A New Optimistic Concurrency Control in Valid XML*

CHANGWOO BYUN+, ILKOOK YUN AND SEOG PARK
+Department of Computer Systems and Engineering
Inha Technical College
Incheon, 402-752 South Korea
Department of Computer Science and Engineering
Sogang University
Seoul, 121-742 South Korea

When numerous users and applications work concurrently on Extensible Markup Language (XML) documents, the transaction management of these XML documents becomes an important issue. The detection of conflicts between operations becomes vital in the query optimization for a query compiler and concurrency for a transaction manager. XML data require tailor-made conflict detection mechanisms since traditional solutions for the relational model are inadequate. Such mechanisms need to take full advantage of the hierarchical structure of semi-structured data. In this paper, we will solve a fundamental problem of the concurrency control for valid XML documents, which is the detection of conflicts between operations. We will formalize READ-UPDATE and UPDATE-UPDATE conflicts in valid XML documents, and propose effective algorithms to detect such conflict when the update operations are specified using XPath expressions. Furthermore, we will show that our proposed optimistic concurrency control scheme is a solution to the phantom and the pseudo-conflict problems.

Keywords: valid XML, transaction management, locking protocol, optimistic concurrency control, conflict, conflict detection

1. INTRODUCTION

Since the Extensible Markup Language (XML) [1] has become a standard for the distribution and sharing of information, there is a need for effective means to manage persistent XML data as a database.

Recently, several works [3-5] have been conducted in updating XML documents on well-formed XML documents. The approaches discussed in these studies assume that only a single user is accessing an XML database. With the advent of native and XML-enabled databases [6-8], it is important for multiple users to be able to modify the databases simultaneously.

Several tree-based locking protocols [14] were proposed. Although the tree-based locking has been known to reduce the locking overhead for releasing unnecessary locks, it is impossible for us to identify the elements to lock before traversing the XML tree in lock conflict tests. Moreover, the tree-based locking cannot be free from a deadlock and a phantom problem. The phantom problem is solved by using multigranularity locking protocols [15, 16, 18, 20, 21, 23]. Even though the multigranularity locking protocol ef-

fectively preserves serializability from the phantom problem, it may fail to achieve a high concurrency due to pseudo-conflicts in the case of XML databases. Such conflicts arise from inserting new elements into or deleting child elements from intention lock elements while other transactions perform on an XML database. Z. Sardar and B. Kemme [22] proposed two snapshot-based concurrency control mechanisms that avoid locking. Although the snapshot-based concurrency control scheme never delays read- only transactions, it does not guarantee serializability, and it could lead to a lost update problem.

The optimistic concurrency control provides non-blocking and deadlock freedom. However, analytical studies [10, 11] and simulations [12, 13] revealed that the locking approach performed better than the pure optimistic concurrency control. The main reason for this is the risk of a high abort rate or even cyclic restarts.

This paper will consider a new optimistic concurrency control scheme in valid XML. Aborts are bound to occur under an optimistic scheme. However, if aborts can be detected early, the optimistic concurrency control can help avoid some unnecessary "wasted work." In addition, restarts are designed to have low fetch costs and can thus be executed very quickly. We present effective detection mechanisms of conflicts between operations on valid XML documents which avoid locking.

The main contributions of this paper are the following:

- READ-UPDATE conflicts as well as UPDATE-UPDATE conflicts in valid XML documents are defined.
- Effective detection algorithms of conflicts in valid XML documents are proposed.
- Propositions on detection algorithms independent from the underlying XML database engine, which could be built on any XML DBMS or work as a stand-alone service are identified.
- Proposed detection algorithms are supported for an optimistic concurrency control protocol that avoids locking.

The remainder of the paper is organized as follows: Section 2 briefly reviews current work on XML concurrency control and introduces our motivation. In section 3, new schema information is presented to describe the semantics of conflicts in valid XML documents, and effective detection algorithms of conflicts are also provided. Then the proposed detection algorithms to handle predicates are extended. In section 4, comparisons of the performance results reveal the strengths of our optimistic concurrency control scheme. Section 5 discusses extending the results to make DELETE-UPDATE conflicts less strict. Finally, section 6 presents possible future research works. Section 7 summarizes the significant findings discussed in this paper.

2. RELATED WORK

2.1 Lock-based Concurrency Control Scheme

As native XML database systems [6-8] become increasingly popular, it is essential for multiple users to concurrently access the same documents that are based on a fine-granularity concurrency control. Due to the nested hierarchical nature of the XML data model, the concurrency schemes are based on hierarchical locking protocols.
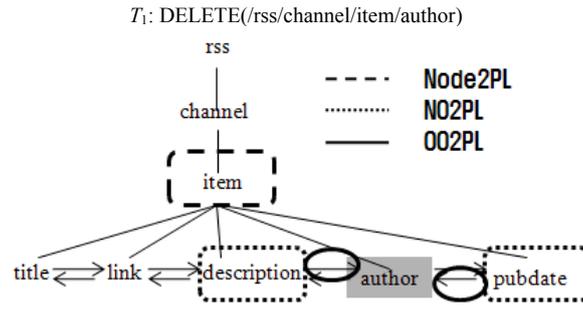
T_1: DELETE(/rss/channel/item/author)

Fig. 1. M-Locks held by T_1 for four lock-based protocols.

In [14], four lock-based protocols which can ensure serialization have been proposed, namely, Doc2PL, Node2PL, NO2PL, and OO2PL. The Doc2PL protocol locks at the document level. Fig. 1 shows on which items the four lock-based protocols acquire locks. The Node2PL protocol acquires locks for parent nodes (in this case, *item* node). The NO2PL acquires locks for all nodes whose pointers are traversed or modified (in this case, *description* and *pubdate* nodes). The OO2PL protocol locks pointers.

Those locking schemes have been known to reduce the locking overhead for releasing unnecessary locks in the course of processing a transaction while ensuring serializability.

Unfortunately, the tree-based locking protocols traverse the XML tree in lock conflict tests to identify the elements to lock. Moreover, the load of the lock manager increases in proportion to the number of locks that we hold, which will end up making the transaction throughput worse. In addition, the tree-based locking protocols cannot be free from a deadlock and a phantom problem.

Fig. 2 shows an INSERT of phantom element on the *item* node. In order to retrieve all elements contained in *rss/channel/item*, T_1 first has to set R lock on *item* and then set S lock on *title*, *link* and *description* nodes, respectively. T_1 release its R lock on *item* according to the rules of the tree-based locking protocol since it successfully acquires a lock on its child elements. Now, suppose that at the same time T_2 does request an X lock

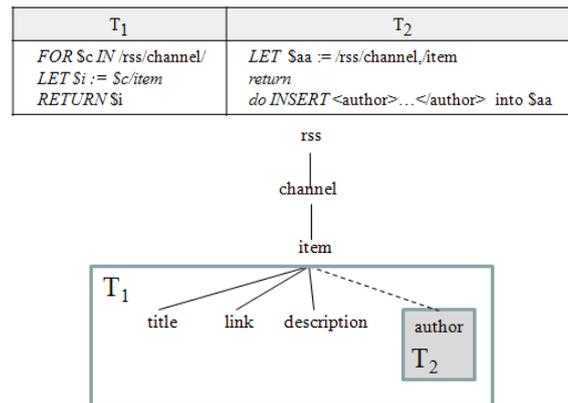| T_1	T_2
FOR $c IN /rss/channel/ *LET $i := $c/item* *RETURN $i*	*LET $aa := /rss/channel,/item* *return* *do INSERT <author>…</author> into $aa*

Fig. 2. INSERT of phantom element on the *item* node.

on *item*. This lock request can be granted immediately, and the insertion of a new element, *author*, into the *item* is allowed. If T_1 tries to retrieve collections of elements contained in *item* once more, the second results include a phantom element, *author*.

The phantom problem can be solved in existing commercial data management systems by using multigranularity locking protocols. In order to manage subtree-level locks on a hierarchical data structure, the multigranularity locking scheme involves a shared lock, an exclusive lock, and intention locks which are composed of intention shared locks and intention exclusive locks. DGLOCK [15] has been a protocol for semantic locking on the DataGuide, a characteristic of which is that it locks the path index instead of the document itself. For a simple path query without predicates, two path-locking schemes [16], namely, Path Lock Propagation and Path Lock Satisfiability can be used. T. L. Saito and S. Morishita [20, 21] have developed a high concurrent XML database and proposed an efficient index for the detection of conflicts. Even though the multigranularity locking protocol effectively preserves serializability from phantom problems, it may fail to achieve a high concurrency due to pseudo-conflicts in the case of XML databases. Such conflicts arise from inserting new elements into or deleting child elements from intention lock elements while other transactions perform an XML database.

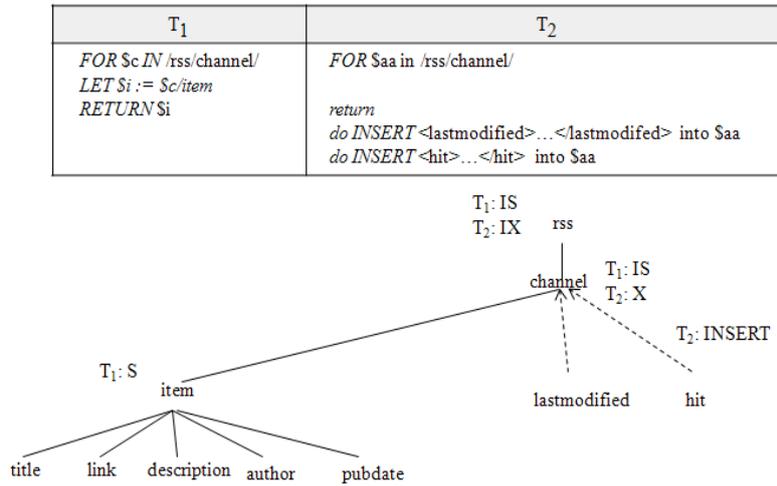| T_1	T_2
FOR c *IN* /rss/channel/ *LET* $i := $c/item$ *RETURN* i	*FOR* aa *in* /rss/channel/ *return* *do INSERT* <lastmodified>...</lastmodifed> into aa *do INSERT* <hit>...</hit> into aa

Fig. 3. A pseudo-conflict on the *channel* element.

Fig. 3 shows a pseudo-conflict phenomenon in XML tree. T_2 is assumed to insert *lastmodified* and *hit* nodes into the *channel* node, while T_1 is retrieving XML data from location given by an XPath expression, *rss/channel/item*. T_1 has to place an *IS* lock on the *channel* and *rss* nodes, and then set an *S* lock on the *item* node. Now, suppose that at the same time, T_2 requests an *X* lock on the *channel* node in order to insert the *lastmodified* and *hit* context nodes by T_2. For ensuring serializability, this lock request should not be granted until T_1 releases the *IS* lock on *item*. However, this schedule misses an opportunity for enhancing concurrency due to the strict scheduling rules of the multigranularity locking protocol.

In addition, most of the current research works based on locking have proposed

many types of locks such as scanning, navigational and declarative read operations, as well as arbitrary element/subtree insertions/deletions/replacements. M. Haustein and T. Härder [18, 23] have used eight lock modes in an XML Transaction Coordinator (XTC) system, which is a prototype database system supporting transaction-protected native XML document processing. Such a system has much more complex conversion rules. When applied to the context node, the locks on its entire ancestor path must also be converted.

2.2 Optimistic Concurrency Control Scheme

Z. Sardar and B. Kemme [22] have proposed two snapshot-based concurrency control mechanisms that avoid locking. OptiX is a variation of an optimistic concurrency control adjusted to use snapshots and work on XML data. SnaX provides the snapshot isolation. The snapshot-based concurrency control scheme is a multi-version concurrency control algorithm. A transaction T_i executing under the snapshot-based concurrency control scheme conceptually reads data from the committed state of the database as of time $start(T_i)$, the time when T_i started. Hence, T_i basically reads a snapshot of the data at that particular time. This snapshot contains all writes of committed transaction and no writes of uncommitted transactions. T_i then holds the results its own writes in local memory store until it commits at time $commit(T_i)$. Although the snapshot-based concurrency control scheme never delays read-only transactions, there are some inherent weaknesses in snapshot isolation. It does not guarantee serializability and it could lead to lost updates.

2.3 Motivation

M. Raghavachari and O. Shmueli [19] presented formalizations of node conflicts and tree conflicts against READ-INSERT and READ-DELETE conflicts. They were able to show that conflict detection is NP-complete when operations use the child and descendant axes exclusively and thus proposed efficient polynomial algorithms for update conflict detection when the patterns do not use branching (for example, predicates). However, Raghavachari and Shmueli's research focused on well-formed XML documents only. The complexity of conflicts becomes less severe when schema information, such as DTD, is available. Knowledge of the DTD can reduce the number of conflicts and hence increase the degree of cooperation.

We give a brief description of the basic optimistic concurrency control scheme as proposed in [9]. A transaction consists of three phases: a *read phase*, a *validation phase*, and a *write phase*. In the *read phase*, the required objects are read from the database, and write operations are performed on local copies of the database objects. In the *validation phase*, a check for serializability is performed. If validation is successful, the objects modified by the transaction are written into the database (*write phase*); otherwise, the transaction is restarted. The important part of an optimistic concurrency control is the validation phase, that is, the scheduler tests whether or not a transaction's execution has been "correct" in the sense of conflict serializability. To elucidate, a transaction T_i passes validation if for all concurrent transactions T_j that are already validated, $WS(T_j) \cap RS(T_i) = \varnothing$. The read set $RS(T_i)$ denotes the nodes read by T_i, and the write set $WS(T_j)$ denotes the nodes written by T_j.

This paper will consider a new optimistic concurrency control in valid XML. The optimistic concurrency control provides non-blocking and deadlock freedom. However, previous analytical studies [10, 11] and simulations [12, 13] revealed that the locking approach performed better than the pure optimistic concurrency control. The main reason is the risk of a high abort rate or even cyclic restarts. Aborts are bound to occur under an optimistic scheme. Thus, if aborts can be detected early (that is, before *read phase* and *validation phase*), the optimistic concurrency control can help avoid some unnecessary "wasted work." In addition, restarts are designed to have low fetch costs and can be executed very quickly.

3. ALGORITHM OF EFFECTIVE CONFLICT DETECTION

3.1 Schema Information

XML [1] has two types of validation: well-formedness and grammatical validity. A well-formed document must follow the XML rules for a physical document's structure and syntax. A valid XML document is a well-formed document that also confirms to the stricter rules specified in a document type definition (DTD) or an XML Schema. The DTD describes a document's structure, specifies which element types are allowed, and defines the properties for each element. If a document has a DTD and the document satisfies the DTD, then the document is considered valid. If it does not, then it is invalid.

An element declaration may contain a sequence content model (symbol: ,), a choice content model (symbol: |), or cardinality (?: zero or one of the elements, *: zero or more of the elements, and +: one or more of the elements). Choices, sequences, and suffixes can be combined in arbitrarily complex fashions to essentially describe any reasonable content model. The item enclosed in parentheses can also be nested inside other choices or sequences.

We show a DTD in Fig. 4, which we consider as a running example in our paper.

```
<!ELEMENT rss (channel)*>
<!ELEMENT channel (item*, title, link, description, lastmodified,(author | editor)?, hit, rank?)>
<!ELEMENT item (title, link, description, author, pubdate)>
<!ELEMENT title #PCDATA>
<!ELEMENT link #PCDATA>
<!ELEMENT description #PCDATA>
<!ELEMENT lastmodified #PCDATA>
<!ELEMENT author #PCDATA>
<!ELEMENT editor #PCDATA>
<!ELEMENT hit #PCDATA>
<!ELEMENT rank #PCDATA>
<!ELEMENT pubdate #PCDATA>
```
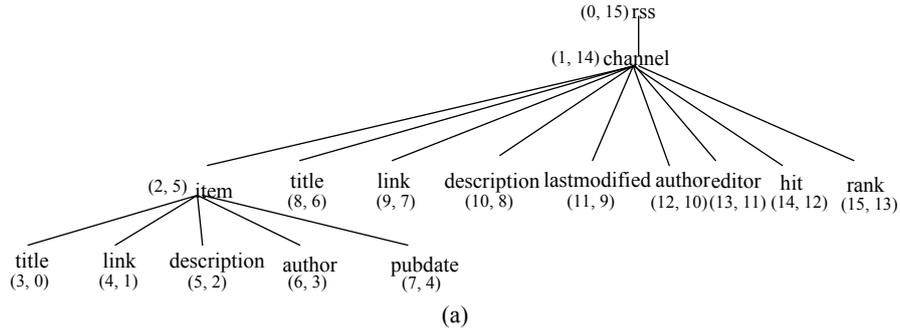
Fig. 4. A sample DTD.

Fig. 5 (a) shows that the nodes of the DTD tree are assigned with PRE(order) and POST(order) ranks of the DTD shown in Fig. 4. Fig. 5 (b) shows the actual relational DTD representation. PRE is a preorder rank of the DTD tree, which simultaneously serves as a node identifier. LEVEL refers to the distance from the tree's root. SIZE is the

(a)

Tag-Name	Pre	size	level	post	Tag-Name	Pre	size	level	post
rss	0	15	0	15	title	8	0	2	6
channel	1	14	1	14	link	9	0	2	7
item	2	5	2	5	description	10	0	2	8
title	3	0	3	0	lastmodified	11	0	2	9
link	4	0	3	1	author	12	0	2	10
description	5	0	3	2	editor	13	0	2	11
author	6	0	3	3	hit	14	0	2	12
pubdate	7	0	3	4	rank	15	0	2	13

(b)

Fig. 5. (a) DTD PRE/POST structure of Fig. 4; (b) Relational storage of (a).

number of nodes n the subtree below any node. This PRE/SIZE/LEVEL encoding is equivalent to the PRE/POST since POST = PRE + SIZE – LEVEL [24]. For example, the preorder rank of a node *item* (*rss/channel/item*) is 2, SIZE 5, and LEVEL 2. Thus, the postorder rank of *item* is 5. This is called the PRE/POST Structure (PPS) of the DTD tree. We use this information to detect conflicts of operations in section 3.3.

3.2 Conflict in XML

Research about the updates of the XML document has been processed in syntax definition and resolution of semantics problems which occur during update processes and inconsistency between the schema and XML instance. For instance, Igor Tatarinov *et al.* [3] introduced INSERT, DELETE, REPLACE, and RENAME operations. The XML Query Working Group introduces the requirements for the XQuery Update Facility [4] and specifies usage scenarios for the XQuery Update Facility [5].

This paper considers INSERT, DELETE, and REPLACE operations. The types of INSERT, DELETE, and REPLACE operations are referred from [3]. The update operations take a set of parameters (*XPE, child, new_content*). *XPE* is an XPath expression; valid types for *child* and *new_content* include PCDATA, attribute, and element. Their meanings are redefined as follows:

- INSERT(*XPE, child*): INSERT allows the insertion of new data (*child*) as a child of *XPE*.
- DELETE(*XPE*): DELETE allows the deletion of all *children* of *XPE* as well as *itself*.

- REPLACE (*XPE*, *new_content*): REPLACE allows replacement of all children of *XPE* with a *new_content* one.

Table 1 shows the relation between XQuery Update Facility and related operations in this paper.

Table 1. Update operations corresponding to XQuery update facility.

XQuery Update Facility	Operations
FOR $c *IN* /rss/channel/ *LET* $i := $c/item *RETURN* $i	READ($i)

LET $aa := in /rss/channel
      $bb := $aa/lastmodified
      $cc := $bb/item[2]
 RETURN
   do REPLACE $bb *WITH*
 \<lastmodified> 20070301 10:00
 \</lastmodified>,
 do DELETE $cc
 do INSERT \<item> … \</item> *into* $aa
	REPLACE($bb/lastmodified, "20070301 10:00")
	
	 DELETE ($cc)
	 INSERT($aa/item, " …")

In [25], READ operation represented by an XPath expression declares the query requirement by identifying the node of interest via the path from the root of the document to the elements which serve as the root of the subtrees to be returned. The UPDATE operation, which is also represented by an XPath expression, declares the requirement by updating the subtrees to be returned by the XPath expression.

Conflicts are defined between the READ-UPDATE operations or UPDATE-UPDATE operations on XML data.

An XML document is modeled as a labeled tree, with each node labeled with a symbol from an infinite alphabet Σ. The set of all trees over Σ will be denoted as T_Σ. The following are definitions of conflicts for both a READ operation R_i and an UPDATE operation U_j.

Definition 1 [READ-UPDATE conflict] R_i has a conflict with U_j if there exists $t \in T_\Sigma$, $R_i(U_j(t)) \neq R_i(t)$.

The semantics of $R_i(U_j(t)) \neq R_i(t)$ means that if the scope of a READ operation includes that of an UPDATE operation, then the two operations are a READ-UPDATE conflict.

In Fig. 6 (a), let the roots of the subtrees of R_i and U_j (in this case, DELETE) be x and y, respectively. In this case, $R_i(U_j(t))$ excludes the gray part of the tree t. However, $R_i(t)$ includes the gray part of the tree t. Thus, $R_i(U_j(t)) \neq R_i(t)$. Also, if the roots of the subtrees of R_i and U_j (DELETE) are z and y, respectively, then $R_i(U_j(t)) \neq R_i(t)$ in a similar way. In Fig. 6 (b), let the roots of the subtrees of R_i and U_j (in this case, INSERT or REPLACE) be x and y, respectively. In this case, $R_i(U_j(t))$ includes the bright gray (that
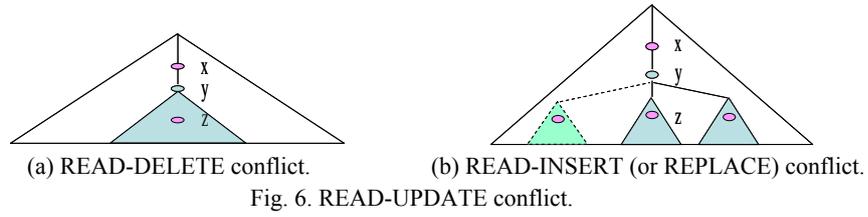
(a) READ-DELETE conflict.   (b) READ-INSERT (or REPLACE) conflict.

Fig. 6. READ-UPDATE conflict.

is, dotted line) part of the tree t which are modified subtrees by U_j. However, $R_i(t)$ excludes the bright gray part of the tree t. Thus, $R_i(U_j(t)) \neq R_i(t)$. In addition, if the root of the subtrees of a READ operation is of a descendant relation (including child relation) against that of an UPDATE operation, then the answer of the detection of conflict is "yes."

Up to this point, only READ-UPDATE conflicts have been discussed. UPDATE-UPDATE conflicts are of interest as well. The following defines a conflict between an UPDATE operation U_i and an UPDATE operation U_j.

Definition 2 [UPDATE-UPDATE conflict] U_i has a conflict with U_j if there exists $t \in T_\Sigma$, $U_i(U_j(t)) \neq U_j(U_i(t))$.

The semantics of $U_i(U_j(t)) \neq U_j(U_i(t))$ means that if the scope of U_i includes all parts or some parts of that of U_j, or vice versa, then the two operations are an UPDATE-UPDATE conflict.

In Fig. 7, let the roots of the subtrees of U_i and U_j be x and y, respectively. In this case, $U_i(U_j(t))$ modifies the modified subtrees of U_j in the tree t. On the other hand, $U_j(U_i(t))$ modifies the modified subtrees of U_i. Thus, $U_i(U_j(t)) \neq U_j(U_i(t))$.
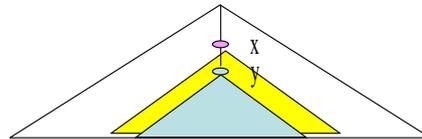

Fig. 7. UPDATE-UPDATE conflict.

3.3 Conflict Detection Algorithm

At this point, two conflicts have been defined, namely, READ-UPDATE and UPDATE-UPDATE conflicts. Effective detection algorithms of conflicts are then proposed. The basic notion pursued is one of the encoding trees. The proposed algorithm uses the PRE(order) and POST(order) ranks of the root of subtrees of operations to map each (PRE, POST) onto a two-dimensional plane, for instance, the PRE/POST plane. The detection of conflicts is then reduced to process a part of the regions in this PRE/POST plane as shown in Fig. 9.

Before proceeding with a detailed description of the PRE/POST plane, the target node should be defined. In section 3.2, READ or UPDATE operations were represented by an XPath expression. The root of subtrees to be returned is called the target node. The

(PRE, POST) value(s) of the target node may be obtained through the PRE/POST structure shown in Fig. 5.

Definition 3 [Target Node]   A target node of a given XPath is the last node except for the predicates.

The following are some operation examples:

O_1: READ(/rss/channel/item/description), the target node is a *description* node, and the (PRE, POST) value of the target node is (5, 2).

O_2: REPLACE(/rss/channel/lastmodified, "20070301 11:30"), the target node is a *last modified* node, and the (PRE, POST) value of the target node is (11, 9).

O_3: INSERT(/rss/channel/item, "xxxx"), the target node is an *item* node, and the (PRE, POST) value of the target node is (2, 5).

O_4: READ(/rss//description), the target node is a *description* node, and the (PRE, POST) values of the target node are (5, 2) and (10, 8).

O_5: DELETE(/rss/channel/item [1]), the target node is an *item* node, and the (PRE, POST) value of the target node is (2, 5).

O_6: REPLACE(/rss/channel/author, "chang"), the target node is an *author* node, and the (PRE, POST) value of the target node is (12, 10).

Note that there may also be more than two (PRE, POST) pairs of the target node of an operation. Look at operation O_4, for example. The target node of O_4 is the description node. Thus, the (PRE, POST) pairs of the name are (5, 2) and (10, 8). However, all (PRE, POST) pairs may not be the (PRE, POST) pairs of an operation. Again, look at operation O_1. The target node of O_1 is the description node. Although the (PRE, POST) pairs of the description are (5, 2) and (10, 8), only (5, 2) is a child node of (2, 5) of the *item* node. Its main idea is that the preorder (postorder) value of each node of an operation is less (greater) than that of the target node of the operation. Fig. 8 shows the pruning algorithm that eliminates the unsuitable (PRE, POST) pairs of an operation.

Input: an XPath expression of an operation
Output: suitable (PRE, POST) values of the target node of the operation
BEGIN
1. for each (Pr_{tn}, Po_{tn}) value of the target node of the operation
2. {for (Pr_{step}, Po_{step}) value of each node of the operation
3.   if $(!(Pr_{step} < Pr_{tn}$ and $Po_{step} > Po_{tn}))$
4.   break;
5.   suitable (PRE, POST) set $\leftarrow (Pr_{tn}, Po_{tn})$
6. }
END

Fig. 8. The Prune-TNs algorithm.

The following describes the semantics of the PRE/POST plane as shown in Fig. 9. Let the (Pr_{Oi}, Po_{Oi}) and (Pr_{Oj}, Po_{Oj}) pairs be the (PRE, POST) values of two operations O_i, and O_j, respectively.
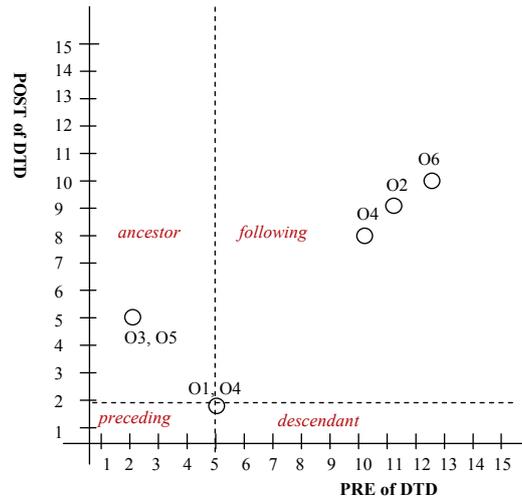
Fig. 9. PRE/POST plane of operations.

Definition 4 [upper-right partition: O_i is a *FOLLOWING* relation against O_j] *FOL-LOWING* means that the preorder and postorder values of an operation O_i are greater than those of another operation O_j subject to $Pr_{Oi} > Pr_{Oj}$, $Po_{Oi} > Po_{Oj}$.

Definition 5 [lower-left partition: O_i is a *PRECEDING* relation against O_j] *PRE-CEDING* means that the preorder and postorder values of an operation O_i are less than those of another operation O_j subject to $Pr_{Oi} < Pr_{Oj}$, $Po_{Oi} < Po_{Oj}$.

Definition 6 [upper-left partition: O_i is an *ANCESTOR* relation against O_j] *AN-CESTOR* (including ***PARENT***) means that the preorder (postorder) value of an operation O_i is less (greater) than that of an operation O_j, respectively, subject to $Pr_{Oi} < Pr_{Oj}$, $Po_{Oi} > Po_{Oj}$.

Definition 7 [lower-right partition: O_i is a *DESCENDANT* relation against O_j] *DE-SCENDANT* (including ***CHILD***) means that the preorder (postorder) value of an operation O_i is greater (less) than that of an operation O_j, respectively, subject to $Pr_{Oi} > Pr_{Oj}$, $Po_{Oi} < Po_{Oj}$.

Definition 8 [O_i is a *SELF* relation against O_j] *SELF* means that the (PRE, POST) pair of an operation O_i is equal to that of an operation O_j subject to $Pr_{Oi} = Pr_{Oj}$, $Po_{Oi} = Po_{Oj}$.

As shown in Fig. 9, O_3 and O_5 are ***ANCESTOR*** relations against O_1, and O_2 and O_6 are ***FOLLOWING*** relations against O_1. Based on Definitions 4 to 8, if two operations are not READ operations, conflicts can be detected between them.

Detection of READ-UPDATE Conflict Let (Pr_{Ri}, Po_{Ri}) and (Pr_{Uj}, Po_{Uj}) be (*PRE, POST*) values of the target nodes of R_i and U_j, respectively. R_i has a conflict with U_j, which is subject to

$$Pr_{Ri} < Pr_{Uj}, Po_{Ri} > Po_{Uj} \quad (\textbf{ANCESTOR}) \tag{1}$$
$$Pr_{Ri} > Pr_{Uj}, Po_{Ri} < Po_{Uj} \quad (\textbf{DESCENDANT}) \tag{2}$$
$$Pr_{Ri} = Pr_{Uj}, Po_{Ri} = Po_{Uj} \quad (\textbf{SELF}). \tag{3}$$

Eqs. (1) or (3) means that the target node of R_i is an ***ANCESTOR*** (***SELF***) relation against that of U_j. In this case, R_i reads all the modified parts of U_j. Eq. (2) means that the target node of R_i is a ***DESCENDANT*** relation against that of U_j. In this case, R_i reads some modified parts of U_j. Therefore, R_i and U_j are a READ-UPDATE conflict.

As shown in Fig. 9, O_1 has a conflict with O_3 and O_5 because O_1 is a ***DESCEN-DANT*** relation against O_3 and O_5. However, O_1 has no conflicts with O_2 and O_6 because O_1 is a ***PRECEDING*** relation against O_2 and O_6. Since O_1 and O_4 are READ operations, they have no conflict with each other.

Detection of UPDATE-UPDATE Conflict   Let (Pr_{Ui}, Po_{Ui}) and (Pr_{Uj}, Po_{Uj}) be (*PRE, POST*) values of the target nodes of an UPDATE operation U_i and an UPDATE operation U_j, respectively. U_i has a conflict with U_j, which is subject to

$$Pr_{Ui} < Pr_{Uj}, Po_{Ui} > Po_{Uj} \quad (\text{ANCESTOR}) \tag{4}$$
$$Pr_{Ui} > Pr_{Uj}, Po_{Ui} < Po_{Uj} \quad (\text{DESCENDANT}) \tag{5}$$
$$Pr_{Ui} = Pr_{Uj}, Po_{Ui} = Po_{Uj} \quad (\text{SELF}). \tag{6}$$

The semantics of Eqs. (4)-(6) is similar to that of READ-UPDATE conflict.

As shown in Fig. 9, O_3 has a conflict with O_5 because O_3 is a ***SELF*** relation against O_5. However, O_3 has no conflicts with O_2 and O_6 because O_3 is a ***PRECEDING*** relation against O_2 and O_6.

Theorem 1   The ***DETECTION*** algorithms detect all *READ-UPDATE* or *UPDATE-UP-DATE* conflicts when two operations are limited to *XPath* expressions without predicates.

Proof: The ***DETECTION*** of Conflicts algorithm abstracts the ***ANCESTOR*** (Definition 6), ***DESCENDANT*** (Definition 7) and ***SELF*** operations (Definition 8) against an UP-DATE operation and accordingly, the ***FOLLOWING*** and ***PRECEDING*** operations against the UPDATE operation could be ignored since these operations do not have any conflicts with the UPDATE operation.

Let (Pr_{TN}, Po_{TN}) be the (*PRE, POST*) value of the target node of an UPDATE operation U. Suppose that *SN* is any node in the sub-nodes of the U. Then the (*PRE, POST*) value of *SN* is as follows:

$$Pr_{TN} < Pr_{SN} < Pr_{TN} + \text{SIZE}(TN),$$
$$Po_{last\_sibling\_node\_of\_TN} < Po_{SN} < Po_{TN}. \tag{E1}$$

Suppose that *FN* is any node in the following nodes of the U. Then the (PRE, POST) value of *FN* is as follows:

$$Pr_{TN} < Pr_{FN}, Po_{TN} < Po_{FN}. \tag{E2}$$

In particular, the (PRE, POST) value of the first following node (*first_FN*) of *U* is as follows:

$$Pr_{first\_FN} = Pr_{TN} + \text{SIZE}(TN) + 1. \tag{E3}$$

Thus, by Eqs. (E1) and (E2), $Po_{SN} < Po_{TN} < Po_{FN}$, and by Eqs. (E1) and (E3), we obtain Eq. (E4).

$$Pr_{SN} < Pr_{TN} + \text{SIZE}(TN) < Pr_{first\_FN} = Pr_{TN} + \text{SIZE}(TN) + 1 \tag{E4}$$

Eq. (E4) means that the preorder value of any node (*SN*) in the sub-nodes of *U* is less than that of the first following node (*first_FN*) of the *U*. Therefore, $Pr_{SN} < Pr_{FN}$. As a result, $Pr_{SN} < Pr_{FN}$ and $Po_{SN} < Po_{FN}$ can be obtained. This means that the **FOLLOWING** operations have no conflicts with *U*, as will be proven. The same is the case with **PRE-CEDING** operations and hence the proof.

However, the definition of these conflicts is not completely straightforward. There are some exceptions in UPDATE-UPDATE conflicts. These exceptions will be discussed in detail in section 4.

3.4 Handling Predicates

In section 3.3, the **DETECTION** algorithms of conflicts are based on the simple case in which predicates are not used in READ or UPDATE operations. Here, we extend this to handle predicates.

There are three cases in a READ-UPDATE conflict: a READ with predicates and an UPDATE without predicates; a READ without predicates and an UPDATE with predicates; and a READ with predicates and an UPDATE with predicates. If a READ or an UPDATE has conflicts with each other without considering predicates, then both the first and second cases are the same, namely, conflict, since a predicate is only a qualifying expression that is used to select a subset of the nodes of a document instance.

One example could be a READ operation with predicates R_1 and a REPLACE operation without predicates RP_1.

R_1: READ(/rss/channel/item[title="xxxx"]), the target node is an *item* node, and the (PRE, POST) value of the target node is (2, 5).

RP_1: REPLACE(/rss/channel/item/author, <author> chang </author>), the target node is an *author* node, and the (PRE, POST) value of the target node is (6, 3).

Without considering predicates, according to the **DETECTION** algorithm of the READ-UPDATE conflict, R_1 has a conflict with RP_1 because the target node of R_1 is an **ANCESTOR** (in this case, **PARENT**) relation against RP_1. As shown in Fig. 10, although R_1 has a predicate [title="xxxx"], the predicate is a qualifying expression that is used to select a subset of item nodes of a document instance. In other words, item nodes which have a title child node, the text value of which is "xxxx", still have a conflict with RP_1.

It is similar to the detection of a READ without predicates – UPDATE with predicates conflict.
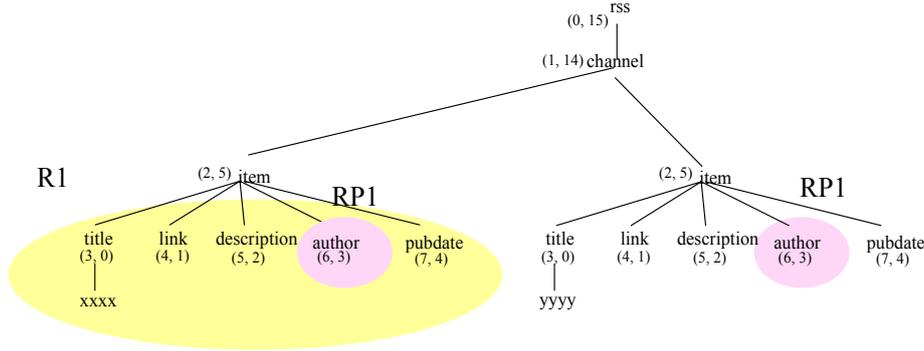
Fig. 10. A sample of READ with predicates and UPDATE without predicates.

Meanwhile, the third case is different and requires careful consideration. One example could be a READ operation R_2 and a REPLACE operation RP_2, which only have path-based predicates.

R_2: READ(/rss/channel[item/description]/author) or the target node is an *author* node, and the (PRE, POST) value of the target node is (12, 10).

RP_2: REPLACE(/rss/channel[title]/author, <author> chang </author>) or the target node is an *author* node and the (PRE, POST) value of the target node is (12, 10).

Recall that a predicate is a qualifying expression used to select a subset of the nodes of a document instance. Specifically, a path-based predicate is a Boolean predicate, that is, the predicate [item/description] is true if the relevant channel has at least one child item element having at least one child description element. However, the decision of the Boolean predicate is under the influence of a current document instance. Our approach is a pre-processing method, that is, a non-instance support from underlying databases. Thus, regardless of path-based predicates, according to the **DETECTION** algorithm of the READ-UPDATE conflict, R_2 has a conflict with RP_2 since the target node of R_2 is a **SELF** relation against RP_2.

Another example could be a READ operation R_3 and a REPLACE operation RP_3.

R_3: READ(/rss/channel/item[title="xxxx"]), the target node is an *item* node, and the (PRE, POST) value of the target node is (2, 5).

RP_3: REPLACE(/rss/channel/item[title="yyyy"]/author, <author> chang </author>), the target node is an *author* node, and the (PRE, POST) value of the target node is (6, 3).

According to the **DETECTION** algorithm of the READ-UPDATE conflict, R_3 has a conflict with RP_3 because the target node of R_3 is an **ANCESTOR** (in this case, **PARENT**) relation against RP_3. However, since the subset of R_3 is different from that of RP_3, they have no conflict with each other. This observation leads to two extended detection algorithms such as detection READ with predicates – UPDATE with predicates conflict and detection UPDATE with predicates – UPDATE with predicates conflict.

The case between a READ operation with value-based predicates and an UPDATE operation with path-based predicates, or vice versa, is similar to the preceding one.

For simplicity, listed below are some definitions of the functions:

− PREDICATES(o): returns the predicates set for an operation o.
− VALUE(p): returns the value for a predicate p.
− (Pr_p, Po_p): (PRE, POST) pair(s) of the target node of a predicate p.

Detection of READ with predicates – UPDATE with predicates Conflict   Suppose that R_i has a conflict with U_j, without predicates. If the following condition is satisfied, then both R_i and U_j, with predicates, have no conflict with each other. Otherwise, they have a conflict with each other.

− The type of all predicates of R_i and U_j are value-based predicate types.
− Each predicate of R_i is a SELF relation against each corresponding predicate of U_j (one-to-one and onto), or vice versa, but there is more than one arbitrary predicate set (p_i, q_j), the values of which are not the same.

$$PREDICATES(R_i) = PREDICATES(U_j), \exists p_i \in PREDICATES(R_i), \exists q_j \in PREDICATES(U_j)$$
$$\Rightarrow (Pr_{pi} = Pr_{qj}, Po_{pj} = Po_{qj}) \wedge (VALUE(p_i) \neq VALUE(q_j))$$

It is similar to the detection of an UPDATE with predicates – UPDATE with predicates conflict.

4. EVALUATION

4.1 Solving the Problems

Generally, the optimistic concurrency control provides non-blocking and deadlock freedom. In this paper, our major goal is to develop a new optimistic concurrency control scheme in valid XML, which solves the phantom problem caused by the tree-based locking scheme, the pseudo-conflict problem by the multigranularity locking scheme, and the lost updates problem by the snapshot-based scheme.

Fig. 11 shows a sample of solving the phantom problem caused by the tree-based locking scheme. In our approach, T_1 has a conflict with T_2 because T_2 is a **DESCEN-DANT** relation against T_1 ($Pr_{T2} > Pr_{T1}$ and $Po_{T2} < Po_{T1}$). Thus, the phantom phenomenon does not occur.

Fig. 12 shows a sample of solving the pseudo-conflict problem caused by the multi-granularity scheme. In our approach, T_2 is a **FOLLOWING** relation against T_1 ($Pr_{T2} > Pr_{T1}$ and $Po_{T2} > Po_{T1}$). Therefore, they have no conflict with each other.

4.2 Self-performance Evaluation

With respect to conflict detection, it is difficult to compare our pre-processing **DE-TECTION** method with a lock conflict test because this test should traverse the XML tree.
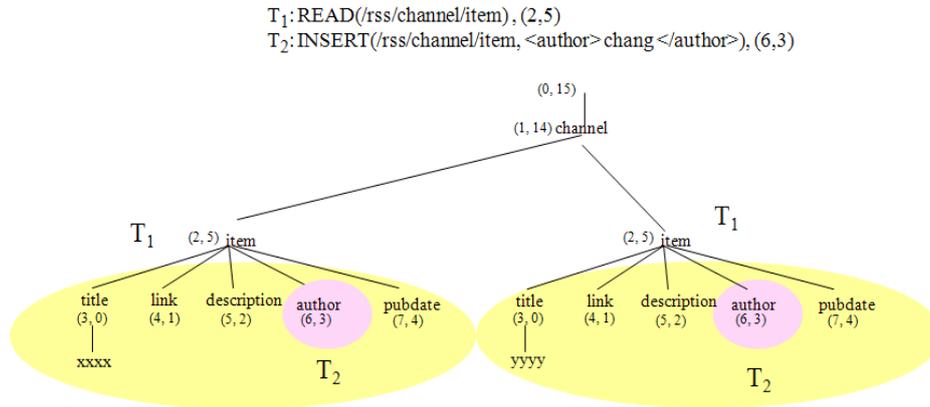
T_1: READ(/rss/channel/item), (2,5)
T_2: INSERT(/rss/channel/item, <author> chang </author>), (6,3)

Fig. 11. A sample of solving the phantom problem.

T_1: READ(/rss/channel/item), (2,5)
T_2: INSERT (/rss/channel/lastmodified, "..."), (11,9)
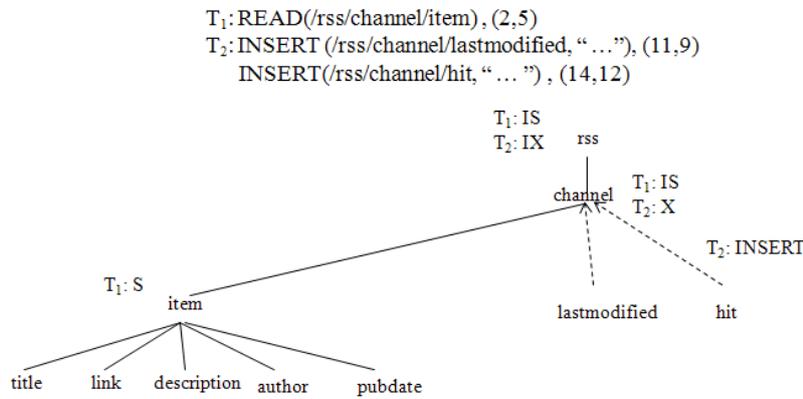INSERT(/rss/channel/hit, "..."), (14,12)

Fig. 12. A sample of solving the pseudo-conflict problem.

Hence, we evaluated the self-performance of the ***DETECTION*** algorithms of conflicts. We implemented the ***DETECTION*** algorithm in the Java programming language using the Eclipse v.3.1.1 development tool. Our experiments were performed on a Pentium IV 2.66GHz platform with an MS-Windows XP OS and 1 GB of main memory.

According to an auction.dtd generated by the publicly available XMark [26], we measured the speed of the construction time by referring to the hash table generation time of a DTD, as shown in Fig. 5 (b). Fig. 13 shows each construction time. Based on this, the metadata of a DTD have minimal overhead.

We tested the conflict detection time against 1000, 5000, 10000, 40000, 90000, and 250000 random operation work sets, as shown in Fig. 14. We assume that a transaction has an operation. Although it is important to evaluate query performance, update performance, and transaction throughput according to the proportion of READs to UPDATEs, there is no point in evaluating them in the conflict detection part because conflict detection is used to compare each PRE/POST between READs and UPDATEs.
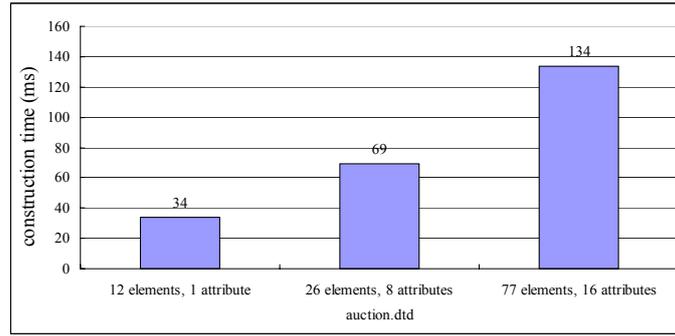
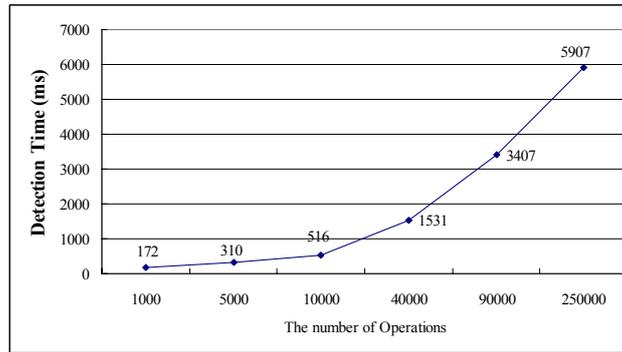Fig. 13. The construction time of a DTD's PRE/POST structure.

Fig. 14. The conflict detection time to an operation.

5. DISCUSSION

Although this paper has already proposed the strict UPDATE-UPDATE conflict in section 3.2, there is still a need to tackle the variant of DELETE-UPDATE conflicts out of UPDATE-UPDATE conflicts, which are called conservative DELETE-UPDATE conflicts.

Definition 9 [Conservative DELETE-UPDATE conflict] If the target node of a DELETE D_i is an *ANCESTOR* relation against that of any UPDATE U_j, then U_j is ignored.

One example could be a DELETE operation D_1 and a REPLACE operation RP_4.

D_1: DELETE(/rss/channel/item), the target node is an *item* node, and the (PRE, POST) value of the target node is (2, 5).

RP_4: REPLACE(/rss/channel/item/author, <author> chang </author>), the target node is an *author* node, and the (PRE, POST) value of the target node is (6, 3).

According to the *DETECTION* algorithm of the UPDATE-UPDATE conflict, D_1 has a conflict with RP_4 because the target node of D_1 is an *ANCESTOR* (in this case,

PARENT) relation against RP_4. Observe that if RP_4 is executed after D_1, then RP_4 should be rejected since the author element is removed by D_1. However, since the DETECTION algorithm of the UPDATE-UPDATE conflict is only interested in the UPDATEUP-DATE conflict, a rejection is not necessary. If RP_4 is simply ignored, then the effect will be the same, namely, $D_1(RP_4(t)) = D_1(t)$, where $t \in T_\Sigma$. This observation leads to a conservative DELETE-UPDATE (including DELETE, INSERT, and REPLACE) conflict.

6. FUTURE WORK

In most cases, the pure optimistic concurrency control performed worse than the locking approach because of a high abort rate or cyclic restarts. However, our proposed optimistic concurrency control scheme detects a transaction's abort before the *read* phase and avoids the interval between the *read* and *validation* phases. Thus, restarts are designed to have low fetch costs and can thus be executed very quickly.

As this paper lacks comparison with other approaches in terms of some examples, the scheme could be strengthened by additional evidence demonstrating its strengths over the conventional approaches. The quantitative analysis of execution times according to a real time variance of computerized resources needed for the proposed optimistic concurrency control scheme must be included in future works.

In addition, the change of an XML document by an UPDATE operation may affect other operations. We will develop a dynamic optimistic concurrency control protocol in valid XML. A *validation* phase in traditional optimistic protocols is needed to test whether or not the schedule produced so far is still a serializable one. However, a *validation* phase in a dynamic optimistic concurrency control protocol is needed not only to test whether or not the schedule produced so far is still a serializable one, but also to test if UPDATE operations in a transaction are valid in a corresponding DTD.

7. CONCLUSION

Concurrency is bound to emerge as a major challenge as more and more popular scalable database applications are built. As native XML database systems become increasingly popular, fine-granularity concurrency control becomes imperative in order to allow multiple users to concurrently access the same documents.

In this paper, we primarily presented effective detection algorithms of conflicts in valid XML documents, which exploit the tree properties encoded in the PRE/POST plane to detect READ-UPDATE and UPDATE-UPDATE conflicts. We investigate an optimistic concurrency control approach for valid XML, which possesses the properties of deadlock freedom and predictable blocking time.

This approach detects a conflict between operations early. Thus, our detection algorithms drastically reduce transaction restarts. In addition, our approach is a pre-processing mechanism which is independent from the underlying valid XML database engine. Consequently, the techniques proposed in this paper could be built on top of any valid XML DBMS.

The detection of conflicts between operations is very interesting in the query optimization for a query compiler and concurrency for a transaction manager. As such, it is expected that the proposed techniques are very useful in developing them.

REFERENCES

1. Extensible Markup Language (XML) 1.0, 2nd ed., http://www.w3.org/TR/2000/REC-xml-20001006.
2. XQuery 1.0 and XPath 2.0 Full-Text, http://www.w3.org/TR/xquery-full-text/.
3. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Updating XML," in *Proceedings of the 21st International ACM SIGMOD Conference on Management of Data*, 2001, pp. 413-424.
4. XQuery Update Facility Requirements, http://www.w3.org/TR/xquery-update-requirements.
5. XQuery Update Facility Use Cases, http://www.w3c.org/TR/xqupdateusecases.
6. T. Fiebig, S. Helmer, C. C. Kanne, G. Moerkotte, J. Meumann, R. Shiele, and T. Westmann, "Anatomy of a native XML base management system," *VLDB Journal*, Vol. 11, 2002, pp. 291-314.
7. H. V. Jagadish, S. Al-Kahalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A native XML database," *VLDB Journal*, Vol. 11, 2002, pp. 274-291.
8. X. Meng, D. Luo, M. L. Lee, and J. An, "OrientStore: A schema based native XML storage system," in *Proceedings of International Conference on Very Large Data Bases*, 2003, pp. 1057-1060.
9. P. A. Bernstein, V. Hardzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Massachusetts, 1987.
10. R. Agrawal and D. J. DeWitt, "Integrated concurrency control and recovery mechanisms: Design and performance evaluation," *ACM Transactions on Database Systems*, Vol. 10, 1985, pp. 529-564.
11. D. A. Menasce and T. Nakanishi, "Optimistic versus pessimistic concurrency control mechanisms in database management systems," *Information Systems*, Vol. 7, 1982, pp. 88-92.
12. R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Transactions on Database Systems*, Vol. 12, 1987, pp. 609-654.
13. M. J. Carey and M. R. Stonebraker, "The performance of concurrency control algorithms for database management systems," in *Proceedings of International Conference on Very Large Data Bases*, 1984, pp. 107-118.
14. S. Helmer, C. C. Kanne, and G. Moerkotte, "Evaluating lock-based protocols for cooperation on XML documents," *ACM SIGMOD Record*, Vol. 33, 2004, pp. 58-63.
15. T. Grabs, K. Böhm, and H. J. Schek, "XMLTM: Efficient transaction management for XML documents," in *Proceedings of the 13th ACM Conference on Information and Knowledge Management*, 2002, pp. 142-152.
16. S. Dekeyser, J. Hidders, and J. Paredaens, "A transaction model for XML databases," *World Wide Web Journal*, 2003, pp. 1-36.
17. W. Zhang, D. Liu, and W. Sun, "XR-lock: Locking XML data for efficient concurrency control," in *Proceedings of the 5th World Congress on Intelligent Control and Automation*, 2004. pp. 3921-3925.
18. M. Haustein and T. Härder, "An efficient infrastructure for native transactional XML processing," *Data & Knowledge Engineering*, Vol. 61, 2007, pp. 500-523.

19. M. Raghavachari and O. Shmueli, "Conflicting XML updates," in *Proceedings of the 10th International Conference on Extending Database Technology*, 2006, pp. 552-569.
20. T. L. Saito and S. Morishita, "Xerial: An update tolerant and high concurrent XML database," in *Proceedings of the 4th Data Mining Workshop*, 2004, pp. 13-20.
21. T. L. Saito and S. Morishita, "Efficient integration of structure indexes of XML," in *Proceedings of the 12th International Conference on Database Systems for Advanced Applications*, 2007, pp. 781-792.
22. Z. Sardar and B. Kemme, "Don't be a pessimist: Use snapshot based concurrency control for XML," in *Proceedings of the 22nd International Conference on Data Engineering*, 2006, pp. 130-132.
23. M. Haustein, T. Härder, and K. Luttenberger, "Contest of XML lock protocols," in *Proceedings of International Conference on Very Large Data Bases*, 2006, pp. 1069-1080.
24. T. Grust, "Accelerating XPath location steps," in *Proceedings of the 21st International ACM SIGMOD Conference on Management of Data*, 2002, pp. 109-120.
25. S. Mohan, A. Sengupta, Y. Wu, and J. Klinginsmith, "Access control for XML-A dynamic query rewriting approach," in *Proceedings of the 14th ACM Conference on Information and Knowledge Management*, 2005, pp. 251-252.
26. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse, "The XML benchmark project," Technical Report INS-R0103, CWI, Amsterdam, Netherlands, 2001.

Changwoo Byun (邊昶又) received the B.S., M.S. and Ph.D. degrees in the Department of Computer Science and Engineering from Sogang University, Seoul, Korea, in 1999, 2001 and 2007 respectively. Since Sep. 2007, he has been working in the Department of Computer Systems and Engineering of Inha Technical College. His areas of research include role-based access control model, access control for distributed systems, access control for XML data, XML transaction management, and Ubiquitous security.

Ilkook Yun (尹一國) received the B.S. degree in the Department of Computer Science and Engineering from Sogang University, Seoul, Korea, in 2006. Since 2006, he has been studying for a M.S. at the Department of Computer Science and Engineering of Sogang University. His areas of research include XML transaction management, concurrency control for XML data, and Web DB security.

Seog Park (朴錫) is a Professor of Computer Science at Sogang University. He received the B.S. degree in Computer Science from Seoul National University in 1978, the M.S. and the Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1980 and 1983, respectively. Since 1983, he has been working in the Department of Computer Science and Engineering, Sogang University. His major research areas are database security, real-time systems, data warehouse, digital library, multimedia database systems, role-based access control and Web database. Dr. Park is a member of the IEEE Computer Society, ACM and the Korea Information Science Society. Also, he has been a member of Database Systems for Advanced Application (DASFAA) steering committee since 1999.