

XPath-based Concurrency Control in XML Document Management

EUN HYE CHOI[†] and TATSUNORI KANAI[†]

Although concurrency control has been recognized as an important issue in XML data management, there are still few previous works that provide the concurrency of transactions retrieving and modifying the same XML documents. To overcome this problem, we propose a new XPath-based concurrency control scheme, called XPCC, that guarantees high concurrency and serializability of concurrent transactions to the same XML document. The proposed approach considers unrestricted XML documents and general XPath query, and the key ideas of XPCC are as follows: (1) semantic locks are set on XPath expressions used in transaction accesses, and (2) two versions of an XML document, a version containing updates of each transaction and a version containing updates of all concurrent transactions, are utilized for conflict checks. In XPCC, at the time that each transaction access is requested, the conflict to violate serializability is detected based on the equivalence check for results of XPath evaluation against two versions of documents. Since the proposed approach enables locking to be at the level of precise data actually retrieved and updated in XML documents, high concurrency can be achieved assuring serializability in XML data management.

1. Introduction

As the eXtensible Markup Language (XML)²⁾ becomes widely adopted in various application areas, the number of XML documents is rapidly growing and the subsequent need for sharing those documents by multiple users and applications is increasing more and more. For example, consider a system that manages metadata of contents in XML documents. The metadata will be retrieved using a query language to XML documents such as XPath⁵⁾. At the same time, the metadata such as the content's location will be updated when changing the location of contents or replicating contents. In such application environments where transactions concurrently retrieve and modify XML data in the same document, data consistency and concurrency control are important issues.

In general, XML documents are stored as either binary large objects (BLOBs) in relational databases or XML format in native XML databases. In the former case, the entire document is replaced when modifying data, and thus concurrent updating on the same document cannot be processed. In the latter case, similarly, locking at the level of entire documents is often adopted, and thus no concurrency of transactions to the same document is actually provided¹⁾. Few solutions to the concurrency control problem for XML document manage-

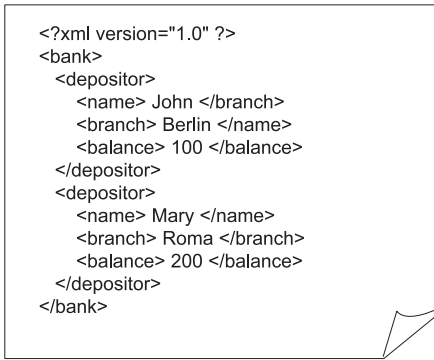
ment have been proposed so far. To overcome this problem, we propose an XPath-based concurrency control method, called XPCC, that achieves high concurrency of transactions retrieving and updating data in the same XML document while maintaining strict data consistency and serializability at the same time.

In XPCC, we consider XML documents with no restriction and the general XPath query for transaction accesses. Since XPath, which is to address parts of an XML document, is being used as a base in a number of other XML standards and query languages such as XQuery³⁾, XPCC can be easily extended for other querying based on XPath.

From the concurrency point of view, one of the most fundamental problem is the serializability constraint, which requires that concurrent transactions produce the same result as the same transactions executed in a certain sequential order and prevent the phantom problem⁶⁾. As traditional techniques to solve the serializability problem, pessimistic locking approach and optimistic approach have been extensively investigated so far.

In the optimistic approach, all transaction accesses are concurrently performed without locking and consistency across concurrent transactions is validated at commit time of each transaction. Although this approach involves less locking overhead, it has shortcomings such that the validation phase is necessary instead of locking and the whole transactions are aborted if there is a conflict.

[†] Corporate Research & Development Center, Toshiba Corporation



```

<?xml version="1.0" ?>
<bank>
  <depositor>
    <name> John </branch>
    <branch> Berlin </name>
    <balance> 100 </balance>
  </depositor>
  <depositor>
    <name> Mary </name>
    <branch> Roma </branch>
    <balance> 200 </balance>
  </depositor>
</bank>

```

Fig. 1 An XML document.

In the pessimistic locking approach, there are two general strategies, predicate locking⁶⁾ and its approximation, granular locking⁹⁾. In predicate locking, locks are set on predicates used in transaction accesses and conflicts are detected by checking the satisfiability of predicates. Although predicate locking is a complete solution to the serializability requirement, it is quite expensive since the satisfiability between arbitrary two predicates is known to be NP-complete.

In contrast, in granular locking, locks are set on granules of data instead of predicates. Since the granular locking is less expensive, it has been widely adopted for relational database management systems. However, in XML document management considering the general XPath query, the granular locking approach hardly provides high concurrency and ensures the consistency at the level of XML semantics. For example, consider an XML document shown in **Fig. 1** and two concurrent transactions T_1 and T_2 . Suppose that T_1 inserts a new depositor next to depositor “Mary”. If T_2 retrieves data using XPath expression “depositor[name=’Mary’]”, there is no conflict. However, if T_2 retrieves the same data using XPath expression “depositor[position()=last]”, we must block either T_1 or T_2 . To ensure the data consistency, all elements about depositors should be locked in granular locking, and then no concurrency on the document is provided.

To overcome this problem, we adopt a kind of improved predicate locking called *precision locking*¹²⁾ in XPCC. In precision locking, conflicts between predicates and updates are checked instead of those between two predicates in the following way: As a transaction performs a read and a write, the predicate used in the read and the update by the write are posted

in a predicate list and an update list, respectively. In order to check the conflicts between predicates and updates by different transactions, each predicate (update) posted is checked against updates (predicates) by other transactions in the update list (the predicate list). When a conflict is detected, the access is delayed until the other transaction that the access conflicts finishes. Precision locking performs the conflict check against only actual predicates and updates, and thus it provides high concurrency and a relatively lower cost than the predicate locking. Although the precision locking can preserve high concurrency for XML data from its generality, the problem to perform practical conflict checks still remains.

For the conflict check, analyzing the satisfiability of an arbitrary set of updates with the XPath expression used in each read is necessary but is difficult since XPath is a path-based and semistructured query language, not a generic boolean predicate. In order to handle this deficiency, we propose a method called an equivalence check that can detect any conflict between the update and the XPath expression while evaluating the XPath expression with two documents such that one contains the update and the other does not. The semantics of XPath with the proposed equivalence check is also presented in this paper.

The basic ideas of the proposed method XPCC are as follows: First, we introduce a new document management model, which handles the three kinds of documents: (1) an original document in the database, denoted by D_{st} , (2) a local copy of D_{st} containing the updates by each active transaction T_i , denoted by D_i , and (3) a local copy of D_{st} containing all concurrent updates, denoted by D_{all} . Next, conflicts are checked based on the proposed equivalence check performed in XPath evaluation with two documents. As for the conflict check for each read requested, documents D_i and D_{all} are used for the equivalence check. As for the conflict check for each write requested, previous states of documents regenerated from D_{st} are used for the equivalence check.

The rest of this paper is organized as follows: Previous works related on concurrency control on XML documents are addressed in Section 2. In Section 3, we describe the models of XML documents, transactions and XPath. The document management model and the overview of XPCC are presented in Section 4. In Sections

5 and 6, we explain the conflict check based on XPath evaluation with equivalence checking and propose the algorithms for the conflict check. Finally, we summarize this paper with some future works in Section 7. In addition, the denotational semantics of the proposed XPath evaluation with equivalence checking is presented in the appendix.

2. Related Works

To our best knowledge, the only two concurrency control approaches (Refs. 7), 11) that consider concurrent retrieving and updating on the same XML documents have been proposed so far.

Jea et al. proposed a method called *XLP* in Ref. 11). *XLP* adopts granular locking and considers axis directions containing ancestor nodes in XPath expressions differently from simple tree locking. However, since *XLP* physically locks data retrieved and updated in XML documents, it can not prevent the phantom problem when considering predicate expressions of the XPath.

In Ref. 7), Grabs, et al. proposed a locking method called *DGLOCK* that can tackle this problem. *DGLOCK* is a combination of granular locking and precision locking based on the alternative structure of an XML document, called a *DataGuide*⁸⁾. *DGLOCK* utilizes the characteristic of the *DataGuide* such that the same path occurs exactly once, and locks are set on all nodes in the *DataGuide* that match any of the path expressions. Predicate locks are also set on nodes of which values are referred in predicate expressions. Although *DGLOCK* can provide a high degree of concurrency, it has several restrictions since locking relies on *DataGuide* as the underlying structure. First, it could be applied to XML documents with limited node types since *DataGuide* does not support every types of nodes in XML specification. Secondly, it assumes only simple XPath expressions for querying such that path expressions have an axis containing descendant nodes and predicates are conjunctions of the form $[x \theta c]$ with node x , $\theta \in \{=, \in, \neq, \leq, \dots\}$ and constant c .

Against Refs. 7), 11), the proposed concurrency control can be applied to XML documents and XPath expressions with no restriction.

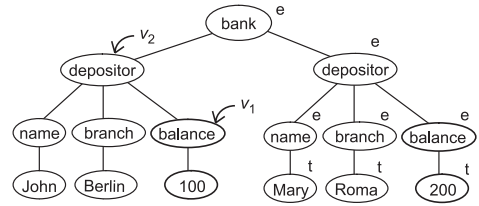


Fig. 2 A tree model.

3. Preliminaries

In this section, we first describe the models of XML documents and transaction accesses and next give the semantics of XPath and some definitions used in our research.

3.1 XML Documents and Transaction Accesses

An XML document (*document*, for short) is modeled as a tree whose nodes are classified into various node types, e.g., *element*, *text* and *attribute*, as defined in XPath data model⁵⁾. Figure 1 and Fig. 2 show an example XML document and its representation as a tree, respectively. In Fig. 2, “e” and “t” attached to each node represent that the node is an element node and a text node, respectively.

We assume that transactions access a document using general XPath expressions in Ref. 5). In this paper, we consider a single document as a target of transaction accesses, just for simplicity. Transactions read and write data in a document in the following manner.

(1) A *read* is performed by specifying a document and an XPath expression that identifies a node (or nodes) in the document. In the following, a read operation is denoted by $read(\epsilon)$ with an XPath expression ϵ , and the result of evaluating XPath expression ϵ on document D is denoted by $get(D, \epsilon)$.

(2) A *write* is performed by specifying a write operation and a target node in the document. The target node is selected by a read before the write. We assume all the three kinds of write operations: insert, delete and replace. In the following, the state of document D updated by a write w (a set of writes W) is denoted by

In this paper, we assume that the reader is familiar with basic notations of XPath, e.g., a location path, a location step and an axis, and its evaluation. For more detail, refer to Ref. 5).

Actually, an XPath expression is evaluated with a context node in the document. For simplicity, we consider the root node of the document as the context node otherwise indicated.

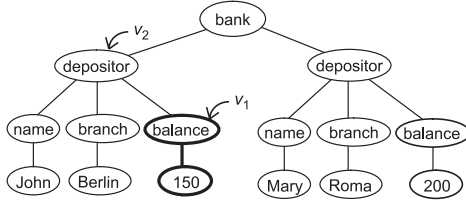


Fig. 3 An updated document.

$p ::= p \mid p \mid /p \mid p/p \mid p[q] \mid p[e] \mid$
 $axis :: nodetest$
 $q ::= q \text{ and } q \mid q \text{ or } q \mid q = q \mid q! = q \mid$
 $q < q \mid q > q \mid q \leq q \mid q \geq q \mid p \mid e$
 $e ::= e + e \mid e - e \mid e * e \mid e \div e \mid$
 $e \text{ mod } e \mid -e \mid p \mid q$
 $axis ::= forward_axis \mid reverse_axis$
 $forward_axis ::= self \mid child \mid descendant \mid$
 $descendant-or-self \mid$
 $following \mid$
 $following-sibling$
 $reverse_axis ::= parent \mid ancestor \mid$
 $ancestor-or-self \mid$
 $preceding \mid$
 $preceding-sibling$
 $nodetest ::= name \mid * \mid text() \mid node()$

Fig. 4 Abstract syntax of XPath.

$D+w$ ($D+W$).

Example 1 Consider a read with expression $\epsilon_1 = \text{“depositor[name=‘John’]/balance”}$ to the document shown in Fig. 2. Node v_1 in Fig. 2 is then selected as a result of the read, i.e. $get(D, \epsilon_1) = \{v_1\}$. The value of element node v_1 equals to “100”. Consider a write that replaces the value of node v_1 to “150”. Document D is then updated by the write as shown in Fig. 3. The updated nodes are marked with bold lines in the figure. \square

Through this paper, we consider a set of concurrent transactions, $\mathcal{T} = \{T_1, \dots, T_n\}$, that access the same document. Each transaction is a time-ordered sequence of reads and writes. In the following, n denotes the number of transactions and the term *transactions* means concurrent active transactions otherwise indicated. In the following, W_i denotes the sequential set of writes of transaction T_i .

3.2 Semantics of XPath

Figure 4 describes the abstract syntax of XPath expressions we consider. Expression p is a location path, shortly called a pattern or a path. Expressions q and e are referred to as a qualifier and an expression, respectively. To save space, we here omit the precise explanation for the general semantics of XPath. Please refer to Refs. 15) or 17) for more detail. Note

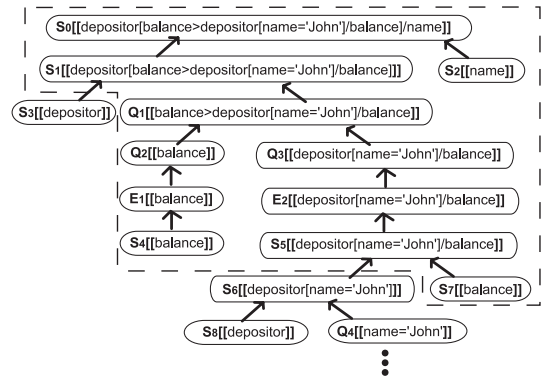


Fig. 5 An example evaluation tree.

that the proposed approach can also be applied to general syntax not mentioned in Fig. 4.

The semantics of XPath is specified by three functions \mathcal{S} , \mathcal{Q} and \mathcal{E} with an expression and a context node. $\mathcal{S}[p]x$ denotes the node-list selected by pattern p with context node x . A node-list represents an ordered set of nodes, and the ordering of a node-list is determined as follows: if path p is a location step with a reverse-axis, the result node-list selected by p is in reverse document order; otherwise, the result node-list is in document order. In addition, $\mathcal{Q}[q]x$ denotes the boolean value of qualifier q , and $\mathcal{E}[e]x$ denotes the numerical value of expression e .

Each function \mathcal{F} ($=\mathcal{S}, \mathcal{Q}, \mathcal{E}$) is evaluated by recursively calling a function (or functions) if its expression has a subexpression. Let $\mathcal{F} \leftarrow \mathcal{F}'$ denote that function \mathcal{F} calls function \mathcal{F}' and refer to such function \mathcal{F}' as a *sub-function* of \mathcal{F} . The evaluation of XPath expression ϵ ($=p, q, e$) is then represented by a directed tree whose root node represents a function with ϵ and child nodes represent sub-functions of a function corresponding to their parent. Hereafter, we refer to the tree representing these function calls in the evaluation of XPath expression ϵ as an *evaluation tree* of ϵ .

Example 2 Consider an XPath expression $\epsilon = \text{“depositor[balance > } \epsilon_1\text{]/name”}$ with ϵ_1 in Example 1. Figure 5 illustrates the evaluation tree of ϵ . For simplicity, the context node associated with each function is omitted in the evaluation tree in Fig. 5. \square

Among three functions \mathcal{S}, \mathcal{Q} and \mathcal{E} , both functions \mathcal{Q} and \mathcal{E} can be recursions of function \mathcal{S} and the results of them are determined from the result node-list of \mathcal{S} . Note that function \mathcal{S} changes the context node but functions

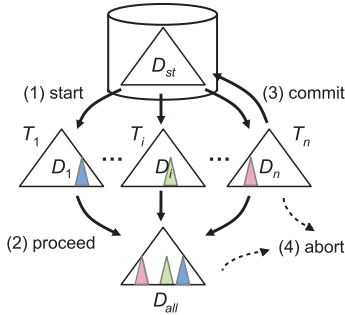


Fig. 6 Document management model.

\mathcal{Q} and \mathcal{E} don't.

4. Overview of XPCC

This section presents our document management model and the overview of the concurrency control mechanism of XPCC.

4.1 Document Management Model

Figure 6 illustrates our document management model. Each triangle in the figure represents a document state and the three kinds of document states are hold:

- D_{st} : the document state committed.
- $D_i (1 \leq i \leq n)$: the document state containing only the updates of transaction T_i .
- D_{all} : the document state containing the updates of all transactions.

Initially, D_i 's and D_{all} are copied from D_{st} , and their states are managed in the following manner:

- (1) When each transaction T_i starts, document D_i is generated as a copy of D_{st} .
- (2) While T_i proceeds, T_i accesses D_i and its update is reflected in both D_i and D_{all} .
- (3) When T_i commits, its updates are reflected in D_{st} and D_j 's ($1 \leq j \leq n, j \neq i$).
- (4) When T_i aborts, D_{all} is regenerated for taking away the updates of T_i contained in D_{all} .

In our document management model, the abort or rollback of a transaction can be easily handled since the updates of each transaction T_i are not contained in D_{st} before commitment.

Here we define an *equivalence of nodes* in different document states as follows.

Definition 1 Nodes v and v' are equivalent each other, denoted by $v \equiv v'$, in the following cases: (1) node v in D_i was copied from node v' in D_{st} , (2) nodes v and v' in different D_i 's were copied from the same node in D_{st} , or (3) nodes v and v' in different D_i 's or D_{all} were updated by the same write. \square

		Previous access by other transactions	
		Read	Write
Requested access	Read	a) -	b) \checkmark
	Write	c) \checkmark	d) -

\checkmark means that conflict checks are needed.

Fig. 7 Conflicts to be checked.

For example, consider the documents shown in Figs. 2 and 3 again. Nodes v_2 's in the two documents are equivalent but nodes v_1 's are not equivalent each other.

4.2 Concurrency Control

The concurrency control of XPCC is performed by checking conflicts between a requested access (read or write) and previous accesses by other concurrent transactions when the access is requested. If a conflict is detected, the requested access is blocked until the transaction that conflicts with the access finishes. (Actually the requested access could preempt the other transaction, but we assume not in this paper.) Deadlock which occurs by blocking transactions can be detected by using well-known solutions such as the wait-for graph⁹).

To ensure transaction serializability, the following two kinds of conflict checks are needed, as shown in Fig. 7: a *read-write conflict* (Case b) and a *write-read conflict* (Case c). No conflict exists between reads (Case a). On the other hand, a *write-write conflict* (Case d) can be detected by a previous read-write conflict or a previous write-read conflict. The reason is that before a write operation on the nodes that leads to a conflict, a read operation retrieving the target nodes must be performed previously. The proposed conflict check classifies into the following two phases: a *read-write check* and a *write-read check*.

- Read-Write Check

Each time a transaction requests a read, check whether the read leads to the read-write conflict with previous writes of other transactions.

- Write-Read Check

Each time a transaction requests a write, check whether the write leads to the write-read conflict with previous reads of other transactions.

In XPCC, the conflict between a read and a write is detected by performing the read against two documents such that one contains the update of the write and the other does not and comparing results of XPath evaluation against

those two documents. The mechanism for conflict detection is described in Section 5, and the mechanisms for the read-write check and the write-read check are described in Section 6.

5. XPath-based Conflict Detection

In this section, we explain how to detect conflicts between reads and writes by different transactions in XPCC. The conflict detection is based on the proposed equivalence checking performed in XPath evaluation under our document management.

5.1 Conflict Detection between Read and Write Accesses

Consider a transaction T_i and its corresponding document D_i which is a local copy for T_i . For a read r_i with an XPath expression ϵ requested by T_i , the result of $r_i (= read(\epsilon))$ is obtained by evaluating expression ϵ on document D_i . The evaluated result of an XPath expression is a node-list (an ordered set of nodes), a boolean, a number or a string. Here we define an *equality of result node-lists* from different document states as follows.

Definition 2 For node-lists N and N' from different document states, $N = N'$ iff $|N| = |N'|$ and for any i -th nodes $n(\in N)$ and $n'(\in N')$ with $1 \leq i \leq |N|$, $n \equiv n'$. \square

Lemma 1 Consider a read r_i requested by a transaction T_i and a sequential set of writes, W_j , of the other transaction T_j . Iff there is a conflict between r_i and W_j ,

$$get(D_i, \epsilon) \neq get(D_i + W_j, \epsilon). \quad (5.1)$$

\square

Obviously, if the result of read r_i to D_i is not equal to the result of r_i to the state of D_i containing the updates by W_j , read r_i leads to a conflict. By Lemma 1, a conflict between read r_i and writes W_j of any other transaction T_j is detected by checking formula 5.1.

Here one must notice that read r_i could conflict with a set of writes by more than one concurrent transactions even when r_i does not conflict with writes of any of the transactions. The reverse case is also considered.

Example 3 Consider that T_1 requests $r_1 = read(\epsilon)$ with XPath expression ϵ in Example 2 to the document shown in Fig. 2. In addition, consider that T_2 replaces the balance of John's branch to "150" ($=W_2$) and T_3 replaces the value of Mary's balance to "150" ($=W_3$).

Read r_1 conflicts with $W_2 + W_3$ but with neither W_2 nor W_3 since $get(D_1, \epsilon) = get(D_1 + W_2, \epsilon) = get(D_1 + W_3, \epsilon) \neq get(D_1 + W_2 + W_3, \epsilon)$. To consider the reverse case, suppose that T_2 replaces the balance of John's branch to "200" ($=W'_2$) and T_3 replaces the value of Mary's balance to "300" ($=W'_3$). Then r_1 conflicts with W'_2 but not with $W'_2 + W'_3$. \square

Lemma 2 Consider a read $r_i = read(\epsilon)$ requested by T_i and a set of all writes, \mathcal{W} , of concurrent transactions \mathcal{T} . Iff read r_i does not lead to a conflict,

$$get(D_i, \epsilon) = get(D_i + \mathcal{W}', \epsilon) \quad (5.2)$$

for writes $\mathcal{W}' (\subseteq \mathcal{W})$ of any combination of transactions in \mathcal{T} . \square

Hereafter, let notation \mathcal{W} denote the set of all writes of concurrent transactions where the writes by different transactions do not conflict each other. In addition, we define an *equivalence of results* evaluated on a document and an updated state of the document by writes \mathcal{W} as follows.

Definition 3 Consider a read $r_i = read(\epsilon)$ and writes \mathcal{W} of concurrent transactions \mathcal{T} .

$$get(D_i, \epsilon) \equiv get(D_i + \mathcal{W}, \epsilon) \quad (5.3)$$

iff for $\mathcal{W}' (\subseteq \mathcal{W})$ of any combination of transactions in \mathcal{T} , equation 5.2 holds. \square

By Lemma 2 and Definition 3, read r_i leads to no conflict iff the result of ϵ on D_i is equivalent to the result of ϵ on $D_i + \mathcal{W}$, that is, the equality in formula 5.2 holds for all combinations of transactions. However, such an equivalence check is extremely time consuming. (The number of all combinations to be considered for each read requested is $2^{n-1} - 1$ where n is the number of concurrent transactions.)

To handle this problem, we present sufficient conditions for the equivalence in formula 5.2 in the following. In XPCC, conflicts are detected by checking such conditions while evaluating XPath expressions on documents, and thus it is not needed to consider all combinations of transactions. Sufficient and efficient conditions for the conflict detection, i.e., those for the equivalence check are proposed in Sections 5.2 and 5.3.

5.2 Equivalence Checking — Sufficient Condition

Consider document D_i , the state of D_i updated by writes \mathcal{W} of concurrent transactions and XPath expression ϵ . (Recall that any writes in \mathcal{W} by different transactions do not conflict each other.) Hereafter, let $x:D_i$ and $x:D'_i$ denote context nodes in documents D_i and D'_i

Recall that the equivalence of nodes was defined in Definition 1.

respectively such that the nodes are equivalent each other.

Lemma 3 Consider documents D_i and $D_i + \mathcal{W}$ and XPath expression ϵ . If $get(D_i, \epsilon) \neq get(D_i + \mathcal{W}, \epsilon)$,

$$\mathcal{S}[[p]]x:D_i \neq \mathcal{S}[[p]]x:(D_i + \mathcal{W})$$

for some functions \mathcal{S} with pattern p called when evaluating ϵ on D_i and $D_i + \mathcal{W}$.

proof: Suppose that no such function \mathcal{S} is called when evaluating ϵ , that is, \mathcal{S} always returns the same results for D_i and $D_i + \mathcal{W}$. Obviously, $get(D_i, \epsilon) = get(D_i + \mathcal{W}, \epsilon)$. This is a contradiction. \square

Theorem 1 Consider documents D_i and $D_i + \mathcal{W}$ and XPath expression ϵ . If $get(D_i, \epsilon) \neq get(D_i + \mathcal{W}, \epsilon)$,

$$\mathcal{S}[[p]]x:D_i \neq \mathcal{S}[[p]]x:(D_i + \mathcal{W}) \quad (5.4)$$

for some functions \mathcal{S} with pattern p called when evaluating ϵ on D_i and $D_i + \mathcal{W}$.

proof: If $get(D_i, \epsilon) \neq get(D_i + \mathcal{W}, \epsilon)$, there is a set of writes $\mathcal{W}' (\subseteq \mathcal{W})$ such that $get(D_i, \epsilon) \neq get(D_i + \mathcal{W}', \epsilon)$ by Definition 3. Let $D'_i = D_i + \mathcal{W}'$ and $D''_i = D_i + \mathcal{W}$. There are two cases to be considered: $get(D_i, \epsilon) \neq get(D''_i, \epsilon)$ and $get(D_i, \epsilon) = get(D''_i, \epsilon)$. In the former case, the theorem holds by Lemma 3. Consider the latter case. Since $get(D_i, \epsilon) \neq get(D'_i, \epsilon)$, a certain function \mathcal{S} such that $\mathcal{S}[[p]]x:D_i \neq \mathcal{S}[[p]]x:D'_i$ is called when evaluating ϵ on D_i and D'_i . Then, there is a node $v \in \mathcal{S}[[p]]x:D_i$ such that no node equivalent to v exists in $\mathcal{S}[[p]]x:D'_i$. This means that the node equivalent to v in D'_i was previously updated (deleted or replaced) by \mathcal{W}' . Since \mathcal{W}' and $\mathcal{W} - \mathcal{W}'$ do not conflict each other, no node equivalent to v exists in $\mathcal{S}[[p]]x:D''_i$, neither. Hence $\mathcal{S}[[p]]x:D_i \neq \mathcal{S}[[p]]x:D''_i$. \square

By Theorem 1, any conflict between a read r_i to D_i and writes by any combination of transactions can be detected by checking formula 5.4 when each function \mathcal{S} is called in the evaluation of XPath expression ϵ on D_i and $D_i + \mathcal{W}$. Note that this is a sufficient but not a necessary condition for a conflict. In other words, calling such function \mathcal{S} whose results with D_i and $D_i + \mathcal{W}$ are not equal does not necessarily cause a conflict. For a highly efficient conflict detection, we propose efficient conditions for equivalence checking in the XPath evaluation in the following.

5.3 Equivalence Checking — Efficient Conditions

Consider an XPath expression ϵ and a document D_i . As mentioned in Section 3.2, $\mathcal{F}[[\epsilon]]x:D_i$ denotes the result of XPath expres-

sion ϵ evaluated on document D_i with context node $x (\in D_i)$, and function \mathcal{F} is recursively evaluated from its sub-functions using the evaluation tree of ϵ .

Here we define an *equivalence* of the results of function \mathcal{F} with document D_i and an updated state of D_i as follows.

Definition 4 Consider a function \mathcal{F} for an expression ϵ and documents D_i and $D_i + \mathcal{W}$.

$$\mathcal{F}[[\epsilon]]x:D_i \equiv \mathcal{F}[[\epsilon]]x:(D_i + \mathcal{W}) \quad (5.5)$$

if for $\mathcal{W}' (\subseteq \mathcal{W})$ of any combination of transactions in \mathcal{T} , equation 5.2 holds. \square

By Definitions 3 and 4, formula 5.3 holds if formula 5.5 holds. This means that if the results of function \mathcal{F} with two documents D_i and $D_i + \mathcal{W}$ are equivalent, the results of XPath expression ϵ with D_i and $D_i + \mathcal{W}$ are also equivalent, and thus a conflict does not occur.

As mentioned before, function \mathcal{F} is evaluated from its sub-functions. Thus a node corresponding to \mathcal{F} must have no children that lead to a conflict in the evaluation tree so that the results of \mathcal{F} with two documents are equivalent.

In the following, we classify function $\mathcal{F} (= \mathcal{S}, \mathcal{Q}, \mathcal{E})$ into six cases according to the syntax of XPath and give the conditions to determine the equivalence for function \mathcal{F} from the results of sub-functions of \mathcal{F} . In XPCC, the equivalence check is performed as follows: When evaluating an XPath expression ϵ used in a read, the equivalence of function \mathcal{F} in formula 5.5 is recursively computed using the proposed conditions. Since the equivalence is a sufficient condition that there is no conflict between read $r_i = read(\epsilon)$ and any writes in \mathcal{W} , conflicts are detected from the equivalence of \mathcal{F} with documents D_i and $D_i + \mathcal{W}$.

Now we propose the conditions to determine the equivalence for each function \mathcal{S} , \mathcal{Q} and \mathcal{E} with documents D_i and $D_i + \mathcal{W}$, and explain why a conflict does not occur if the conditions hold.

First, consider that function \mathcal{S} calls a sub-function \mathcal{S}' . There are three cases to be considered: (Case 1) a location path, (Case 2) a step with a predicate that is a qualifier and (Case 3) a step with a predicate that is an expression.

Suppose that $\mathcal{S}'[[p]]x:D_i \neq \mathcal{S}'[[p]]x:(D_i + \mathcal{W})$. In this case, there then exists a node v in node-list $\mathcal{S}'[[p]]x:D_i$ or $\mathcal{S}'[[p]]x:(D_i + \mathcal{W})$ such that no equivalent node to v exists in the other node-list. By Theorem 1, there then is a possibility of a conflict. However, if such node v does not effect to the result of function \mathcal{S} , a conflict does

not occur.

In order to determine the equivalence for \mathcal{S} , we introduce the notion of a *nondetermination* for nodes in result node lists. A node in a result node-list such that no equivalent node to the node exists in the other node-list and the node effects to the result in the evaluation is referred to as a nondetermined node. In addition, the result nodes selected using a nondetermined node as the context node are also referred to as nondetermined nodes. Precise definitions for each cases 1–3 are given in the following, and the equivalence for \mathcal{S} is determined by Lemma 4.

Case 1 ($p = p_1/p_2/\dots/p_m$)

Consider the case of a location path. In the evaluation of p , each node x' in the result node-list of $\mathcal{S}[p_1]x$ is used as the context node for $\mathcal{S}[p_2]x'$, and the result of $\mathcal{S}[p_1/p_2]x$ is the union of the results of $\mathcal{S}[p_2]x'$ for all nodes in $\mathcal{S}[p_1]x$. This procedure is iterated and each steps are composed from left to right.

Definition 5 For each step with $p_l (1 \leq l \leq m)$, node x'' in $\mathcal{S}[p_l]x':D_i$ ($\mathcal{S}[p_l]x':(D_i + \mathcal{W})$) is nondetermined iff (1) context node x' is nondetermined, or (2) no equivalent node with x'' exists in the other node-list. \square

Case 2 ($p = p_1[q]$)

Consider the case of a step with a predicate q . The result node-list of p contains every node x' in $\mathcal{S}[p_1]x$ such that the boolean value of $\mathcal{Q}[q]x'$ equals to 1.

Definition 6 We define that node x' in $\mathcal{S}[p]x:D_i$ ($\mathcal{S}[p]x:(D_i + \mathcal{W})$) is nondetermined iff (1) context node x is nondetermined, (2) no equivalent node with x' exists in the other node-list, or (3) $\mathcal{Q}[q]x':D_i \neq \mathcal{Q}[q]x':(D_i + \mathcal{W})$. \square

Case 3 ($p = p_1[e]$)

Consider the case of a step with a predicate e . The result node-list of p contains every node x' in $\mathcal{S}[p_1]x$ such that the proximity position of node x' in the node-list equals to the value of $\mathcal{E}[e]x'$.

Definition 7 We define that node x' in $\mathcal{S}[p]x:D_i$ ($\mathcal{S}[p]x:(D_i + \mathcal{W})$) is nondetermined iff (1) context node x is nondetermined, (2) no equivalent node with x' exists in the other node-list, (3) $\mathcal{E}[e]x':D_i \neq \mathcal{E}[e]x':(D_i + \mathcal{W})$, or (4) there is a previous node of x' in $\mathcal{S}[p_1]x:D_i$ ($\mathcal{S}[p_1]x:(D_i + \mathcal{W})$) that is nondetermined. \square

The latest condition (4) is different from the

case of a step with predicate q . The reason is that the proximity position of x' is determined from the number of previous nodes in the node-list.

The equivalence of the results of function \mathcal{S} then can be determined from the existence of nondetermined nodes in the results as follows.

Lemma 4 $\mathcal{S}[p]x:D_i \equiv \mathcal{S}[p]x:(D_i + \mathcal{W})$ if there is no nondetermined node in $\mathcal{S}[p]x:D_i$ and $\mathcal{S}[p]x:(D_i + \mathcal{W})$.

proof: If no nondetermined node exists in $\mathcal{S}[p]x:D_i$ and $\mathcal{S}[p]x:(D_i + \mathcal{W})$, $\mathcal{S}[p]x:D_i = \mathcal{S}[p]x:(D_i + \mathcal{W}')$ for any writes $\mathcal{W}' (\subseteq \mathcal{W})$ by a combination of transactions. By Definitions 3 and 4, the theorem holds. \square

Next, consider the case where function \mathcal{Q} or \mathcal{E} calls a sub-function. There are three cases to be considered: (Case 4) function \mathcal{Q} is a recursion of function \mathcal{S} , (Case 5) function \mathcal{E} is a recursion of function \mathcal{S} , and (Case 6) the remained cases.

Case 4 ($q = p$)

Consider the case where function \mathcal{Q} is a recursion of function \mathcal{S} . The equivalence of the results are determined as follows.

Lemma 5 $\mathcal{Q}[p]x:D_i \equiv \mathcal{Q}[p]x:(D_i + \mathcal{W})$ if there are a node v in $\mathcal{S}[p]x:D_i$ and a node v' in $\mathcal{S}[p]x:(D_i + \mathcal{W})$ such that $v \equiv v'$ and v and v' are not nondetermined.

proof: In the case of $\mathcal{Q}[p]x \leftarrow \mathcal{S}[p]x$, the boolean value of $\mathcal{Q}[p]x$ equals to 1 iff the result node-list of $\mathcal{S}[p]x$ is not empty. Thus, if nodes v and v' such that $v \equiv v'$ and those nodes are not nondetermined respectively exist in $\mathcal{S}[p]x:D_i$ and $\mathcal{S}[p]x:(D_i + \mathcal{W})$, $\mathcal{Q}[p]x:D_i = \mathcal{Q}[p]x:(D_i + \mathcal{W}') = 1$ for any writes \mathcal{W}' by a combination of transactions. By Definitions 3 and 4, the lemma holds. \square

Case 5 ($e = p$)

Consider the case where function \mathcal{E} is a recursion of function \mathcal{S} . The equivalence of the results are determined as follows.

Lemma 6 $\mathcal{E}[p]x:D_i \equiv \mathcal{E}[p]x:(D_i + \mathcal{W})$ if the first node v in $\mathcal{S}[p]x:D_i$ and the first node v' in $\mathcal{S}[p]x:(D_i + \mathcal{W})$ are equivalent and v and v' are not nondetermined.

proof: In the case of $\mathcal{E}[p]x \leftarrow \mathcal{S}[p]x$, the numerical value of $\mathcal{E}[p]x$ is determined from the string-value of the first node in the node-list of $\mathcal{S}[p]x$. Thus, if the first nodes in $\mathcal{S}[p]x:D_i$ and $\mathcal{S}[p]x:(D_i + \mathcal{W})$ are equivalent and are not nondetermined, $\mathcal{E}[p]x:D_i = \mathcal{E}[p]x:(D_i + \mathcal{W})$ for

any writes \mathcal{W}' by a combination of transactions. By Definitions 3 and 4, the lemma holds. \square

Case 6 ($q = q_1 \theta q_2, e = e_1 \theta e_2, e = \theta e_1$
 $\theta \in \{\text{and, or, +, -, } \dots\}$)

Consider the case where function \mathcal{Q} or \mathcal{E} calls sub-functions \mathcal{Q}' or \mathcal{E}' , e.g. $q = q_1 \text{ or } q_2$ and $e = e_1 + e_2$. In such cases, the equivalence of the results is determined as follows.

Lemma 7 $\mathcal{F}[\epsilon]x:D_i \equiv \mathcal{F}[\epsilon]x:(D_i + \mathcal{W})$ if for any sub-function \mathcal{F}' such that $\mathcal{F} \leftarrow \mathcal{F}'$, $\mathcal{F}'[\epsilon']x:D_i \equiv \mathcal{F}'[\epsilon']x:(D_i + \mathcal{W})$.

proof: A sub-function \mathcal{F}' is either \mathcal{Q} or \mathcal{E} , and the equivalence for \mathcal{F}' is determined by the conditions in Lemma 5 and/or Lemma 6. Obviously, $\mathcal{F}[\epsilon]x(D_i) = \mathcal{F}[\epsilon]x(D_i + \mathcal{W})$ for any writes \mathcal{W}' by a combination of transactions if $\mathcal{F}'[\epsilon']x:D_i \equiv \mathcal{F}'[\epsilon']x:(D_i + \mathcal{W})$ for any sub-function \mathcal{F}' . \square

Example 4 We illustrate how the equivalence is determined using the example cases in Example 3. Consider read $r_1 = \text{read}(\epsilon)$ and writes W_2, W_3, W'_2 and W'_3 in Example 3 and the evaluation tree of ϵ in Fig. 5. Let D_i be the document shown in Fig. 2. Then $\mathcal{F}[\epsilon]x:D_i (= \mathcal{S}_0[\epsilon]x:D_i)$ is recursively computed using the evaluation tree in Fig. 5 and the result of $\mathcal{S}_0[\epsilon]x:D_i$ is the node-list containing a node, v , whose child is labeled “Mary”. First, consider writes W_2 and W_3 , and let $\mathcal{W} = W_2 + W_3$. The result of $\mathcal{S}_0[\epsilon]x:(D_i + \mathcal{W})$ is an empty node-list. Since no equivalent node to node v exists in $\mathcal{S}_0[\epsilon]x:(D_i + \mathcal{W})$, node v in $\mathcal{S}_0[\epsilon]x:D_i$ is a nondetermined node. By Lemma 4, it is determined that $\mathcal{S}_0[\epsilon]x:D_i \not\equiv \mathcal{S}_0[\epsilon]x:(D_i + \mathcal{W})$, and thus the conflict between r_1 and $W_2 + W_3$ is detected. Next, consider writes W'_2 and W'_3 , and let $\mathcal{W} = W'_2 + W'_3$. In this case, the equivalence for functions in the area surrounded by a dotted line in Fig. 2 does not hold. For example, consider $\mathcal{S}_7[\text{balance}]x':D_i$ and $\mathcal{S}_7[\text{balance}]x':(D_i + \mathcal{W})$ called in the evaluation. $\mathcal{S}_7[\text{balance}]x':D_i \not\equiv \mathcal{S}_7[\text{balance}]x':(D_i + \mathcal{W})$ since nodes in the results for \mathcal{S}_7 with D_i and $D_i + \mathcal{W}$ are not equivalent each other. Then the equivalence for function \mathcal{E}_2 does not hold by Lemma 6. Similarly, the equivalence for other functions are determined and it is finally determined that $\mathcal{S}_0[\epsilon]x:D_i \not\equiv \mathcal{S}_0[\epsilon]x:(D_i + \mathcal{W})$ although $\mathcal{S}_0[\epsilon]x:D_i = \mathcal{S}_0[\epsilon]x:(D_i + \mathcal{W})$. Thus the conflict between r_1 and W'_2 is detected. \square

We now discuss the cost of the proposed equivalence checking in the XPath evaluation

for conflict detection. The nondetermination of result nodes from the context node in the evaluation of \mathcal{S} and the equivalence of results in the evaluation of \mathcal{Q} and \mathcal{E} are determined with a little effort. On the other hand, for each location step in the evaluation of \mathcal{S} , checking the equivalence of nodes in the different node-lists is needed and it takes a proportional overhead to the number of steps contained in the expression.

We show that the number of equivalence checks needed for evaluating location steps can be reduced as follows: Consider again a location path $p = p_1/p_2/\dots/p_m$. Assume that for some $l(1 \leq l < m)$, $\mathcal{S}[[p_l]x:D_i] \neq \mathcal{S}[[p_l]x:(D_i + \mathcal{W})]$. Then there is a node v in $\mathcal{S}[[p_l]x:D_i$ or $\mathcal{S}[[p_l]x:(D_i + \mathcal{W})]$ that has no equivalent node in the other node-list. In the case of $v \in D_i$, it means that the node equivalent to v in $D_i + \mathcal{W}$ was deleted by \mathcal{W} . Then no nodes equivalent to nodes in a subtree of v exist in $D_i + \mathcal{W}$. In the case of $v \in D_i + \mathcal{W}$, it means that node v is inserted by \mathcal{W} . Then no nodes equivalent to nodes in a subtree of v exist in D_i . Therefore, if the next step p_{l+1} has an axis for searching nodes in the subtree of the context node (let us refer to such an axis as a *down-axis*), the nodes selected by p_{l+1} using a non-equivalent result node of the previous step as a context node are also non-equivalent nodes. Therefore, for each step p_l , the equivalence check for nodes in $\mathcal{S}[[p_l]x:D_i]$ and $\mathcal{S}[[p_l]x:(D_i + \mathcal{W})]$ can be omitted if the axis of the next step p_{l+1} is a down-axis. In other words, the equivalence check for nodes is needed for only the step whose next step has not a down-axis. The denotational semantics of XPath with the proposed equivalence checking is presented in the appendix of this paper.

6. Conflict Check

In this section, we propose the mechanisms for the read-write check and the write-read check in XPCC. The proposed mechanisms are based on the XPath evaluation with equivalence checking presented in Section 5.

6.1 Read-Write Check

This section describes the proposed method for the read-write check. When a read is requested, the read-write check detects any read-write conflict caused by the read requested.

By Definitions 3 and 4, a conflict between a

Such axes in XPath are child, self, descendant, descendant-or-self, attribute and namespace.

read $r_i = read(\epsilon)$ requested by transaction T_i and any previous writes of other transactions can be detected by checking formula 5.5 with documents D_i and $D_i + \mathcal{W}$ where \mathcal{W} is a sequence of all writes, i.e. $\mathcal{W} = W_1 + \dots + W_l + \dots + W_n (l \neq i)$. In our document management, the updates of all previous writes are contained in D_{all} , that is, $D_{all} = D_i + \mathcal{W}$. Theorem 2 then holds.

Theorem 2 If a read $r_i = read(\epsilon)$ causes a read-write conflict with any previous writes by other transactions,

$$\mathcal{F}[\epsilon]x:D_i \not\equiv \mathcal{F}[\epsilon]x:D_{all}. \quad (6.1)$$

□

The proposed algorithm for the read-write check is then as follows: At the time that a transaction T_i requests a read $read(\epsilon)$,

- (1) Check formula 6.1 using the proposed XPath evaluation with equivalence checking.
- (2) If the formula does not hold, r_i causes no read-write conflict and thus T_i proceeds.
- (3) If the formula holds, r_i causes a read-write conflict with other transactions. Find transactions conflict with r_i , and block T_i until such transactions finish.

When r_i causes a read-write conflict, r_i may conflict with a set of transactions even if r_i does not conflict with each of the transactions. Several solutions to handle this problem can be considered although we omit details here. One simple solution to be considered is blocking T_i until all other transactions finish. This is the easiest way but causes a long waiting-time. The other solution is checking the equivalence against $W_1, W_1 + W_2, \dots, W_1 + \dots + W_n$, one after another. For example, if it is checked that $\mathcal{F}[\epsilon]x:D_i \not\equiv \mathcal{F}[\epsilon]x:(D_i + \mathcal{W}')$ for $\mathcal{W}' = W_1 + \dots + W_j$, the set of transactions that conflicts with r_i is a subset of $\{T_1, \dots, T_j\}$. The conflict of r_i is then prevented by blocking r_i until transactions T_1, \dots, T_j finish.

In the read-write check, XPath evaluation is processed for both D_i and D_{all} . The execution cost needed for the proposed read-write check is then the sum of the cost for evaluating an XPath expression and the cost for equivalence checks in the XPath evaluation. Equivalence checks for nodes can be easily implemented by, for instance, sharing a pointer between equivalent nodes in distinct documents in our document management model.

6.2 Write-Read Check

This section describes the proposed method

for the write-read check. When a write is requested, the write-read check detects any write-read conflict caused by the write requested.

Consider a write w_i requested by transaction T_i and a previous read $r_j = read(\epsilon)$ by transaction T_j . Let D'_j be the previous state of document D_j to that read r_j was performed. In addition, let W_i be a sequence of all writes of T_i including w_i . By Definitions 3 and 4, if write w_i causes a write-read conflict with read r_j , then

$$\mathcal{F}[\epsilon]x:D'_j \not\equiv \mathcal{F}[\epsilon]x:(D'_j + W_i + \mathcal{W}') \quad (6.2)$$

where $\mathcal{W}' = W_1 + \dots + W_l + \dots + W_n (l \neq i, j)$. There is no conflict between writes of W_i and any set of writes of \mathcal{W}' since such conflicts are detected by previous conflict checks. The following lemma and theorem then hold.

Lemma 8 If a write w_i requested by T_i causes a write-read conflict with a previous read $r_j = read(\epsilon)$,

$$\mathcal{F}[\epsilon]x:D'_j \not\equiv \mathcal{F}[\epsilon]x:(D'_j + W_i). \quad (6.3)$$

proof: There are two cases to be considered. First, consider the case where read r_j conflicts with writes of W_i . By Lemma 1, formula 6.3 then holds. Next, consider the case where r_j conflicts with the set of writes of W_i and a combination of writes in \mathcal{W}' . Assume that $\mathcal{F}[\epsilon]x:D'_j \equiv \mathcal{F}[\epsilon]x:(D'_j + W_i)$. Then $\mathcal{F}[\epsilon]x:D'_j \not\equiv \mathcal{F}[\epsilon]x:(D'_j + \mathcal{W}')$ because formula 6.2 holds and there is no conflict between writes of W_i and any set of writes of \mathcal{W}' . This is a contradiction since such a nonequivalence is detected in previous conflict checks. Therefore the lemma holds. □

Theorem 3 If write w_i requested by T_i causes a write-read conflict with a previous read $r_j = read(\epsilon)$,

$$\mathcal{F}[\epsilon]x:(D_{st} + W'_j) \not\equiv \mathcal{F}[\epsilon]x:(D_{st} + W'_j + W_i) \quad (6.4)$$

where W'_j denotes a sequence of all writes of T_j before r_j and W_i denotes a sequence of all writes of T_i including w_i .

proof: Recall that D_{st} is the document state committed. If $D'_j = D_{st} + W'_j$ with the previous state D'_j of D_j at the time of r_j , formula 6.4 obviously holds. If $D'_j \neq D_{st} + W'_j$, then D_{st} contains the updates of writes \mathcal{W}' by transactions committed after T_j started. Since r_j does not conflict with any writes in \mathcal{W}' , $\mathcal{F}[\epsilon]x:(D'_j) \equiv \mathcal{F}[\epsilon]x:((D'_j + \mathcal{W}') = (D_{st} + W'_j))$. Formula 6.3 is then equivalent to formula 6.4. Thus the theorem holds. □

By Theorem 3, a conflict between the re-

requested write w_i and a previous read r_j can be detected by checking formula 6.4. For any previous read of other transactions, we need to check whether the write requested leads to a write-read conflict.

The proposed algorithm for the write-read check is as follows: At the time that a transaction T_i requests a write w_i , for each transaction $T_j (i \neq j, 1 \leq j \leq n)$,

- (1) Prepare two documents $D'_i (= D_i + w_i)$ and D which is a copy of D_{st} .
- (2) Trace each access in the access sequence of T_j step by step.
 - (a) If the access is a write, update D and D'_i by the write.
 - (b) If the access is a read $r_j = read(\epsilon)$, check if $\mathcal{F}[\epsilon]x:D \equiv \mathcal{F}[\epsilon]x:D'_i$. If so, then w_i does not conflict with r_j , and thus go to (2). Otherwise, w_i causes a conflict with R_j , and thus block T_i until T_j finishes.

By tracing the access sequence of each transaction $T_j (i \neq j, 1 \leq j \leq n)$ once, the write-read check finishes.

To evaluate the computing cost needed for the proposed write-read check, we define the following notations:

- nr_j : the number of reads in the access sequence of T_j .
- nw_j : the number of writes in the access sequence of T_j .
- cr_j : the cost of performing the equivalence checking.
- cw_j : the cost of performing a write.

When a write is requested, the execution cost needed for the write-read check based on tracing access sequences is equal to

$$\sum_{1 \leq j \leq n, i \neq j} (nr_j \times cr_j + nw_j \times cw_j).$$

Here we proposed the algorithm for the write-read check that uses previous states of D_i 's generated by tracing access sequences of transactions. If multi-version documents are adopted in the document management, regenerating previous states of documents is not necessary and thus the cost for the write-read check can be reduced. We also have an idea for a more efficient write-read check using document D_{all} and its previous states. An improved algorithm for the write-read check was proposed in Ref. 4).

7. Conclusion

In this paper, we proposed the concurrency

control method XPCC guaranteeing serializability in XML document management, assuming concurrent XPath accesses and updates to the same XML documents. XPCC enforces data consistency in XML semantics and achieves great concurrency by the conflict check and locking at the level of precise data in XML documents. In XPCC, conflicts between XPath expressions used in reads and updates of writes by different transactions are detected based on the XPath evaluation with equivalence checking. In order to check conflicts efficiently, we first introduced the new document management model, which handles two versions of a document, document D_i containing the updates of each transaction T_i and document D_{all} containing the updates of all transactions. We next proposed the mechanism of the XPath evaluation with equivalence checking which can detect conflicts while processing the evaluation of an XPath expression on two versions of a document. Under our document management model, conflict checks are performed when each transaction access is requested. As for the read-write check for each read requested by transaction T_i , the XPath expression used in the read is evaluated on both documents D_i and D_{all} and the equivalence of the results is checked. As for the write-read check for each write requested, the conflict check is performed against every previous read while tracing access sequences of transactions.

The weakness of XPCC is that the execution cost for the write-read check seems relatively high although the execution cost for the read-write check is low. Obviously, the cost for the conflict check could be reduced by transaction ordering or depending on isolation levels of transactions or simply the number of write accesses. Analysis of trade-off between the isolation level and the cost of conflict checks is an interesting future subject. In addition, since XPCC handles local copies of the document for each transactions similar to optimistic concurrency control, it involves memory overhead when executing many transactions concurrently. To reduce the memory overhead, we include the implementation of local copies virtually on one document by identifying the transaction that updated each node in the document in future research. The implementation of XPCC and the quantitative analysis of execution times and memory sizes needed for XPCC should also be included in future works.

As our future study, we will also improve XPCC in order to realize concurrency control with less overhead in conflict checks so as to build a sophisticated XML data management system.

Acknowledgments Authors would like to thank anonymous reviewers for their constructive comments which helped in improving the quality of this paper. Authors also would like to thank Prof. Kaname Harumoto, the editor in charge, for his invaluable comments to revise our manuscript.

References

- 1) Bourret, R.: XML and databases, Internet Document (Feb. 2002).
<http://www.rpbouret.com/xml/XMLAndDatabases.htm>
- 2) Bray, T., Paoli, J. and Sperberg-McQueen, C.M.: Extensible markup language (XML) 1.0., *W3C Recommendation* (Feb. 1998).
<http://www.w3.org/XML>
- 3) Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J. and Simeon, J.: XQuery 1.0: An XML Query Language, *W3C Working Draft* (Aug. 2002).
<http://www.w3.org/XML/Query>
- 4) Choi, E. and Kanai, T.: XPath-based concurrency control for XML data, *Proc. IEICE 14th Data Engineering Workshop (DEWS)*, 6-C-4 (Apr. 2003).
- 5) Clark, J. and DeRose, S.: XML Path Language (XPath) 1.0., *W3C Recommendation* (Nov. 1999).
<http://www.w3.org/TR/xpath>
- 6) Eswaran, K.P., Gray, J., Lorie, R. and Traiger, I.: The notions of consistency and predicate locks in a database systems, *Comm. ACM*, Vol.19, No.11, pp.624–633 (Nov. 1976).
- 7) Grabs, T., Böhm, K. and Schek, H.: XMLTM: efficient transaction management for XML documents, *Proc. 19th International Conference on Information and Knowledge Management (CIKM)*, pp.142–152 (Nov. 2002).
- 8) Goldman, R. and Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases, *Proc. 23rd VLDB Conference*, pp.436–445 (Aug. 1997).
- 9) Gray, P. and Reuter, A.: Transaction processing: concepts and technology, Morgan Kaufmann (1993).
- 10) Helmer, S., Kanne, C.-C. and Moerkotte, G.: Isolation in XML bases, Technical Report, University of Mannheim, Germany (2001).
- 11) Jea, K.F.: Concurrency control in XML document databases: XPath locking protocol, *Proc. 9th International Conference on Parallel and Distributed Systems (ICPADS)* (Dec. 2002).
- 12) Jordan, J.R., Banerjee, J. and Batman, R.B.: Precision locks, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.143–147 (Apr. 1981).
- 13) Lomet, D.B.: Key range locking strategies for improved concurrency, *Proc. 19th VLDB Conference*, pp.655–664 (Aug. 1993).
- 14) Mohan, C.: ARIES/KVL: A key-value locking method for concurrency control of multi-action transactions operating on B-tree indexes, *Proc. 16th VLDB Conference*, pp.392–405 (Aug. 1990).
- 15) Olteanu, D., Meuss, H., Furche, T. and Bry, F.: XPath: Looking forward, *Proc. Workshop on XML Data Management (XMLDM)* (March 2002).
- 16) Tatarinov, I., Ives, Z.G., Halevy, A.Y. and Weld, D.S.: Updating XML, *Proc. ACM SIGMOD International Conference on Management of Data*, pp.413–424 (May 2001).
- 17) Wadler, P.: Two semantics for XPath, Technical report (Jan. 2000).
<http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics>

Appendix: Semantics of XPath with Equivalence Checking

Here we present the semantics of XPath that can perform the equivalence checking for the proposed conflict check. For convenience, we refer to the XPath with equivalence checking as *eXPath*. We consider the abstract syntax described in Fig. 4 as the syntax of *eXPath*, but that can be easily extended.

The denotational semantics of *eXPath* is specified by three functions \mathcal{S} , \mathcal{Q} and \mathcal{E} with two arguments, an expression and a pair of contexts $X = (X_1, X_2)$, which is described in **Figs. 8** and **9**. Note that the semantics of *eXPath* holds two contexts to be compared, while the general semantics of XPath is specified by similar functions with only one context. The context for function \mathcal{S} is defined as $X_i = (x_i, u_i)$ with $i = 0, 1$ where x_i is the context node and u_i is a boolean value representing the nondetermination of the context node. The context for functions \mathcal{Q} and \mathcal{E} is defined as $X_i = (x_i, pos_i, size_i, u_i)$ with $i = 0, 1$ where x_i , pos_i , $size_i$ and u_i respectively denote the context node, the context position, the context size and the nondetermination of the context node. As for the context, the difference between the general XPath and our *eXPath* is then that boolean value u_i is added to check whether the

X	$= ((x_0, u_0), (x_1, u_1))$
	<i>UnionExpr</i>
S	$= S[p_1 p_2]X$
S_{R_i}	$= S_{R_i}[p_1]X \cup S_{R_i}[p_2]X$
S_{U_i}	$= \{u_{i,j} (1 \leq j \leq S_{R_i}) \mid u_{i,j} = u_i \vee \neg((x_{i,j} \in S_{R_i}[p_1]X \wedge \exists x \in S_{R_i}[p_1]X, x \equiv x_{i,j}) \vee (x_{i,j} \in S_{R_i}[p_2]X \wedge \exists x \in S_{R_i}[p_2]X, x \equiv x_{i,j}))\}$
	<i>AbsoluteLocationPath</i>
S	$= S[/p]X$
S_{R_i}	$= S_{R_i}[p]((\text{root}(x_0), u_0), (\text{root}(x_1), u_1))$
S_{U_i}	$= S_{U_i}[p]((\text{root}(x_0), u_0), (\text{root}(x_1), u_1))$
	<i>RelativeLocationPath</i>
S	$= S[p_1/p_2]X$
S_{R_i}	$= \{x \mid x' \in S_{R_i}[p_1]X, \text{ if } (\exists x'' \in S_{R_i}[p_1]X, x' \equiv x''), x \in S_{R_i}[p_2]((y_0, u_0), (y_1, u_1)) \text{ where if } i = 0, y_0 = x' \text{ and } y_1 = x''; \text{ otherwise, } y_0 = x'' \text{ and } y_1 = x'; \text{ otherwise, } x \in S_{R_0}[p_2]((x', 1), \emptyset)\}$
S_{U_i}	$= \{u_{i,j} (1 \leq j \leq S_{R_i}) \mid u_{i,j} = u_i \vee \neg((\exists x \in S_{R_0}[p_1]X, \exists x' \in S_{R_1}[p_1]X, x \equiv x') \wedge (x_{i,j} \in S_{R_i}[p_2]((x, u_0), (x', u_1))))\}$
	<i>LocationStep</i>
S	$= S[p[q]]X$
S_{R_i}	$= \{x \mid x \in S_{R_i}[p]X \text{ let } pos_i = \{x_p \mid x_p \in S_{R_i}[p]X, x_p \leq_{order} x\} , \text{ let } size_i = S_{R_i}[p]X \text{ if } (\exists x' \in S_{R_i}[p]X, x \equiv x'), \text{ let } pos_{\bar{i}} = \{x_p \mid x_p \in S_{R_{\bar{i}}}[p]X, x_p \leq_{order} x'\} , \text{ let } size_{\bar{i}} = S_{R_{\bar{i}}}[p]X \text{ if } i = 0, \mathcal{Q}_{R_i}[q](Y, Y') = 1, \text{ otherwise } \mathcal{Q}_{R_i}[q](Y', Y) = 1 \text{ where } Y = (x, pos_i, size_i, u_i), Y' = (x', pos_{\bar{i}}, size_{\bar{i}}, u_{\bar{i}}); \text{ otherwise, } \mathcal{Q}_{R_0}[q]((x, pos_i, size_i, 1), \emptyset) = 1\}$
S_{U_i}	$= \{u_{i,j} (1 \leq j \leq S_{R_i}) \mid u_{i,j} = u_i \vee (\forall x \in S_{R_{\bar{i}}}, x \not\equiv x_{i,j}) \vee (\exists x \in S_{R_{\bar{i}}}, x \equiv x_{i,j}, \mathcal{Q}_u[q]((x_{i,j}, pos_i, size_i, u_i), (x, pos_{\bar{i}}, size_{\bar{i}}, u_{\bar{i}})) = 1)\}$
S	$= S[p[e]]X$
S_{R_i}	$= \{x \mid x \in S_{R_i}[p]X \text{ let } pos_i = \{x_p \mid x_p \in S_{R_i}[p]X, x_p \leq_{order} x\} , \text{ let } size_i = S_{R_i}[p]X \text{ if } (\exists x' \in S_{R_i}[p]X, x \equiv x'), \text{ let } pos_{\bar{i}} = \{x_p \mid x_p \in S_{R_{\bar{i}}}[p]X, x_p \leq_{order} x'\} , \text{ let } size_{\bar{i}} = S_{R_{\bar{i}}}[p]X \text{ if } i = 0, \mathcal{E}_{R_i}[e](Y, Y') = pos_i, \text{ otherwise } \mathcal{E}_{R_i}[e](Y', Y) = pos_{\bar{i}} \text{ where } Y = (x, pos_i, size_i, u_i), Y' = (x', pos_{\bar{i}}, size_{\bar{i}}, u_{\bar{i}}); \text{ otherwise, } \mathcal{E}_{R_0}[e]((x, pos_i, size_i, 1), \emptyset) = pos_i\}$
S_{U_i}	$= \{u_{i,j} (1 \leq j \leq S_{R_i}) \mid u_{i,j} = u_i \vee (\exists l, 1 \leq l \leq j, x_{0,l} \not\equiv x_{1,l}) \vee (\mathcal{E}_u[e]((x_{0,j}, j, size_0, u_0), (x_{1,j}, j, size_1, u_1)) = 1)\}$
S	$= S[axis :: \text{nodetest}]X$
S_{R_i}	$= \{x \mid x \in \mathcal{A}[axis]x, x \text{ satisfies } \text{nodetest}\}$
S_{U_i}	$= \{u_{i,j} (1 \leq j \leq S_{R_i}) \mid u_{i,j} = u_i \vee (S_{R_{\bar{i}}} = \emptyset)\}$

* When $i = 0$ and $1, \bar{i} = 1$ and 0 , respectively.

** $\mathcal{A}[a]x$ is an auxiliary function which denotes the node-list generated by axis a from context node x .

*** $x <_{order} x'$ with two nodes x and x' in a node-list denotes that x is previous to x' in the node-list.

Fig. 8 Semantics of S .

$X = ((x_0, pos_0, size_0, u_0), (x_1, pos_1, size_1, u_1))$ <p style="text-align: center;"><i>OrExpr, AndExpr, EqualityExpr, RelationalExpr</i></p> $\begin{aligned} \mathcal{Q} &= \mathcal{Q}[q_1 \oplus q_2]X, \oplus : \text{and, or, =, ! =, <, >, <=, >=} \\ \mathcal{Q}_{R_i} &= \mathcal{Q}_{R_i}[q_1]X \oplus \mathcal{Q}_{R_i}[q_2]X \\ \mathcal{Q}_u &= \mathcal{Q}_u[q_1]X \vee \mathcal{Q}_u[q_2]X \end{aligned}$ <p style="text-align: center;"><i>Pattern</i></p> $\begin{aligned} \mathcal{Q} &= \mathcal{Q}[p]X \\ \mathcal{Q}_{R_i} &= \text{let } \mathcal{S} = \mathcal{S}[p]((x_0, u_0)(x_1, u_1)), \text{ 1 iff } \mathcal{S}_{R_i} \neq \emptyset \\ \mathcal{Q}_u &= \text{1 iff for any } u_{i,j} \in \mathcal{S}_{U_i}, u_{i,j} = 1 \end{aligned}$ <p style="text-align: center;"><i>Expression</i></p> $\begin{aligned} \mathcal{Q} &= \mathcal{Q}[e]X \\ \mathcal{Q}_{R_i} &= \text{1 iff } \mathcal{E}_{R_i}[e]X \neq 0; \\ \mathcal{Q}_u &= \mathcal{E}_u[e]X \end{aligned}$ <p style="text-align: center;"><i>AdditiveExpr, MultiplicativeExpr</i></p> $\begin{aligned} \mathcal{E} &= \mathcal{E}[e_1 \oplus e_2]X, \oplus : +, -, *, \text{div, mod} \\ \mathcal{E}_{R_i} &= \mathcal{E}_{R_i}[e_1]X \oplus \mathcal{E}_{R_i}[e_2]X \\ \mathcal{E}_u &= \mathcal{E}_u[e_1]X \vee \mathcal{E}_u[e_2]X \end{aligned}$ <p style="text-align: center;"><i>UnaryExpr</i></p> $\begin{aligned} \mathcal{E} &= \mathcal{E}[-e]X \\ \mathcal{E}_{R_i} &= -\mathcal{E}_{R_i}[e]X \\ \mathcal{E}_u &= \mathcal{E}_u[e]X \end{aligned}$ <p style="text-align: center;"><i>Pattern</i></p> $\begin{aligned} \mathcal{E} &= \mathcal{E}[p]X \\ \mathcal{E}_{R_i} &= \text{let } \mathcal{S} = \mathcal{S}[p]((x_0, u_0)(x_1, u_1)), \text{ number}(x_{i,1}) (x_{i,1} \in \mathcal{S}_{R_i}) \text{ if } \mathcal{S}_{R_i} \neq \emptyset; \text{ otherwise, } 0 \\ \mathcal{E}_u &= u_{0,1}(\in \mathcal{S}_{U_0}) \vee u_{1,1}(\in \mathcal{S}_{U_1}) \end{aligned}$ <p style="text-align: center;"><i>Qualifier</i></p> $\begin{aligned} \mathcal{E} &= \mathcal{E}[q]X \\ \mathcal{E}_{R_i} &= \text{1 if } \mathcal{Q}_{R_i}[q]X \neq 0; \text{ otherwise, } 0 \\ \mathcal{E}_u &= \mathcal{Q}_u[q]X \end{aligned}$	
--	--

Fig. 9 Semantics of \mathcal{Q} and \mathcal{E} .

context node is nondetermined or not.

For each function $\mathcal{F}(=\mathcal{S}, \mathcal{Q}, \mathcal{E})$, we write $\mathcal{F}_{R_i}[\epsilon]X$ ($i = 0, 1$) for the result value of expression ϵ evaluated with context X_i . That is, $\mathcal{S}_{R_i}[p]X$ denotes the node-list selected by pattern p , $\mathcal{Q}_{R_i}[q]X$ denotes the boolean value of qualifier q , and $\mathcal{E}_{R_i}[e]X$ denotes the numerical value of expression e . The equality of the results associated with two contexts X_0 and

X_1 is defined as follows: In the cases of \mathcal{Q} and \mathcal{E} , $\mathcal{F}_{R_0}[\epsilon]X = \mathcal{F}_{R_1}[\epsilon]X$ iff the (boolean or numerical) values are same. In the case of \mathcal{S} , $\mathcal{S}_{R_0}[p]X = \mathcal{S}_{R_1}[p]X$ iff $|\mathcal{S}_{R_0}[p]X| = |\mathcal{S}_{R_1}[p]X|$ and for every pair of the j -th nodes $x_{0,j}(\in \mathcal{S}_{R_0}[p]X)$ and $x_{1,j}(\in \mathcal{S}_{R_1}[p]X)$, node $x_{0,j}$ is equivalent to node $x_{1,j}$. Hereafter, $x_{i,j}(\in \mathcal{S}_{R_i}[p]X)$ denotes the j -th node in result node-list $\mathcal{S}_{R_i}[p]X$.

The equivalence of the results associated with two contexts X_0 and X_1 is determined depending on each function as explained in Section 5.3. For each function \mathcal{F} , $\mathcal{F}_u[\epsilon]X$ denotes a boolean value representing the equivalence of $\mathcal{F}_{R_0}[\epsilon]X$

In Ref. 17), notions $\mathcal{S}[p]x_i$, $\mathcal{Q}[q](x_i, pos_i, size_i)$ and $\mathcal{E}[e](x_i, pos_i, size_i)$ are used to indicate the result values of p , q and e with context node x_i . Note that $\mathcal{S}_{R_i}[p]X = \mathcal{S}[p]x_i$, $\mathcal{Q}_{R_i}[q]X = \mathcal{Q}[q](x_i, pos_i, size_i)$ and $\mathcal{E}_{R_i}[e]X = \mathcal{E}[e](x_i, pos_i, size_i)$.

and $\mathcal{F}_{R_i} \llbracket \epsilon \rrbracket X$, i.e., $\mathcal{F}_u \llbracket \epsilon \rrbracket X = 1$ iff $\mathcal{F}_{R_0} \llbracket \epsilon \rrbracket X \equiv \mathcal{F}_{R_1} \llbracket \epsilon \rrbracket X$. In addition, for each function \mathcal{S} , $\mathcal{S}_{U_i} \llbracket p \rrbracket X$ denotes a list whose element $u_{i,j}$ is a boolean value representing the nondetermination of node $x_{i,j}$ in $\mathcal{S}_{R_i} \llbracket p \rrbracket X$. This means that $u_{i,j} = 1$ iff node $x_{i,j}$ is nondetermined. Thus $\mathcal{S}_u \llbracket p \rrbracket X = 1$ iff there is no $u_{i,j} \in \mathcal{S}_{U_i} \llbracket p \rrbracket X$ with $i = 0, 1$ such that $u_{i,j} = 1$, by Lemma 4.

Figures 8 and 9 describe the details of the semantics for function \mathcal{S} and those for functions \mathcal{Q} and \mathcal{E} respectively. Using the proposed semantics of eXPath, one can efficiently perform conflict detection associated with two document states and an XPath expression by checking the equivalence of the results evaluated.

(Received June 20, 2003)

(Accepted October 2, 2003)

(*Editor in Charge* Kaname Harumoto)



Eun Hye Choi received M.E. and Ph.D. degrees in computer engineering from Osaka University in 1999 and in 2002, respectively. She has been worked for Toshiba Corp. from 2002 to 2003. Her current research interests are in the areas of XML database and transaction processing and distributed computing. She is a member of the IEEE.



Tatsunori Kanai received his B.E. and M.E. degrees in Information Science from Kyoto University in 1984 and 1986 respectively. He has been working in Toshiba Corp. since 1989 and now is a senior research scientist of Corporate Research and Development Center of Toshiba. His research interests include computer architecture, online transaction processing and information architecture. He is a member of IPSJ and IEICE.