The VLDB Journal (2007) 16:371–387
DOI 10.1007/s00778-006-0037-7

REGULAR PAPER

The dynamic predicate: integrating access control with query processing in XML databases

Jae-Gil Lee · Kyu-Young Whang · Wook-Shin Han · Il-Yeol Song

Received: 30 September 2005 / Revised: 10 March 2006 / Accepted: 13 July 2006 / Published online: 19 December 2006

Abstract Recently, access control on XML data has become an important research topic. Previous research on access control mechanisms for XML data has focused on increasing the efficiency of access control itself, but has not addressed the issue of integrating access control with query processing. In this paper, we propose an efficient access control mechanism tightly integrated with query processing for XML databases. We present the novel concept of the *dynamic predicate* (DP), which represents a dynamically constructed condition during query execution. A DP is derived from instance-level authorizations and constrains accessibility of the elements. The DP allows us to effectively integrate authorization checking into the query plan so that unauthorized elements are excluded in the process of query execution. Experimental results show that the proposed access control mechanism improves query processing time significantly over the state-of-the-art access control mechanisms. We conclude that the DP is highly effective in efficiently checking instance-level authorizations in databases with hierarchical structures.

Keywords Access control · Query processing · XML databases · Privacy/security

J.-G. Lee (✉) · K.-Y. Whang
Department of Computer Science and Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology(KAIST), 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, South Korea
e-mail: jglee@mozart.kaist.ac.kr

K.-Y. Whang
e-mail: kywhang@mozart.kaist.ac.kr

W.-S. Han
Department of Computer Engineering, Kyungpook National University, 1370 Sankyuk-dong, Book-gu, Daegu 702-701, South Korea
e-mail: wshan@knu.ac.kr

I.-Y. Song
College of Information Science and Technology, Drexel University, Philadelphia, PA 19104, USA
e-mail: song@drexel.edu

1 Introduction

The amount of private data stored in databases is rapidly growing [24], and the awareness of privacy protection on data is rapidly increasing [1,2,25]. Among many research issues, access control has played a crucial role in privacy protection [1]. Access control prevents an adversary from illegitimately accessing private data stored in databases. As XML is emerging as a new standard for data representation and exchange on the internet, access control on XML data has become an important research topic [5,7,8,10,12–14,20,23,26,34].

Several access control models for XML data have been reported in the literature [5,12,14]. These models provide ways to grant authorizations and to control access to XML data through authorizations. That is, these models allow users to access only authorized XML data elements. These models have some common characteristics: an authorization can be specified at the schema level or at the instance level; an authorization on an element implies those on its descendant elements in the XML data hierarchy (called an *implicit authorization*); and an explicit authorization can override an implicit authorization. These characteristics make access control in XML databases complicated. Thus, access control tends to be time-consuming and degrades query-processing performance.

There have been a number of efforts to develop efficient access control mechanisms for XML data. Previous research has mainly dealt with the methods of efficiently searching for authorizations [5,12,13,26,34] or those of effectively reducing the number of authorization checks that need to be performed at run time [10,20,23]. These research activities have successfully improved the performance of access control itself. In general, access control should be accompanied by query processing to return only authorized query results. Nevertheless, despite the close relationship between access control and query processing, there has not been any work towards integration of these two kinds of operations.

We contend that the tight integration of access control with query processing can significantly improve query-processing performance. Let us present a simple example. Suppose that a user issues a query that retrieves all the *drug* elements in an XML document and that a very large proportion of *drug* elements are unauthorized. If a query processor is aware of the authorization information, it can exclude beforehand most of the *drug* elements from query processing, thus drastically saving query processing cost. Hence, the authorization information can be exploited to help the query engine better optimize query evaluation.

In this paper, we develop an efficient access control mechanism applicable to existing access control models [5,12,14] for XML data. The novelty of our access control mechanism is tight integration of access control with query processing. The key idea is to regard an authorization as a query condition to be satisfied. The development consists of two steps: (1) devising a mechanism that efficiently searches for authorizations and (2) integrating this access control mechanism with the query plan.

We first propose an access control mechanism that utilizes an authorization index and nearest neighbor search technique [18] for efficiently searching for authorizations. We implement an authorization index as a multi-dimensional index [15]. The proposed mechanism first maps the elements in XML data with authorizations to two-dimensional points and store them in the authorization index. Since accessibility of an element is determined by the explicit authorization specified on the nearest ancestor element with an explicit one [5,10,12,26], we adopt a nearest neighbor search technique to efficiently find that explicit authorization.

We then propose the notion of the *dynamic predicate* (*DP*), which represents a dynamically constructed condition during query execution. We note that accessibility of an element is dynamically determined during query execution because authorizations can be specified at the instance level. The DP is applied for checking the accessibility of such elements during query execution. Here, the DP indicates which elements are authorized or unauthorized. By inserting the DP into the query plan, we effectively integrate access control with query processing. Due to this integration, unauthorized elements are excluded in an early phase of query processing, thus significantly improving the performance.

There are two major reasons why we need a mechanism different from those for relational databases: instance-level authorizations and data hierarchies. Typically, relational databases support only schema-level authorizations without supporting instance-level authorizations. Supporting instance-level authorizations is expensive since we need to access the elements themselves to check their accessibility. We reduce the cost by using the notion of the *DP*, i.e., by representing the authorization information of a set of elements having an identical authorization with that of the element being currently accessed. The DP is highly effective in checking instance-level authorizations in databases with hierarchical structures. A large number of data elements can be filtered out along the data hierarchies, thus making the DP more effective. We call this feature *hierarchical filtering*. In contrast, relational databases have flat structures and cannot take advantage of this hierarchical filtering.

Recently, the importance of information inference is being widely recognized [1]. Information inference is the ability of inferring the information that a user is not permitted to know using the information that the user knows [1]. The extent of information inference is dependent on the strategy for determining authorized query results. There are two different strategies for determining authorized query results. The first one is to check authorizations only for the instances returned as the query result [5,12,20,23,26]. The second one is to check authorizations for all the instances of other elements that qualify result instances as well as for the result instances themselves [10]. The second strategy significantly reduces information inference, and we call it the *inference-blocking* strategy. To show progressive development, we first develop our access control mechanism for the first strategy in Sects. 4 and 5, and then, proceed to the second strategy in Sect. 6.

In summary, the contributions of this paper are as follows:

- We propose a new notion of the authorization index combined with the nearest neighbor search technique that allows effective determination of accessibility.
- We propose a new notion of the DP that allows effective integration of access control with query processing.

- We propose an efficient access control mechanism tightly integrated with query processing—using the notion of the DP.
- We make our access control mechanism minimize information inference by adopting the inference-blocking strategy.
- We demonstrate, by extensive experiments, that our access control mechanism significantly outperforms existing mechanisms.

The rest of this paper is organized as follows. Sect. 2 describes existing access control models and mechanisms for XML data. Section 3 presents a simple access control mechanism. Section 4 proposes an integrated access control mechanism that uses DPs. Section 5 discusses our approach to reducing information inference. Section 6 presents the results of performance evaluation. Finally, Sect. 7 concludes the paper.

2 Related work

We explain existing access control models [5,12,14] for XML data and access control mechanisms in these models. These models provide ways to grant/revoke authorizations on XML data stored in the database and to control access to authorized XML data.

2.1 Access control models

An authorization in existing access control models is generally defined as a 5-tuple $< s, o, a, sign, imply\_option >$ [5,12,14]. Here, s is a user, a user group, a role, or a credential [7] to whom an authorization is granted; o an XML document or an XML element/attribute protected by the authorization; a an action being allowed or prohibited; *sign* has one of + or −, which indicates whether the action is allowed or prohibited; *imply_option* represents whether the authorization implies [1] authorizations on descendant elements. s is called the *authorization subject*, o the *authorization object*, and a with *sign* the *authorization type*.

Authorizations can be specified at the schema level or at the instance level. Schema-level authorizations are specified on a DTD, and these are applicable to all XML documents that are instances of the DTD. Instance-level authorizations are specified on an XML document, and these are applicable only to the XML document to which the authorizations have been granted.

Authorizations are classified into explicit or implicit, strong or weak, and positive or negative ones [12,27]. An *explicit authorization* is explicitly specified on an element of a DTD or an element of an XML document; an *implicit authorization* is implied by an explicit authorization specified on the nearest ancestor element with an explicit one. A *strong authorization* does not allow an implicit authorization to be overridden; a *weak authorization* allows an implicit authorization to be overridden by an explicit authorization. A *positive authorization* allows accesses for a specific action; a *negative authorization* prohibits accesses for a specific action.

Implication of an authorization introduces potential conflicts among authorizations. That is, an element may have both implicit and explicit authorizations for different authorization types. To resolve these conflicts, many access control models for XML data use the *most specific overrides* policy [5,10,12]. An explicit authorization granted on a lower-level element always overrides a weak authorization granted on a higher-level element according to this policy. On the other hand, any explicit authorization cannot be granted on a lower-level element of an element having a strong authorization by definition.

2.2 Access control mechanisms

Due to implicit authorizations, authorizations on the ancestors should be examined to determine accessibility of an element. Bertino et al. [5] have proposed top-down and bottom-up strategies that determine the accessibility by traversing paths between the root and the element to be examined. Top-down strategies traverse paths beginning from the root; bottom-up strategies traverse paths beginning from the element to be examined. Thus, both strategies access the ancestors of the element to be examined. If elements to be examined are scattered in the database, in the worst case, these strategies may access the whole database to check authorizations [5].

Damiani et al. [12] have proposed a view-based access control mechanism. This mechanism creates and maintains a separate view for each user, which contains exactly the set of data elements that the user is authorized to access. After views are constructed, users can simply run their queries against the views. Although views are prepared off-line, in general, this mechanism suffers from high maintenance and storage costs especially for a large number of users [20]. Fan et al. [13] have proposed another view-based access control mechanism. This mechanism generates not only a view (called a *security view*), but also a view DTD to which the security view conforms. The view DTD is used for improving efficiency of query rewriting and optimization. This mechanism also shares the common drawback

[1] Most access control models for XML data use the term "propagate," but we use the term "imply" for consistency.

of view-based access control mechanisms—high maintenance cost for a large number of users [8].

Murata et al. [23] have proposed a static analysis method for reducing the number of authorization checks during run time. This method classifies an XML query at compile time into three categories: entirely authorized, entirely prohibited, or partially authorized ones. Entirely authorized or entirely prohibited queries can be executed without access control. However, the static analysis cannot obtain any benefits when a query is classified as a partially authorized one. To remedy this problem, Luo et al. [20] have proposed a query-rewriting method *QFilter* that converts a partially authorized query into an entirely authorized one. QFilter uses the Non-deterministic Finite Automata (NFA) for query rewriting. However, a query rewritten by QFilter tends to be complicated (i.e., contain unions of many path expressions) as the number of authorization increases. Both methods, however, are able to support only schema-level authorizations since they do not examine the actual database.

Recently, some access control mechanisms whose models are slightly different from traditional ones have been proposed. Yu et al. [34] use an access control model where an authorization on an element does not imply authorizations on its descendant elements. In this type of model, the number of authorizations tends to proliferate since every authorization has to be explicitly granted. Thus, Yu et al. have proposed the *compressed accessibility map* (*CAM*), a mechanism of compressing neighboring authorizations to avoid such proliferation. Cho et al. [10], using an access control model where instance-level authorizations are granted only according to predetermined patterns (called *security annotations*), have proposed an optimization method (which we call *SCA* for convenience) minimizing the number of security checks that need to be performed at run time. Instead of unconditionally performing a recursive check on the ancestors of an element, the SCA method optimally determines when a recursive check can be eliminated or simplified to a local check on the element.

2.3 Comparison with our access control mechanism

Our access control mechanism has desirable properties compared with earlier access control mechanisms. First, it can speed up query processing by excluding unauthorized elements early on during query processing. Second, it obviates the need for accessing the whole database by searching only the authorization index as opposed to the top-down and bottom-up strategies [5]. Third, it does not suffer from the overhead of maintaining views as opposed to the view-based access control mechanisms [12,13]. Fourth, it supports instance-level authorizations as opposed to the static analysis methods [20,23] which handle only schema-level authorizations.

3 A simple access control mechanism

In this section, we present a simple access control mechanism. We first present the problem definition in Sect. 3.1. We discuss the structure of the *authorization index* that stores instance-level authorizations in Sect. 3.2 and propose a simple access control algorithm that utilizes the authorization index and nearest neighbor search technique in Sect. 3.3.

3.1 The problem definition

We develop an efficient access control mechanism applicable to existing access control models [5,12,14] for XML data. That is, given an access control policy (i.e., a set of authorizations granted), an XML document, and a query, our access control mechanism retrieves the query results authorized according to the access control policy. The XPath [11] language, which is a core component of the XQuery language, is used for specifying XML queries.

We first summarize the features of the access control model adopted in this paper. As stated in Sect. 2, an authorization is defined as a 5-tuple $< s, o, a, sign, imply\_option >$. A path expression confirming to the XPath [11] standard is used for specifying an authorization object. Hence, we are able to support protection granularity levels ranging from an XML document to an XML element/attribute. If the path expression points to the root element, the granularity level is the whole document. If a set of elements is generated as the result of the path expression, we assume that the authorization is granted on every element. Besides, we support the content (i.e., the text node) of an element as the granularity level. Here, the content of an element is also protected by the authorization on the element as in most access control models. Authorizations are classified into explicit or implicit, strong or weak, and positive or negative ones. The authorization type of a positive authorization is represented by a or $+a$, and that of a negative authorization by $\neg a$ or $-a$.

The *most specific overrides* [5,10,12,27] policy is employed to resolve conflicts among authorizations. That is, an explicit authorization on an element overrides any authorizations specified on the ancestors of the element. On the other hand, if there exists no authorization—either explicit or implicit—on an element, the element is considered as inaccessible [12,14,20,23,26].

It is possible to specify multiple explicit authorizations for different authorization types on the same element, thus allowing accesses to the element for multiple actions (e.g., for both *read* and *update*). If both a positive explicit authorization and a negative explicit authorization for the same authorization type are specified on the same element, however, the negative one overrides the positive one (called the *denials take precedence* policy) [5,12,14,20,23,26].

We now explain the ways of performing access control. Executing a query requires authorizations for a specific authorization type (e.g., *read*), and we call it the *authorization type* of the query. The elements on which an authorization has been granted for the authorization type of the query are accessible to the user, but other elements are inaccessible. We elaborate on the instance-level checking since this checking is very expensive compared with the schema-level checking. Hereafter, we refer to instance-level authorizations simply as *authorizations*.

Example 1 Figure 1 shows an example of an XML document and authorizations. Let us call this XML document *hospital.xml*. The authorization <$user_A$, *document*(*"hospital.xml"*)*/hospital, read, +, imply* [2]> allows $user_A$ to read the *hospital* element and its descendant elements of *hospital.xml*. However, the authorization <$user_A$, *document*(*"hospital.xml"*)*/hospital/patient*[2], *read, −, imply*> overrides the authorization above and prohibits $user_A$ from reading the second *patient* element and its descendant elements. Suppose $user_A$ issues a query *//patient//drug* which requires *read* authorizations. Here, the only three *drug* elements under the first *patient* element are retrieved as the query result.

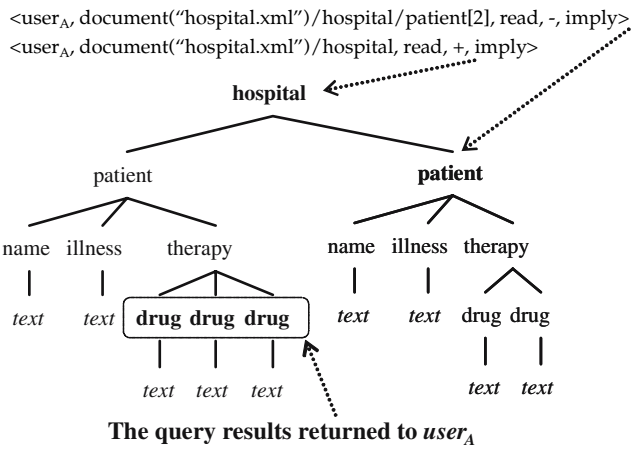

Fig. 1 An example of an XML document and authorizations

3.2 Authorization indexes

We use a two-dimensional index [15] to implement the authorization index. This is because elements with an explicit authorization are represented by two-dimensional points (*start*, *end*) according to the numbering scheme [3,6,9,19] used in many XML query-processing methods. The (*start*, *end*) points are stored in the authorization index. In the numbering scheme, (*start*, *end*) represents an ancestor–descendant relationship between XML elements and satisfies the following two conditions [19]: (1) for any element u and its parent element v, the interval $(start_u, end_u)$ is contained in the interval $(start_v, end_v)$; (2) for two sibling elements u and v, if u is a predecessor of v in preorder traversal, then $end_u < start_v$. Therefore, a is an ancestor of d if and only if $start_a < start_d \wedge end_a > end_d$.

For the authorization index, we can use any kind of indexes that can handle multi-dimensional points. The MLGF [32], R-tree [17], buddy tree [31], and quad tree [29] are examples.

Example 2 Figure 2 illustrates an example of the authorization index implemented using the MLGF. Suppose that authorizations are granted on the elements whose (*start*, *end*) numbers are (7, 9), (10, 20), (26, 28), (29, 36), and (37, 44), respectively. These two-dimensional points are depicted in the two-dimensional space as in Fig. 2a and are indexed in the MLGF as in Fig. 2b. The MLGF is a height-balanced index tree that stores multi-dimensional points [32]. A non-leaf page contains entries of <*region, ref*>, where *ref* is the pointer to the child page, and *region* contains all the regions represented by the entries of the child page pointed by *ref*. A leaf page contains entries of <*point, oid*>, where *point* is the coordinate of the point, and *oid* is the identifier of the object stored in the database.

3.3 Simple access control algorithm using the authorization index and nearest neighbor search

Due to the most specific overrides policy, we can determine accessibility of an element by seeing the authorization granted only on the nearest ancestor element regardless of its authorization type. That is, an element is accessible if the authorization type of this authorization is the same as that of the query; inaccessible otherwise. We refer to this authorization as the *nearest ancestor authorization*. We formally define it in Definition 1 and find it using Lemma 1.

[2] For simplicity, we assume that an authorization on an element always implies authorizations on its descendant elements.

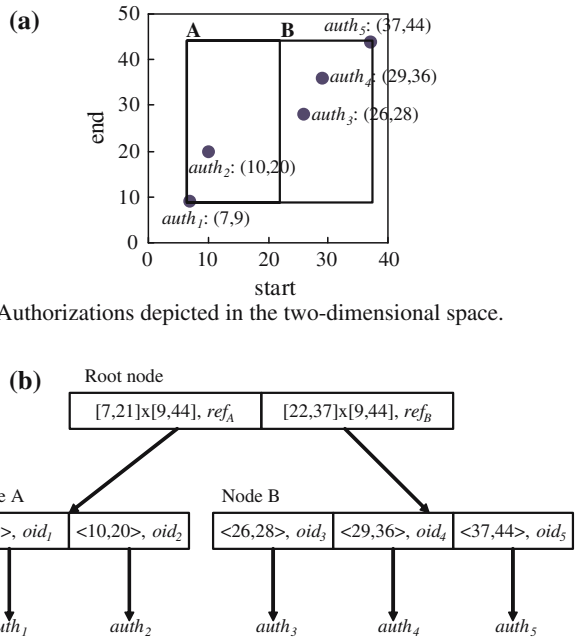

Fig. 2 An example of the authorization index implemented using the MLGF

Definition 1 *An authorization is called the* nearest ancestor authorization $auth_{naa}$ *of the element e if it satisfies the following two conditions:* (1) $auth_{naa}$ *is an explicit authorization granted on the element e or one of its ancestor elements regardless of its authorization type;* (2) *no explicit authorization exists on elements in the path between the element e and the element on which* $auth_{naa}$ *is granted. If a strong authorization satisfies the first condition, it automatically satisfies the second condition by the definition of the strong authorization.*

Lemma 1 *The nearest ancestor authorization* $auth_{naa}$ *of the element e is the one that minimizes* $|start(e) - start(auth)|$*, where auth is an authorization that satisfies* $start(auth) \leq start(e) \wedge end(auth) \geq end(e)$*. That is,* $auth_{naa}$ *has the minimum difference between* $start(e)$ *and* $start(auth)$ *where auth is an authorization located in the upper-left region of the element e in the two-dimensional space. Here,* $start(e)$ *and* $end(e)$ *represent start and end values of the element e;* $start(auth)$ *and* $end(auth)$ *those of the element on which auth is granted.*

Proof See Appendix A. □

We note that multiple nearest ancestor authorizations can exist due to multiple authorizations on the same element. Suppose there exist multiple nearest ancestor authorizations $auth_{naa}^{(1)}, \ldots, auth_{naa}^{(k)}$ whose authorization types are $a_{naa}^{(1)}, \ldots, a_{naa}^{(k)}$, respectively. In this case, we regard them as one authorization whose authorization type is obtained by ORing individual $a_{naa}^{(i)}$'s (i.e., $\bigvee_{i=1}^{k} a_{naa}^{(i)}$).

```
Algorithm Nearest Ancestor Filtering
Input: (1) a set of XML query results R and
           an authorization type a_q of a query
       (2) the authorization index storing authorizations
Output: authorized query results
Algorithm:
01:  for each query result r ∈ R do
         /* search the authorization index */
02:      Find the nearest ancestor authorization auth_naa of r
         according to Lemma 1;
03:      if the authorization type of auth_naa is a_q then
04:          output r;
```

Fig. 3 The simple access control algorithm Nearest Ancestor Filtering

We now propose a simple access control algorithm that uses the authorization index and nearest neighbor search technique. We call it *Nearest Ancestor Filtering*. The algorithm executes a query and, for each query result, searches for the nearest ancestor authorization according to Lemma 1 to check authorizations. Here, the nearest ancestor authorization can be retrieved by using a nearest neighbor search technique [18]. Finally, the algorithm examines whether the authorization type of the nearest ancestor authorization is the same as that of the query, thus checking accessibility of the query result. We note that, to find the nearest ancestor authorization, Nearest Ancestor Filtering accesses the authorization index only once, while the top-down and bottom-up strategies access many ancestors and find the authorization for each ancestor accessed [5]. Figure 3 shows the algorithm Nearest Ancestor Filtering.

4 An integrated access control mechanism

In this section, we propose an access control mechanism tightly integrated with query processing, which is the primary contribution of this paper. We first present an overview in Sect. 4.1, and then, propose a new notion of the dynamic predicate and a technique that integrates authorization checking into the query plan using this notion in Sect. 4.2.

4.1 Overview

The key operation in the integrated access control mechanism is identifying a set of elements that have the same accessibility. The start values of those elements can be formed as an interval, and we call it the *start* interval. A

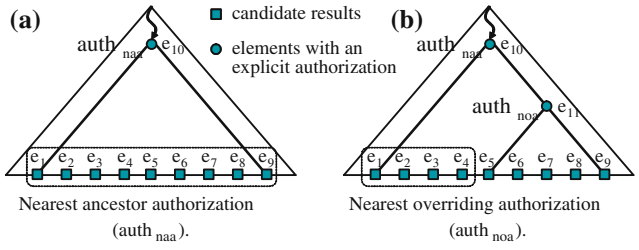

Fig. 4 Elements with the same nearest ancestor authorization

start interval represents either a set of authorized elements or a set of unauthorized elements. During query processing, the start interval can be used to determine accessibility of an element by examining whether the start value of the element is contained in this start interval. With this method, we do not have to search the authorization indexes for each query result as in the algorithm Nearest Ancestor Filtering.

A *start* interval of elements with the same accessibility can be constructed as the interval of elements with the same nearest ancestor authorization because accessibility of an element e is determined by the nearest ancestor authorization of e by Definition 1. Figure 4 shows the elements having the same nearest ancestor authorization. In Fig. 4a, the nearest ancestor authorization of element e_1 is $auth_{naa}$, granted on e_{10}, and no authorization overrides $auth_{naa}$. Therefore, elements e_1–e_9, which are the descendant elements of e_{10}, have $auth_{naa}$ as the nearest ancestor authorization. However, in Fig. 4b, only elements e_1–e_4 have $auth_{naa}$ as the nearest ancestor authorization because $auth_{noa}$, granted on e_{11}, overrides $auth_{naa}$. To consider this case, we define the *nearest overriding authorization* $auth_{noa}$ of the nearest ancestor authorization $auth_{naa}$ as the one that first overrides $auth_{naa}$, when traversing the XML data tree from a current element in preorder. We formally define it in Definition 2 and find it using Lemma 2.

Definition 2 *Suppose $auth_{naa}$ is the nearest ancestor authorization of the element e and is a weak authorization. An authorization is called the* nearest overriding authorization *$auth_{noa}$ of $auth_{naa}$ if it satisfies the following two conditions:* (1) *$auth_{noa}$ is an explicit authorization granted on a descendant element of the element having $auth_{naa}$ regardless of its authorization type;* (2) *no other explicit authorization exists, when traversed in preorder, between the element e and the element having $auth_{noa}$. If $auth_{naa}$ is a strong authorization, however, $auth_{noa}$ does not exist by the definition of the strong authorization.*

Lemma 2 *Suppose $auth_{naa}$ is the nearest ancestor authorization of the element e and is a weak authorization. Then, the nearest overriding authorization $auth_{noa}$ of $auth_{naa}$ is the one that minimizes $|start(e) - start(auth)|$, where auth is an authorization that satisfies $start(auth) > start(e) \geq start(auth_{naa}) \wedge end(auth) < end(auth_{naa})$.*

Proof See Appendix B. □

We now present Lemma 3 for constructing the start interval having the same nearest ancestor authorization as that of the element e. All the elements whose start value is contained in this interval are accessible or inaccessible depending on the authorization type of the nearest ancestor authorization of the element e.

Lemma 3 *Suppose that the nearest ancestor authorization of the element e is $auth_{naa}$, and the nearest overriding authorization of $auth_{naa}$ is $auth_{noa}$. If $auth_{naa}$ does not exist, we assume that $auth_{naa}$ is a negative authorization specified on the root element. This assumption is reasonable since an element with no authorization is regarded as inaccessible in our access control policy. Then, any element whose start value is contained in the following interval has the same nearest ancestor authorization as that of the element e.*

- *if $auth_{noa}$ exists:* $[start(e), start(auth_{noa}))$
- *if $auth_{noa}$ does not exist:* $[start(e), end(auth_{naa}))$

Proof See Appendix C. □

4.2 Dynamic predicates (DPs)

We propose the notion of the dynamic predicate[3], which represents a dynamically constructed condition during query execution. We note that, due to instance-level authorizations, the start interval constructed by Lemma 3 is *data-dependent* and should be evaluated at run time. The DP inserted in the query plan allows this data-dependent start interval to be evaluated at run time. By evaluating the start interval, we can filter out unauthorized elements from query processing early on, thus significantly improving query performance. Therefore, we apply the notion of the DP for determining accessibility of elements so as to tightly integrate access control

[3] Oracle has used the term "dynamic predicate" for a meaning different from that of our DP. Oracle Enterprise Edition provides a facility that automatically appends a predicate to a user's query at query compilation time. Oracle refers to this predicate as a DP. It is defined by the database administrator based on his access control policy. The DP is used for hiding some tuples from a user just like the way views are used for access control. We note that the DP in Oracle is static rather than dynamic. It is determined during query compilation and is not changed during query execution. In contrast, the DP in this paper is indeed dynamic because it is changed continuously due to its property of being data-dependent.

with the query plan. Definition 3 formally defines the DP when it is applied to access control.

Definition 3 *A dynamic predicate for the element e is a condition dynamically constructed during query execution, representing a start interval of elements with the same accessibility. A DP is represented as* $([start_{begin}, start_{end}), sign)$. *Here,* $[start_{begin}, start_{end})$ *represents the start interval, and sign represents accessibility (+) or inaccessibility (−) of the elements that belong to this start interval. A DP is constructed as follows:*

(1) $[start_{begin}, start_{end})$: *the start interval generated according to Lemma 3;*

(2) *sign:* $\begin{cases} + & \text{if the nearest ancestor authorization} \\ & \text{allows accesses to the element } e; \\ - & \text{otherwise.} \end{cases}$

Example 3 Suppose $user_A$ issues a query *//patient//drug*, which requires *read* authorizations, against the XML document in Fig. 5. During query processing, a *drug* element (11, 13) is retrieved, and then, the DP for (11, 13) is constructed according to Definition 3 as follows. (1) The nearest ancestor authorization of (11, 13) is the one granted on (2, 46) by Definition 1, and its nearest overriding authorization is the one granted on (22, 45) by Definition 2. This case falls into the first category in Lemma 3, and thus, $[start_{begin}, start_{end})$ becomes [11, 22). (2) The nearest ancestor authorization, which is granted on (2, 46), prohibits $user_A$ from reading the element (11, 13) because its authorization type is $\neg read$. Thus, *sign* of the DP becomes −. Therefore, if we insert ([11, 22), −) into the query plan, *drug* elements (11, 13) ∼ (17, 19) are excluded from the query results, thus improving the query performance.

Only DPs with the − sign can enhance the performance when they are inserted into the query plan, because these predicates exclude elements. DPs with the + sign make the elements whose start values are contained in $[start_{begin}, start_{end})$ be included into query processing. However, DPs with the + sign should also be inserted into the query plan because a DP with the + sign allows us to find the element for constructing the next DP. This element is the first one whose start value is not contained in $[start_{begin}, start_{end})$.

The earlier the DPs are evaluated in the query plan, the earlier we can exclude unauthorized elements from query processing, thus improving query performance. Thus, we *push down* DPs in the query plan. It is similar to query optimization in relational databases, pushing down selection predicates to reduce the number of retrieved tuples as early as possible. While the query optimization in relational databases pushes down selection predicates only once during query compilation, however, the proposed optimization technique dynamically generates the predicates and consecutively pushes them down during query processing.

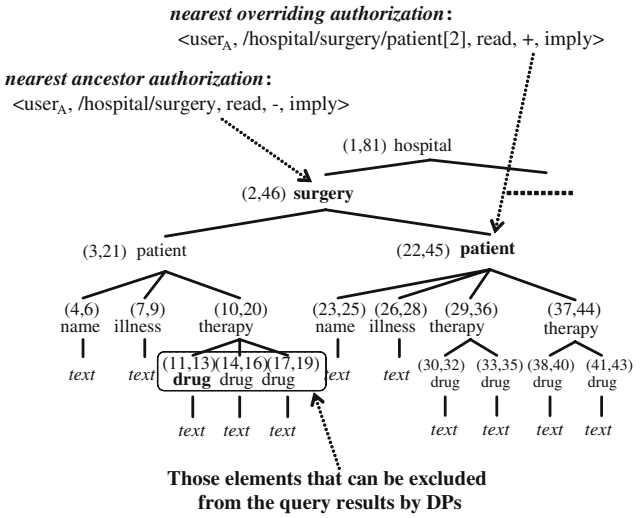

Fig. 5 An example that shows performance improvement when using DPs

To *push-down* DPs, we first insert selection (σ) operators into the query plan right above the operators that scan query elements during query compilation, and then, assign DPs as the predicates to these selection operators during query processing.

Figure 6 shows query plans for executing the query $//e_A//e_B//e_C$ before and after using DPs. The query plan is constructed using the scheme proposed by Wu et al. [33] extended with authorization operators. Here, the query plan is similar to that of relational algebra: the query plan is a tree where query elements are leaf nodes, and operators are internal nodes. Figure 6a shows a basic query plan that does not use DPs. The authorization checking operator Ω executes the algorithm Nearest Ancestor Filtering presented in Sect.3.3 and returns authorized results. The operator // returns pairs of ancestor–descendant elements. A well-known processing method of the operator // is the structural join [3,9, 19]. The $scan(e_A)$, $scan(e_B)$, and $scan(e_C)$ operators scan each element in the order of start values to execute the structural join. The basic query plan is constructed in such a way that it first retrieves each result of the query $//e_A//e_B//e_C$, and then, performs authorization checking with the operator Ω. The output of each operator is pipelined to the input of the parent operator [16].

Figure 6b shows the query plan augmented with DPs. This plan is constructed such that the DP push-down operator Ψ is inserted at top of the query plan, and σ_{DP}

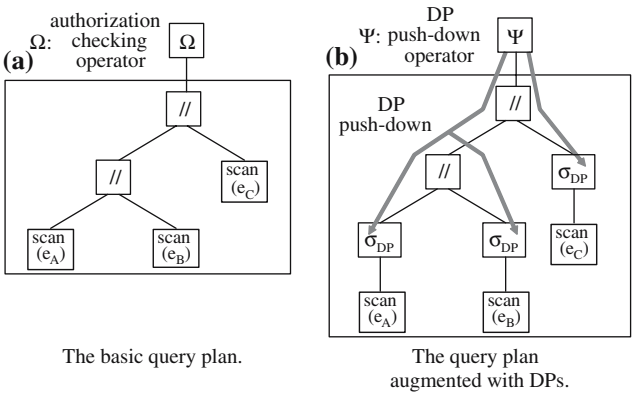

Fig. 6 Query plans before and after using DPs

operators right above $scan(e_A)$, $scan(e_B)$, and $scan(e_C)$ operators. The role of the Ψ operator is to initialize and update DPs as query processing proceeds, pushing them down to σ_{DP} operators. Here, the arrow lines indicate that DPs generated by the Ψ operator are pushed down to the σ_{DP} operators. By using these DPs, only authorized instances of e_A, e_B, and e_C are provided to the input of // operators.

The DP push-down operator Ψ examines whether the nearest ancestor authorization of each query result retrieved is different from that of the previous query result. Only when it is different, the operator Ψ pushes down a newly constructed DP into σ_{DP} operators. The algorithm of the operator Ψ is constructed by using the *iterator* [16] model commonly used in query processors of commercial DBMSs. In the iterator model, the operators composing the query plan receive an input from the child operators, and then, provide the processed result to the parent operator. To obtain a result from each operator, it provides GetNext() function as the interface.

Figure 7 shows the algorithm for GetNext() of the operator Ψ. We call it *Dynamic Predicate Filtering*. The algorithm consists of two steps. In the first step, the algorithm pushes down the constructed DP into the child operator P and retrieves each authorized query result by calling P.GetNext() (lines 2–6). At this time, it transmits the DP to the child operator through the argument of P.GetNext(). This DP is recursively transmitted to all the σ_{DP} operators. When Ψ.GetNext() is first called, the algorithm retrieves the first candidate query result $e_{candidate}$ and constructs an initial DP for $e_{candidate}$ using Definition 3 (lines 3–5). In the second step, the algorithm constructs the DP using Definition 3 only when the DP for the query result e needs to be reconstructed (lines 7–8). $[start_{begin}, start_{end})$ of the DP is the start interval representing the set of elements with the same nearest ancestor authorization as that of the previous query result. Therefore, we can easily determine the element

```
Algorithm Dynamic Predicate Filtering (Basic)
/* DP: a Dynamic Predicate */
01: Ψ.GetNext() {
        /* Ψ has only one child */
02:     Let P be the child operator of the DP push-down operator Ψ;
        /* initially, no DP has been constructed */
03:     if ( Ψ.GetNext() is first called ) then
04:         Let e_candidate be the first instance, which has the smallest
            start value, of the element type of the query results;
05:         Construct an initial DP for the instance e_candidate
            using Definition 3;
        /* get an authorized query result and push down the DP */
        /* P.GetNext() returns a result of P
           recursively transmitting the DP to σ_DP operators */
06:     e := P.GetNext(DP);
07:     if ( start(e) is not contained in [start_begin, start_end) of DP ) then
            /* reconstruct the DP */
08:         Construct the DP for the element e using Definition 3;
09:     return e;   /* return an authorized query result */
10: }
```

Fig. 7 The integrated access control algorithm Dynamic Predicate Filtering (Basic)

that triggers reconstruction of the DP by examining whether the start value of the element e is contained in this interval (line 7).

Example 4 Figure 8 shows an example of executing the algorithm Dynamic Predicate Filtering (Basic), when issuing a query *//patient//drug* against the XML document in Fig. 5. In line 4 of Fig. 7, the algorithm retrieves the first candidate query result, *drug* element (11, 13). In line 5, $DP = ([11, 22), -)$ is constructed as shown in Example 3 using Definition 3. In line 6, the algorithm pushes down $DP = ([11, 22), -)$ all the way to the σ_{DP} operators by calling P.GetNext(). Here, the operator P is the // operator. Then, *drug* elements, (11, 13), (14, 16), and (17, 19), whose start values are contained in [11, 22), are excluded from query processing. (Filtering *patient* elements is discussed in Example 6.) Thus, the algorithm retrieves *drug* element (30, 32) which is next to [11, 22). In line 8, $DP = ([30, 45), +)$ is newly constructed, which is used when Ψ.GetNext() is called next.

Dynamic Predicate Filtering is optimally applied to the query plan that satisfies the following condition: elements are scanned and query results generated in the order of start values. Most structural join algorithms satisfy this condition [3,6,9].[4] Thus, we fully exploit the properties of structural join algorithms, which are the most popular methods for processing XML queries.

[4] When this condition is not satisfied, however, we can simply extend Dynamic Predicate Filtering in such a way that it accumulates DPs instead of replacing the old one with a new one (although the performance will not be as good).

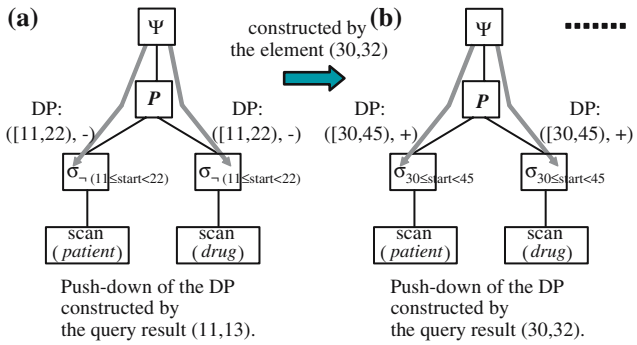

Fig. 8 An example of executing the algorithm Dynamic Predicate Filtering (Basic)

Dynamic Predicate Filtering can be used also for twig queries [6]: for example, $//e_A[.//e_B//e_C]//e_D$. Twig queries are processed in two consecutive phases [6]. In the first phase, individual linear paths are extracted from the twig query ($//e_A//e_B//e_C$ and $//e_A//e_D$ in the example query), and the result for each is retrieved. In the second phase, results for individual linear paths are merge-joined. We can use Dynamic Predicate Filtering during the first phase that handles only linear path queries.

5 Further enhancement of the access control mechanism to minimize information inference

Information inference is the ability of inferring the information that a user is not permitted to know using the information that the user knows [1]. We discuss a further enhancement of our access control mechanism to minimize information inference. We determine authorized query results using the following strategy: we check authorizations for all the instances of other elements that qualify result instances as well as for the result instance themselves.[5] We call it the *inference-blocking* strategy. This strategy can significantly reduce information inference compared with that presented in previous sections.

Example 5 Consider the XML document in Fig. 5. Suppose that a user is allowed to access the *drug* elements, but not the *patient* elements. If only query results are checked for authorizations, a query *//patient*[*./name ='Lee']//drug* could allow the user to infer the existence of a patient named 'Lee.' That is, non-empty query results indicate the existence of such a patient. This inference clearly violates the desired intent. In contrast, the inference-blocking strategy prevents this inference. Since the user is not allowed to access the *patient* elements, the query result is always empty regardless of the existence of such a patient.

We first define the instances for which we need to check authorizations. We call the set of these instances the *result tree* and define it using the scheme proposed by Miklau and Suciu [22]. Miklau and Suciu model an XML document and an XPath query as trees. Here, a node of an XML document is an element instance, and that of an XPath query is an element type. A tree representing an XML document is called an *XML tree*, and that representing an XPath query a *tree pattern*. The node representing the query result in a tree pattern is called the *distinguished output node*. The set of edges of a tree pattern is partitioned into the set of the parent–child edges and the set of the ancestor–descendant edges. Miklau and Suciu define an *embedding* as a mapping between the nodes of an XML document and the nodes of an XPath query. Given a tree pattern p and an XML tree t, an *embedding* e of p into t is a total function e : {nodes of p} $\rightarrow$ {nodes of t} which satisfies the following condition: (1) $\forall(x, y) \in$ {parent–child edges of p}, $e(y)$ is a child of $e(x)$ in t; (2) $\forall(x, y) \in$ {ancestor–descendant edges of p}, $e(y)$ is a descendant of $e(x)$ in t; and (3) $\forall x \in$ {nodes of p}, the element name of x is the same as that of $e(x)$.

We now formally define the result tree in Definition 4.

Definition 4 *A* result tree *rt of a tree pattern p is a tree constructed by embedding the nodes of p, that is, by replacing* $x \in$ {nodes of p} *with* $e(x)$. $NODES(rt)$ *denotes the set of nodes in a result tree rt, and* $OUTPUT(rt)$ *the node in rt that corresponds to the distinguished output node in p.*

We now enhance our access control algorithm so as to check authorizations for all the nodes in the result tree of a query. The enhancement of Nearest Ancestor Filtering is straightforward: we search for the nearest ancestor authorization and check the authorization for each node in the result tree retrieved. Figure 9 shows the enhanced algorithm Dynamic Predicate Filtering adopting the inference-blocking strategy. It maintains DPs dedicated to each element type specified in the query to check accessibility of the instances of that element type. The algorithm constructs a DP for each node in the result tree and pushes down all the DPs constructed (lines 3–6). The DP is transmitted to the σ_{DP} operator checking the instances of the element type to which this DP is dedicated. Then, the algorithm examines whether the DP for

[5] The meaning of the qualifying instances becomes ambiguous when XPath queries involve negation. Since most of XPath queries do not allow negation [28], we do not consider it here leaving it as the topic of a future paper.

```
Algorithm Dynamic Predicate Filtering
/* DP: a Dynamic Predicate */
01:  Ψ.GetNext() {
02:      Let P be the child operator of the DP push-down operator Ψ;   /* Ψ has only one child */
03:      if ( Ψ.GetNext() is first called ) then    /* initially, no DP has been constructed */
04:          Let rt_candidate be a tree consisting of the instances having the smallest start value
             for each element type specified in a query;
05:          Construct initial DPs for each node n ∈ NODES(rt_candidate) using Definition 3;
         /* get an authorized result tree and push down the current DPs */
         /* P.GetNext() returns a result (a result tree) of P
            recursively transmitting each DP to the corresponding σ_DP operator */
06:      rt := P.GetNext(DPs);
07:      for ( each node n ∈ NODES(rt) ) do
             /* DP(n) means the DP constructed for the element n */
08:          if ( start(n) is not contained in [start_begin, start_end) of DP(n) ) then
09:              Construct the DP(n) using Definition 3;   /* reconstruct the DP */
10:      return OUTPUT(rt);   /* return an authorized query result */
11:  }
/* This function represents GetNext() of P as well as those of // and σ_DP below P. */
12:  P.GetNext(DPs) {
13:      if ( P is the σ_DP operator ) then
14:          Let Pσ be the child operator of P;   /* Pσ is the scan() operator */
15:          Assign the corresponding DP as the selection predicate to σ_DP;
16:          while ( e := Pσ.GetNext(DPs) ) do
17:              if ( e satisfies the DP ) then return e;   /* return an authorized instance */
18:      else   /* P is the // operator */
19:          Let P_//1 and P_//2 be the child operators of P;   /* // has two children */
             /* rt_1 and rt_2 represent intermediate results from P_//1 and P_//2, respectively */
20:          while ( rt_1 := P_//1.GetNext(DPs) and rt_2 := P_//2.GetNext(DPs) ) do
21:              if ( rt_1 and rt_2 satisfy ancestor-descendant relationship ) then
22:                  return a tree consisting of rt_1 and rt_2;
23:  }
```

Fig. 9 The Dynamic Predicate Filtering algorithm adopting the inference-blocking strategy

each node in the result tree newly retrieved needs to be reconstructed (lines 7–9). *P*.GetNext(), which is called in Ψ.GetNext(), returns a result (i.e., a result tree) of P transmitting recursively each DP to the corresponding σ_{DP} operator (lines 12–23). In lines 20–21, we find the elements that satisfy ancestor–descendant relationship using the scheme proposed by Chien et al. [9]. Chien et al. construct, for each element type in an XML document, two-dimensional indexes storing the $(start, end)$ points of elements to speed up query processing. These indexes are also used to find the instance having the smallest start value for each element type in line 4. We discuss the effect of the inference-blocking strategy on performance in Sect. 6.2.

Example 6 Reconsider Example 4 with Dynamic Predicate Filtering in Fig. 9. The result trees retrieved are pairs of the *patient* element and the *drug* element satisfying ancestor–descendant relationship. For example, the first result tree is the pair of the *patient* element $(3, 21)$ and the *drug* element $(11, 13)$. Dynamic Predicate Filtering maintains two separate DPs: one dedicated to the *patient* element and one dedicated to the *drug* element. For the first result tree, the former becomes $DP(patient) = ([3, 22), -)$, and the latter $DP(drug) = ([11, 22), -)$. Then, the former is transmitted to the σ_{DP} operator above the $scan(patient)$ operator to filter out unauthorized *patient* elements early on. Similarly, the latter is transmitted to that above the $scan(drug)$ operator.

6 Performance evaluation

In this section, we evaluate the performance of our access control mechanism compared with existing ones for XML data. We describe the experimental data and environment in Sect. 6.1 and present the results of the experiments in Sect. 6.2.

6.1 Experimental data and environment

We compare the performance of query processing with six different access control mechanisms: the top-down

strategy, the bottom-up strategy, the CAM method [34], the SCA method [10], Nearest Ancestor Filtering, and Dynamic Predicate Filtering. The first and second ones are considered as naive ones; the third and fourth ones as state-of-the-art ones. We adopt the inference-blocking strategy for determining authorized query results. As the performance measure, we use the wall clock time.

To measure the performance variation depending on *the types of data sets*, we use the XMark [30] benchmark data, which is a synthetic data set, and the TreeBank [21] data, which is a real data set. The XMark benchmark data consists of a large XML document. We use 10 MB, 100 MB, and 1 GB XML documents to measure the performance variation depending on *the database size*. In the XMark benchmark data, subtrees with a similar structure occur repeatedly at the same level. The TreeBank data consists of an XML document of 86 MB, in which elements are recursively nested in many levels.

To measure the performance variation depending on *the number of explicit authorizations*, we vary the ratio of the number of elements on which explicit authorizations are granted to the total number of elements in a data set from 0.01 to 100%. To determine an authorization object, we randomly pick 0.01–100% of elements in a data set. Here, we first pick the root element to make sure all the elements have an authorization—either explicit or implicit. Multiple explicit authorizations can be granted on the same element unless we try to grant both a positive authorization and a negative authorization for the same authorization type. To determine an authorization type, we randomly select one from five authorization types (e.g., read, write, update, create, and delete). We grant positive authorizations and negative authorizations with the ratio of 9:1. We also perform experiments with the ratio of 1:9.

For the CAM method, we treat implicit authorizations in our access control model as explicit ones that are identical to their parents' in CAM because, in CAM, every authorization has to be explicitly granted. Then, after compression, only those authorizations corresponding to explicit authorizations in our access control model remain and are indexed by CAM. Thus, authorizations indexed by CAM become the same as those indexed by our authorization index. To index authorizations, CAM constructs a trie consisting of the path strings of the elements having authorizations.

To measure the performance variation depending on *the types of queries*, we run simple queries of type $//e_A//e_B$ and complex queries of type $//e_A//e_B//e_C//e_D//e_E$ (a linear path expression) or $//e_A[.//e_B]//e_C$ (a branch path expression). Since queries of the same type have similar tendencies, we present the experimental results for the queries in Fig. 10, which show the representative tendency for each data set. Read authorizations are required to execute these queries.

dataset	simple queries	complex queries
XMark	//person//interest	(1) //site//open_auctions//open_auction//bidder//increase (2) //open_auctions[.//bidder]//seller
TreeBank	//NP//NN	//SBAR//S//VP//PP//NP

Fig. 10 XML queries used in the experiments

To compare the performance of our access control method with that of the SCA method, we reproduce the environmental setting used by Cho et al. [10] because, in SCA, instance-level authorizations are granted only according to security annotations. Cho et al. [10] have used two different settings for security annotations: (1) the instances of only three element types are permitted to have authorizations (called a *sparse* DTD) or (2) the instances of half of element types are permitted to have authorizations (called a *dense* DTD). Using these settings, they have issued the query *//site*[*.//open_auction/initial*]*/people//name* against the XMark benchmark data.

We conduct all the experiments on a SUN Ultra 60 workstation with 450 MHz CPU and 512 MB of main memory. We use the MLGF [32] as the multi-dimensional index for storing authorizations, and the page size for data and indexes is set to be 4,096 bytes. We use the SP-GiST [4] to implement the trie for the CAM method. Since this trie is constructed for each user [34], our authorization index is constructed for each user to make a fair comparison. For the SCA method, we store an authorization as an attribute of an element in the same way as done by Cho et al. [10]. For the top-down and bottom-up strategies, we store an authorization as a tuple in an authorization table in the same way as done by Rabitti et al. [27], and this table is consulted through the B^+-tree index to find the authorization for a given element.

6.2 Results of the experiments

6.2.1 Effects of the database size and the number of explicit authorizations

Figure 11 shows the wall clock time for the query *//person//interest* as the database size and the number of explicit authorizations are varied. These results indicate that Dynamic Predicate Filtering outperforms the existing methods by a large margin. Specifically, compared with the existing methods, Dynamic Predicate Filtering improves the performance by 1.8–12.1 times in Fig. 11a; by 2.0–70.7 times in Fig. 11b; and by 2.1–107.1 times in Fig. 11c. This margin increases as the database size gets

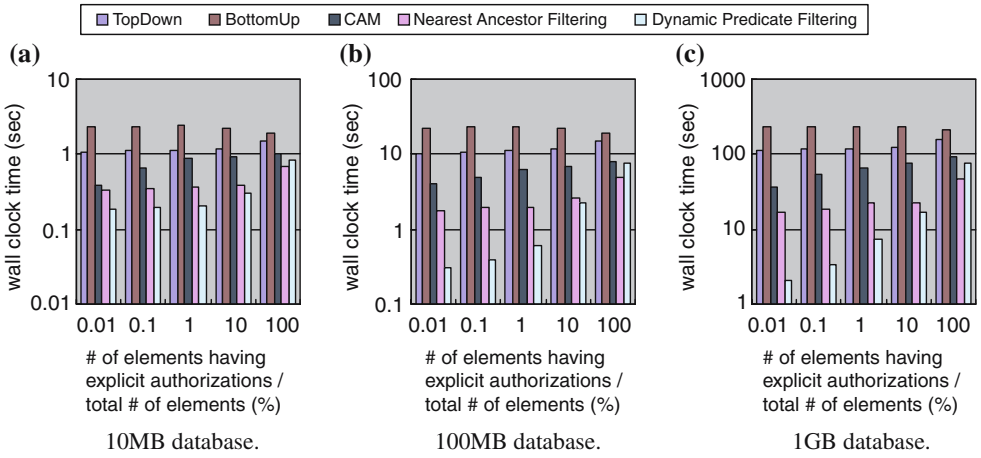

Fig. 11 The wall clock time for the query *//person//interest* against the XMark benchmark data

larger. The reason is that the number of elements that are excluded by a DP increases in a larger database since the number of elements in a subtree pruned at the same level increases. This effect is more marked when the number of explicit authorizations is small. The reason is that the number of elements that are excluded by a DP gets larger since the number of implicit authorizations that are overridden is small.

When explicit authorizations are granted on 100% of the elements, Dynamic Predicate Filtering shows a minor degradation (less than 1.6 times) in comparison with Nearest Ancestor Filtering since constructing DPs in Dynamic Predicate Filtering incurs overhead of searching for the nearest overriding authorization without any benefit. In XML databases, a large number of elements are granted implicit authorizations exploiting the data hierarchy. Granting explicit authorizations on every element is rare. Typically high compression ratio of the CAM method shown by Yu et al. [34] supports this argument. Besides, the experiments show that Dynamic Predicate Filtering outperforms Nearest Ancestor Filtering when explicit authorizations are granted on less than 70% of the elements. Figure 11 shows that the wall clock time of Nearest Ancestor Filtering slightly increases at 100%. It turns out that this increase occurred because the depth of the authorization index was increased by one.

Nearest Ancestor Filtering shows a performance similar to CAM with a minor improvement. The reason is that, as discussed in Sect. 6.1, CAM is a trie structure indexing authorizations after compression and is analogous to our authorization index. The minor improvement is due to the efficiency of the index structure and efficiency in finding the nearest ancestor. On the other hand, Dynamic Predicate Filtering improves the performance compared with CAM by 1.2–16.9 times. This is because Dynamic Predicate Filtering reduces query processing time by filtering unauthorized elements early on, while CAM does not.

6.2.2 Effects of query types

Figure 12 shows the wall clock time for a more complex query *//site//open_auctions//open_auction//bidder//increase*, which has a long path expression, as the database size and the number of explicit authorizations are varied. These results indicate that Dynamic Predicate Filtering outperforms the existing methods for a complex query with larger margins than for a simple query. The reason is that, if a subtree is pruned by a DP at a higher level, the cost for checking query conditions over that subtree can be much more reduced in a complex query. Compared with the existing methods, Dynamic Predicate Filtering improves the performance by 1.8–25.9 times in Fig. 12a; by 2.0–106.0 times in Fig. 12b; and by 2.0–165.8 times in Fig. 12c.

The wall clock time for a twig query *//open_auctions*[*.//bidder*]*//seller* shows a tendency similar to that of Fig. 11. For a twig query, since Dynamic Predicate Filtering is applied to each linear path, the overall improvement of Dynamic Predicate Filtering compared with other methods is the sum of the improvement obtained from these linear paths. In the XMark data of 100 MB, Dynamic Predicate Filtering improves the wall clock time by 1.1–17.4 times over CAM and by 0.9–7.6 times over Nearest Ancestor Filtering.

6.2.3 Effects of the depth of XML data

Figure 13 shows the wall clock times for the queries *//NP//NN* and *//SBAR//S//VP//PP//NP* against the Tree

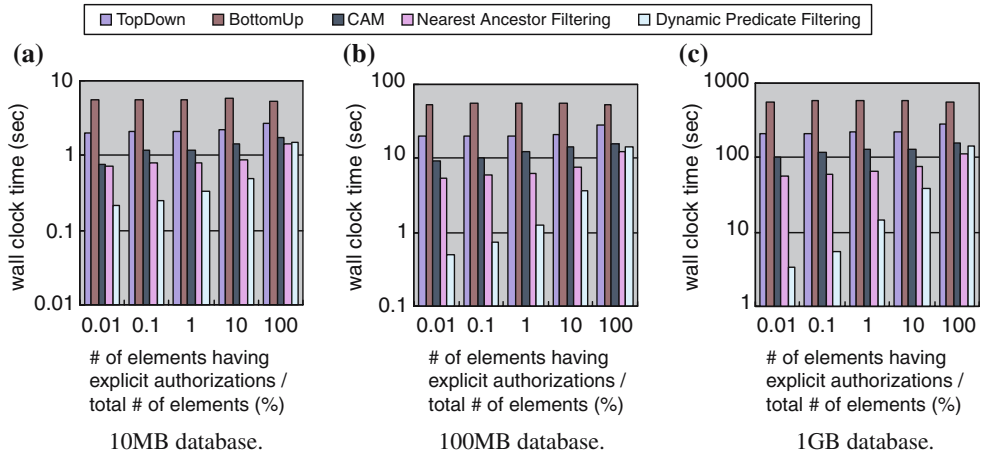

Fig. 12 The wall clock time for the query *//site//open_auctions//open_auction//bidder//increase* against the XMark benchmark data

Bank data of 86 MB as the number of explicit authorizations is varied. These results indicate that Dynamic Predicate Filtering outperforms the existing methods even more for XML data having many levels of recursively nested elements like the TreeBank data. Specifically, in Fig. 13b, Dynamic Predicate Filtering improves the performance by 5.9–216.3 times over the top-down strategy; by 9.2–492.9 times over the bottom-up strategy; by 1.1–31.1 times over CAM; and by 0.9–30.3 times over Nearest Ancestor Filtering. We note that the differences between Dynamic Predicate Filtering and the existing methods are much larger in Fig. 13 than in Figs. 11 and 12. The reason is that the number of elements that are excluded by a DP increases in XML data having many levels since the number of elements in a subtree pruned at a higher level increases. This result is natural since DPs computed in Lemma 3 inherently work well for hierarchical structures by virtue of *hierarchical filtering*.

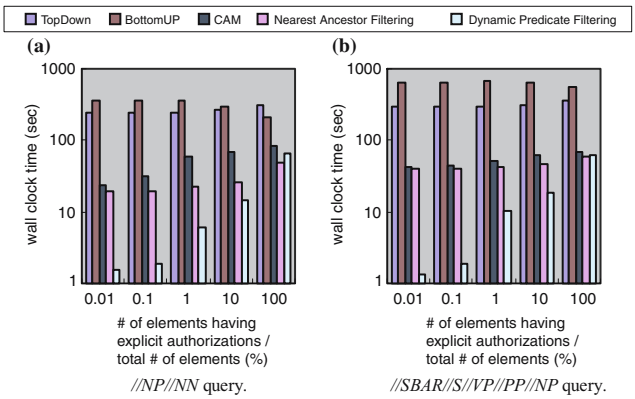

Fig. 13 The wall clock times for simple and complex queries against the TreeBank data (86 MB)

6.2.4 *Effects of the inference-blocking strategy*

Figure 14 shows the composition of the results in Fig. 12: the top portion of the bar represents the wall clock time overhead caused by the inference-blocking strategy. The wall clock times of the top-down and bottom-up strategies are shown to be insensitive to the inference-blocking strategy since these strategies access the ancestors including the qualifying instances to find the nearest ancestor authorization.

The wall clock times of Nearest Ancestor Filtering and Dynamic Predicate Filtering increase only slightly (less than 26.5%). It turns out that these increases are blunted by virtue of the buffering effect since most of the explicit authorizations on the qualifying instances are stored in the same page of the authorization index as the ones on result instances. This is possible because the start values of the qualifying instances tend to be close to those of result instances. These results indicate that adopting the inference-blocking strategy in our method does not incur much additional overhead.

6.2.5 *Comparison between our method and the SCA method*

Figure 15 shows the wall clock times of Dynamic Predicate Filtering and SCA for the experimental setting used by Cho et al. [10]. Dynamic Predicate Filtering outperforms SCA by 1.5–14.4 times in Fig. 15a and by 1.3–2.2 times in Fig. 15b despite that SCA exploits additional information (security annotations). Furthermore, the performance improvement increases as the database size gets larger. The reason for this advantage is that SCA reduces only the number of authorization checks performed during query processing, while Dynamic Predicate Filtering also reduces the number of elements

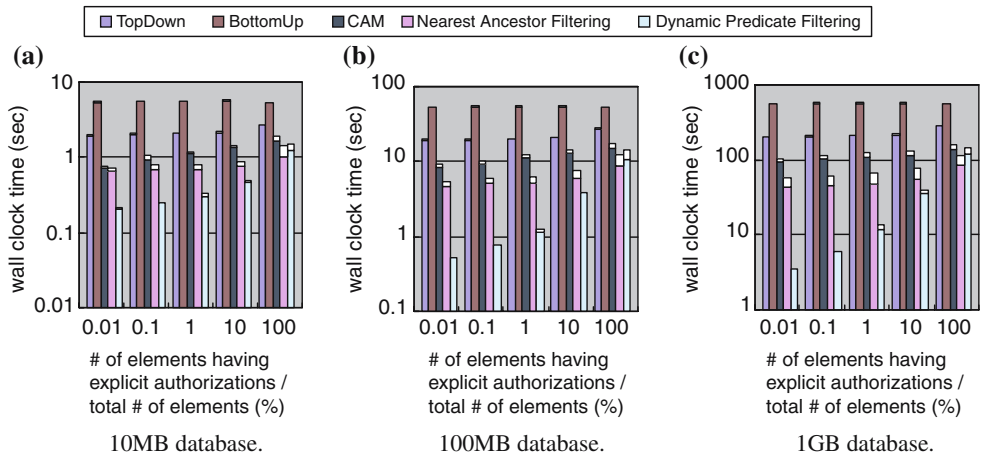

Fig. 14 The composition of the results in Fig. 12 (the top portion is the wall clock time overhead of the inference-blocking strategy)

accessed during query execution by filtering out unauthorized elements early on. This is exactly the advantage of integrating access control with query processing, which we propose in this paper. The difference between Dynamic Predicate Filtering and SCA is much larger in the sparse DTD than in the dense DTD because the number of elements excluded in Dynamic Predicate Filtering gets larger in the sparse DTD due to a smaller number of explicit authorizations. (In the sparse DTD, every *people* element is granted explicit ones. In the dense DTD, every *people*, *person*, and *open_auction* elements are granted explicit ones.)

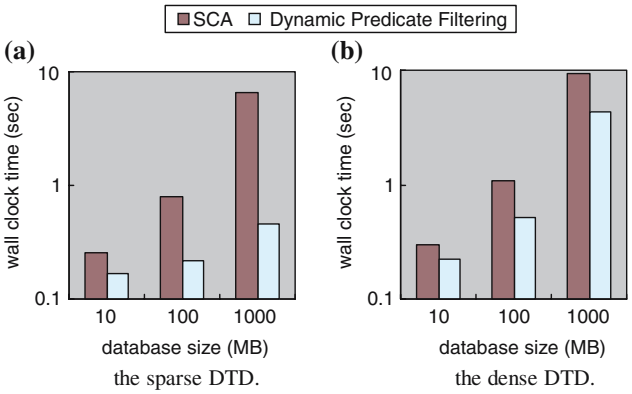

Fig. 15 The comparison between our method and SCA using the experimental setting by Cho et al.

6.2.6 Effects of the authorization types

In all the experiments stated above, we have repeated the test for the same queries with the ratio of positive and negative authorizations changed from 9:1 to 1:9. We have observed the performance of Dynamic Predicate Filtering only slightly improves by 10.8% on the average. The reason for this minor improvement is that a large proportion of the elements excluded are those on which an authorization has been granted for other than the authorization type of the query, and thus, negative authorizations do not much affect the performance.

7 Conclusions

In this paper, we have proposed an efficient access control mechanism tightly integrated with query processing for XML databases. To achieve this integration, we have presented a novel concept of the dynamic predicate, which represents a dynamically constructed condition during query execution. The DP is derived from instance-level authorizations and constrains accessibility of the elements. By inserting the DP into the query plan, we effectively integrate access control with query processing. Due to this integration, unauthorized elements are excluded in an early phase of query processing, thus significantly improving the performance of query processing with access control. We have implemented this access control mechanism as the algorithm Dynamic Predicate Filtering.

The primary factors that make this integration using the DP highly effective are instance-level authorizations and data hierarchies, both inherent in XML databases. First, the cost for checking instance-level authorizations is reduced significantly by using the DP, i.e., by representing the authorization information of a set of elements having an identical authorization with that of the element being currently accessed. Second, filtering unauthorized elements by the DP can be propagated

downward along the data hierarchies, which we call *hierarchical filtering*.

We have also minimized information inference by applying the inference-blocking strategy to the decision of the authorized query result. This strategy obviously reduces information inference compared with the basic strategy that checks authorizations only for the result instances. We have verified, through experiments, that enforcement of this strategy in our access control mechanism does not add much overhead by virtue of the buffering effect.

To show the efficiency of our access control mechanism, we have performed extensive experiments using synthetic and real data sets. Experimental results show that Dynamic Predicate Filtering improves the wall clock time by up to 31 times over CAM and by up to 14 times over SCA. These improvements indicate that the advantages of excluding beforehand unauthorized elements (and also authorization checks on them) in Dynamic Predicate Filtering are superior to those of indexing authorizations in CAM or those of reducing the number of authorization checks performed during run time in SCA. The results also show that the advantage of Dynamic Predicate Filtering becomes more prominent as (1) the database size, (2) the length of path expressions for queries, or (3) the depth of XML data increases. These results demonstrate the effectiveness of access control integrated with query processing.

Overall, we believe that we have provided an efficient mechanism for checking instance-level authorizations in databases with hierarchical structures. As a future work, we plan to extend our mechanism so as to handle larger fragments of XPath or more complex query languages such as XQuery.

Acknowledgements This work was supported by the Ministry of Science and Technology (MOST)/Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

Appendix A: Proof of Lemma 1

We show that the authorization obtained by Lemma 1 satisfies the two conditions in Definition 1. First, by the definition of the numbering scheme, $auth_{naa}$ is granted on the element e or one of its ancestor elements. Let us call that element e_{naa}. Thus, $auth_{naa}$ satisfies the condition (1). Second, let us assume that the authorization minimizing $|start(e) - start(auth)|$ is $auth_{naa}$ and that an explicit authorization $auth_{exp}$ exists on an element in the path between the element e and the element e_{naa}. Then, by the definition of the numbering scheme, $start(e_{naa}) < start(auth_{exp}) < start(e)$. (We note that $start(auth_{naa}) = start(e_{naa})$.) This contradicts the assumption. Therefore, no authorization exists on elements in the path between the element e and the element e_{naa}. Thus, $auth_{naa}$ satisfies the condition (2).

Appendix B: Proof of Lemma 2

We show that the authorization obtained by Lemma 2 satisfies the two conditions in Definition 2. First, by the definition of the numbering scheme, $auth_{noa}$ is granted on a descendant element e_{noa} of the element e_{naa} on which $auth_{naa}$ is granted. Thus, $auth_{noa}$ satisfies the condition (1). Second, let us assume that the authorization minimizing $|start(e) - start(auth)|$ is $auth_{noa}$ and that an explicit authorization $auth_{exp}$ exists, when traversed in preorder, on an element in the path between the element e and the element e_{noa}. Then, by the definition of the numbering scheme, $start(e) < start(auth_{exp}) < start(e_{noa})$. (We note that $start(auth_{noa}) = start(e_{noa})$.) This contradicts the assumption. Therefore, no authorization exists, when traversed in preorder, between the element e and the element e_{noa}. Thus, $auth_{noa}$ satisfies the condition (2).

Appendix C: Proof of Lemma 3

We show that any element whose start value belongs to the start interval defined in Lemma 3 has the same nearest ancestor authorization as that of the element e. In this proof, e_{naa} (or e_{noa}) denotes the element on which $auth_{naa}$ (or $auth_{noa}$) is granted. Before starting the proof, we point out that the start value increases in preorder [19]. First, we consider the case where $auth_{noa}$ exists. By Definition 2, no explicit authorization exists, when traversed in preorder, on elements in the path from the element e to the element right before e_{noa}. Then, the start values of those elements having no explicit authorization are formed as the interval $[start(e), start(e_{noa}))$ by the definition of the numbering scheme. Thus, any element in this start interval has $auth_{naa}$ as the nearest ancestor authorization. Second, we consider the case where $auth_{noa}$ does not exist. By Definition 2, no explicit authorization exists, when traversed in preorder, on elements in the path from the element e through the rightmost descendant e_{rd_naa} of the element e_{naa}. We note that $start(e_{rd\_naa})$ is immediately smaller than $end(e_{naa})$. Then, the start values of those elements having no explicit authorization are formed as the interval $[start(e), end(e_{naa}))$ by the definition of the numbering scheme. Thus, any element in this start interval has $auth_{naa}$ as the nearest ancestor authorization.

References

1. Aggarwal, G. et al.: Enabling privacy for the paranoids. In: Proceedings of 30th International Conference on Very Large Data Bases, Toronto, Canada, pp. 708–719, Aug./Sept. 2004
2. Agrawal, R. et al.: Hippocratic databases. In: Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, pp. 143–154, Aug. 2002
3. Al-Khalifa, S. et al.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of 18th International Conference on Data Engineering, San Jose, California, pp. 141–152, Feb. 2002
4. Aref, W.G., Ilyas, I.F.: SP-GiST: an extensible database index for supporting space partitioning trees. J. Intell. Inform. Syst. **17**(2–3), 215–240 (2001)
5. Bertino, E. et al.: Specifying and enforcing access control policies for XML document sources. World Wide Web J. **3**(3), 139–151 (2000)
6. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: Proceedings of 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, pp. 310–321, June 2002
7. Carminati, B., Ferrari, E.: Management of access control policies for XML document sources. Int. J. Inform. Sec. **1**(4), 236–260 (2003)
8. Carminati, B., Ferrari, E.: AC-XML documents: improving the performance of a web access control module. In: Proceedings of 10th ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden, pp. 67–76, June 2005
9. Chien, S.-Y. et al.: Efficient structural joins on indexed XML documents. In: Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, pp. 263–274, Aug. 2002
10. Cho, S. et al.: Optimizing the secure evaluation of twig queries. In: Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, pp. 490–501, Aug. 2002
11. Clark, J., DeRose, S.: XML path language (XPath) Version 1.0, W3C Recommendation, Nov. 1999
12. Damiani, E. et al.: A fine-grained access control system for XML documents. ACM Trans. Inform. Syst. Sec. **5**(2), 169–202 (2002)
13. Fan, W., Chan, C.Y., Garofalakis, M.N.: Secure XML querying with security views. In: Proceedings of 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, pp. 587–598, June 2004
14. Fundulaki, I., Marx, M.: Specifying access control policies for XML documents with XPath. In: Proceedings of 9th ACM Symposium on Access Control Models and Technologies, Yorktown Heights, New York, pp. 61–69, June 2004
15. Gaede, V., Gunther, O.: Multidimensional access methods. ACM Comput. Surveys **30**(2), 170–231 (1998)
16. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surveys **25**(2), 73–170 (1993)
17. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, pp. 47–57, June 1984
18. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2), 265–318 (1999)
19. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proceedings of 27th International Conference on Very Large Data Bases, Rome, Italy, pp. 361–370, Sept. 2001
20. Luo, B. et al.: QFilter: fine-grained run-time XML access control via NFA-based query rewriting. In: Proceedings of 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, pp. 543–552, Nov. 2004
21. Marcus, M.P., Marcinkiewicz, M.A., Santorini, B.: Building a large annotated corpus of English: The Penn Treebank. Comput. Linguist. **19**(2), 313–330 (1993)
22. Miklau, G., Suciu, D.: Containment and equivalence for a fragment of XPath. J. ACM **51**(1), 2–45 (2004)
23. Murata, M., Tozawa, A., Kudo, M.: XML access control using static analysis. In: Proceedings of 10th ACM Conference on Computer and Communications Security, Washingtion, DC, pp. 73–84, Oct. 2003
24. Information and Privacy Commissioner of Ontario, Intelligent Software Agents: Turning a Privacy Threat into a Privacy Protector, Apr. 1999
25. Information and Privacy Commissioner of Ontario, An Internet Privacy Primer: Assume Nothing, Aug. 2001
26. Qi, N., Kudo, M.: Access-condition-table-driven access control for XML database. In: Proceedings of 9th European Symposium on Research in Computer Security, French Riviera, France, pp. 17–32, Sept. 2004
27. Rabitti, F. et al.: A model of authorization for next-generation database systems. ACM Trans. Database Syst. **16**(1), 88–131 (1991)
28. Ramanan, P.: Covering indexes for XML queries: bisimulation – Simulation = Negation. In: Proceedings of 29th International Conference on Very Large Data Bases, Berlin, Germany, pp. 165–176, Sept. 2003
29. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surveys **16**(2), 187–260 (1984)
30. Schmidt, A.R. et al.: XMark: a benchmark for XML data management. In: Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, pp. 974–985, Aug. 2002
31. Seeger, B., Kriegel,H.-P.: The buddy-tree: an efficient and robust access method for spatial data base systems. In: Proceedings of 16th International Conference on Very Large Data Bases, Queensland, Australia, pp. 590–601, Aug. 1990
32. Whang, K.-Y., Krishnamurthy, R.: The multilevel grid file—a dynamic hierarchical multidimensional file structure. In: Proceedings of International Conference on Database Systems for Advanced Applications, Tokyo, Japan, pp. 449–459, Apr. 1991
33. Wu,Y., Patel, J.M., Jagadish H.V.: Structural join order selection for XML query optimization. In: Proceedings of 19th International Conference on Data Engineering, Bangalore, India, pp. 443–454, Mar. 2003
34. Yu, T. et al.: A compressed accessibility map for XML. ACM Trans. Database Syst. **29**(2), 363–402 (2004)