDE (2001) pp.313-320.
21. M. Kudo and S. Hada: XML Document Security based on Provisional Authorization. ACM CCS (2000) pp.87-96.
22. Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB (2001) pp.361-370.
23. G. Miklau and D. Suciu: Containment and equivalence for an XPath fragment. ACM PODS (2002) pp.65-76.
24. M. Murata, A. Tozawa, M. Kudo and S. Hada: XML Access Control Using Static Analysis. ACM CCS (2003) pp.73-84.
25. F. Neven and T. Schwentick: XPath containment in the presence of disjunction, DTDs, and variables. ICDT (2003) pp.315-329.
26. Y. Papakonstantinou and V. Vassalos: Query rewriting for semistructured data. ACM SIGMOD (1999) pp.455-466.
27. P. T. Wood: Containment for XPath fragments under DTD constraints. ICDT (2003) pp.300-314.
28. T. Yu, D. Srivastava, L. V. S. Lakshmanan, and H. V. Jagadish: Compressed Accessibility Map: Efficient Access Control for XML. VLDB (2002) pp.478-489.