Indian Journal of Science and Technology, *Vol 7(2), 151–157, February 2014*

ISSN (Print): 0974-6846
ISSN (Online) : 0974-5645

A New Approach to Concurrency Control in XML Databases

Sayyed Kamyar Izadi[1*] and Mostafa Haghjoo[2]

[1]Department of Computer Science, Shahid Beheshti University, Tehran, Iran; Sk_Izadi@sbu.ac.ir
[2]Faculty of Computer Engineering, Iran University of Science & Technology, Tehran, Iran; haghjoom@iust.ac.ir

Abstract

One of the most important features of a database in a multi-user environment is its concurrency control mechanism. The existing protocols either provide a restricted concurrency level which is less than what logically could be or provide a high level of concurrency which may lead to some defects. To overcome these problems, we offer a novel locking protocol with some rich locks named "XML Path Locking by Child Consideration" (XPLC). In our approach unlike the existing ones, we consider the child of the node which we want to lock. Also our locks have different granularities according to their types.

Keywords: Concurrency Control, Transaction, XML Database, XPLC

1. Introduction

With the growing popularity of XML, it becomes an important format for exchanging and storing semistructured data. It is now widely used in many applications such as science, biology, business and particularly web applications. Managing huge data stored in XML documents, emphasizes the need for native XML databases[1–4].

Native XML databases support all of the features which are found in traditional database systems. Concurrency Control is one of the most important features of a database in multi-user environments. As a result some concurrency control protocols have been developed for XML databases so far. These protocols can be classified as XPath or DOM model based[5].

Some of the current protocols have constraints which lead to a restricted concurrency level while other protocols have some integrity defects like phantom problem. Moreover improper granularity assigned to the locks, results in using more locks while the transaction is scheduled.

Most of the XPath-based protocols use different types of path locking. XPath Locking Protocol[4] considers two types of operations: one type modifies the structure of the document and the other one keeps the structure without any change. In compatibility matrix it assumes that the locks for different types of operations are compatible with each other and the conflict happens between the locks used for the same types of operations. For an example it says read and insert locks are compatible because one operation is structural and another is unstuctural. But this may lead to phantom problem. In fact, defining compatibility matrix in this way is basically wrong. Furthermore, in this approach the set of pass-by nodes in a path from root to destination nodes in each step are locked by P-lock if they do not conflict with the locks held by other transactions, but after passing this step, all P-locks on the nodes which have been accessed in this step are released. Read lock (R-lock), write lock (W-lock), insertion lock (I-lock) and deletion lock (D-lock) are just applied to destination nodes for reading, writing, inserting and deleting. This locking method causes a potential risk of deleting a subtree by a transaction while other transactions are executing read or write operations on the nodes in lower levels of that subtree which in effect may lead to an integrity defect.

In XPath-based concurrency control[6], every active transaction uses a copy of the main document to execute its updates in it and to check the conflict between the locks of different transactions. Also a copy of document

*Author for correspondence

is produced which contains all updates of all active transactions. When a transaction commits, all the updates done by it are reflected to the main document. So when there are n active transactions, n+1 copy of the main document should be produced. Consequently this is an expensive method and has high overhead.

In Sedna locking method it is claimed that the subtrees can be locked without locking the ancestors of the root in intention mode[7]. In order to achieve this goal it uses a labeling algorithm but does not define its mechanism of assigning labels to the nodes of XML document. Also like previous methods, it doesn't allow two transactions concurrently execute two insert operations under a parent, although it is logically possible and does not lead to any fault.

In the protocol proposed by Dekeyser and Hidders[5,8,9], update operation need to be simulated by sequence of simple delete and insert, so it needs more commands and locks. It is claimed that two write operations can be done concurrently on the same node while assuming writing always implies reading. By considering these assumptions beside the fact that read and write locks are in conflict, it is impossible to execute two concurrent write operations on a node by different transactions, because implicit read of the second write is in conflict with the first write. Thus the second write is not executable concurrently with the first one. In this model during executing an XPath query, all the nodes existing in the path which is traversed, are locked in read mode. This applies a great restriction to concurrency and prevents many update operations which are logically can be done concurrently by that read operation. For example consider XML document in Figure 1 and this XPath query: /Library/Books/Book/Title. Applying this query to the document Figure1 locks Library, Books, Book, Title elements in read mode. Now, if we want to insert a new Chapter node under <Book id=1> it is not possible because we need write lock on it but it has a read lock, and conflict happens. Finally delete operation is only applicable for leaf nodes, so it's not possible to delete a subtree by a single delete command. In order to delete a subtree, every node in it needs a separate command to be deleted one by one in bottom-up order.

Locking protocol introduced in[10] places locks on different granules according to their types. So it involves lower locking overhead by requesting fewer locks for concurrency. Placing intention locks on every node in the path from the root node to those subtrees or nodes that a transaction wants to read or write, besides preventing integrity defects, provides an appropriate level of concurrency. But this approach causes some restrictions by preventing some certain operations from performing concurrently, although logically they are executable concurrently. For

```
<Library>
        <Books>
                <Book  id="1">
                        <Title>Database</Title>
                        <Chapter num="1">
                                <Subject>storage</Subject>
                                <Content>A relational database ….</Content>
                        </Chapter >
                        <Chapter num="2">
                                <Subject>indexing</Subject>
                                <Content>A relational database ….</Content>
                        </Chapter>
                </Book>
                <Book id="2">
                        <Title>Native XML Databases</Title>
                        <Chapter num="1">
                                <Subject>storage</Subject>
                                <Content>A native XML database ….</Content>
                        </Chapter >
                </Book>
        </Books>
        <Magazines>
                <Magazine id = "1">
                        <Title>computer science</Title>
                </Magazine>
        </Magazines>
</library>
```

Figure 1. A fragment of an XML document library.

example in document Figure 1 assume that transaction T1 wants to read every books subject while transaction T2 wants to add a new chapter to the book entitled "Native XML Database". To read all book's subject, T1 should perform path expression /Books/Book/Subject on the Library document. To perform this query T1 requires IS-locks on Library, Books and Book nodes and it requires S-lock on Subject nodes. To add a new chapter, T2 requires IX-locks on Library and Books nodes and it requires X-lock on Book node by Id=1. This node was locked in IS mode by T1 and since IS-lock conflicts with X-lock; T2 cannot acquire its needed locks and should wait. But logically it is possible to add chapters while reading subject of a book and performing these two operations concurrently should not lead to any defect. Also this method does not define any update operation for text nodes.

There are other concurrency control protocols which are listed in[12] that in all of them we did not find the idea of considering the child node in locking.

In this paper we introduce our novel approach named XPLC: "XML Path Locking by Child Consideration". To achieve higher level of concurrency, we introduce the notion of "Child Consideration" in our path locking approach. This means when a node is locked due an operation, the lock type is chosen with respect to not only the operation type but also the next child of the node in the path. This means some of our locks have two parts. The first part is its ordinary lock type chosen from lock table regarding the operation and the second part is chosen with respect to the next child in the path.

In order to use proper number of locks, we have assigned proper granularity to our locks. For example the granularity of our read and delete locks is set at the subtree level while our intention locks are set at the node level. The data model in our approach is based on a simplification of the standard XPath data model which is described in section 3.1.

The remaining parts of this paper are organized as the following: We introduce our approach in Section 2, while we have a serializability analysis and comparison of our work with respect to others in section 3. We have our conclusion and future works in section 4.

2. XPLC: Our Proposed Protocol

XML Path Locking by Child Consideration (XPLC) is our novel protocol for concurrency. In order to clarify our approach, we first present our data model.

2.1 Data Model

The data model used in our approach is based on XPath data model[5,8,9]. In this model there is no difference between element, attribute or text nodes. We classify nodes only to terminal and non-terminal cases. Terminal nodes are the leaf nodes which cannot accept any child. They contain an uninterrupted stream of bytes, such as text strings, graphics, or video/audio sequences.

Definition: An Xtree xt is a tuple (N, B, r, V) where N is a set of nodes, B:N×N is a binary relation representing the directed edges (branches) of the tree xt, and r∈ N is the root node of xt. The function V maps the nodes (except r) to strings representing the node's name.

The Library document in Figure 1 has been represented as an Xtree in Figure 2. The value of each node in Figure 2 is shown in Table 1.

2.2 Operations

Like all traditional databases, XML databases have four operations: Read, Insert, Delete, and Update. In this paper we use the following notations for these operations:

- r(p): This query operation retrieves XML data based on the path p. Path expressions are based on XPath query language syntax.
- a(n, v): This operation creates a new node m with V(m) =v and a new edge (n, m) in the Xtree.
- d(n): This operation deletes node n. All the nodes, whose ancestor is node n, are also deleted. In other words this operation deletes the subtree whose root is node n.
- u(n,v): This operation changes the value of the node n to a new value v. This operation is only done on the terminal nodes.

2.3 Lock Modes and their Granularity

Regarding to the previous defined operations on Xtree and the notion of the child consideration, we propose the lock modes needed to support concurrency between these operations as follow:

- R lock: This lock is set on the nodes which their corresponding subtree is the result of a r(p) query. As a result the granularity of R lock is at subtree level.
- IR_C lock: If a node like n is locked by IR_C lock, it means that a transaction is going to read some descendants of n. The index *c* means that the next child of node

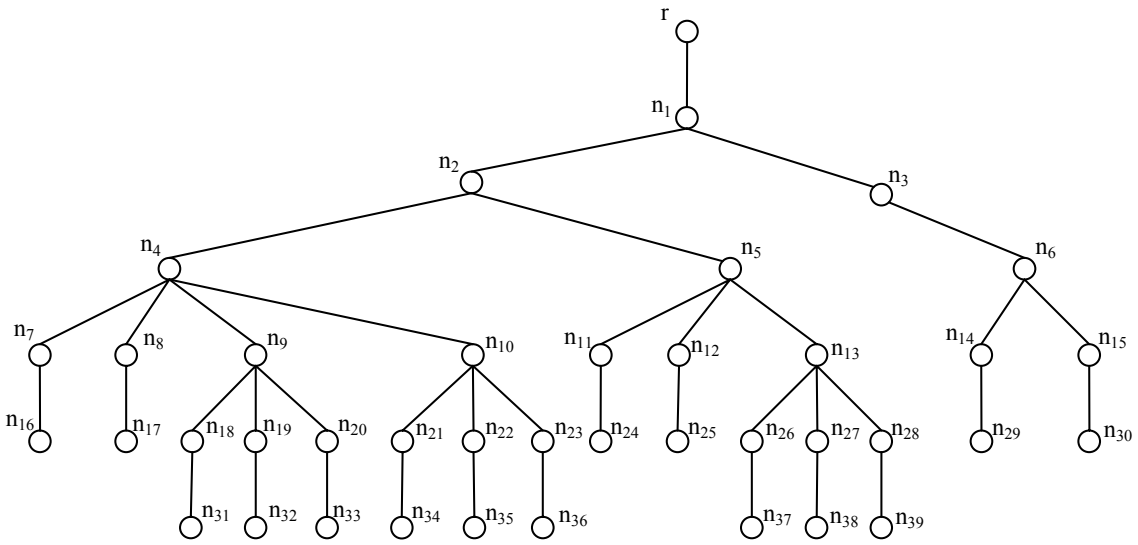

Figure 2. A representation of XML document.

Table 1. Values of the nodes shown in Figure 1

$V(n_1)$ = Library	$V(n_{14})$ = id	$V(n_{27})$ = Subject
$V(n_2)$ = Books	$V(n_{15})$ = Title	$V(n_{28})$ = Content
$V(n_3)$ = Magazines	$V(n_{16})$ = "1"	$V(n_{29})$ = "1"
$V(n_4)$ = Book	$V(n_{17})$ = "Database"	$V(n_{30})$ = computer science
$V(n_5)$ = Book	$V(n_{18})$ = num	$V(n_{31})$ = "1"
$V(n_6)$ = Magazine	$V(n_{19})$ = Subject	$V(n_{32})$ = storage
$V(n_7)$ = id	$V(n_{20})$ = Content	$V(n_{33})$ = A relational database ….
$V(n_8)$ = Title	$V(n_{21})$ = num	$V(n_{34})$ = "2"
$V(n_9)$ = Chapter	$V(n_{22})$ = Subject	$V(n_{35})$ = indexing
$V(n_{10})$ = Chapter	$V(n_{23})$ = Content	$V(n_{36})$ = relational database ….
$V(n_{11})$ = id	$V(n_{24})$ = "2"	$V(n_{37})$ = "1"
$V(n_{12})$ = Title	$V(n_{25})$ = Native XML Databases	$V(n_{38})$ =storage
$V(n_{13})$ = Chapter	$V(n_{26})$ = num	$V(n_{39})$ = A native XML database ….

n with type *c* is the next node that should be locked. This lock is needed to prevent conflicting operations, e.g. delete, from operating on this node, while other reading transactions are active in its subtree. This lock considers the child of the node which it is applied to. This lock is only applicable on non-terminal nodes.

- A_C lock: If a node n is locked by this lock, it means that an append operation a(n, v) is going to add a child node m under the node n. As a result this lock is only applicable to non-terminal nodes and its granularity is at node level. The index *c* in this lock is set equal to v which shows this lock is set to add a node with v type. So it is possible to place an append lock A_c on a node locked in IR_c mode they have different indexes.
- U lock: When an update operation u(n, v) wants to change the content of the node n to v it locks n by U lock. Thus prevents reading or writing this node while updating operation is active. This lock is only applicable to terminal nodes so its granularity is at node level.
- IC lock: This lock is applied to the nodes in the path from root to destination nodes of every update operations e.g. (delete, insert and update). This lock is only applicable to non-terminal nodes with granularity at node level.
- D lock: If a node n is locked by this lock, it means the node n and its corresponding subtree will be deleted. As a result granularity of this lock is at subtree level.

A summary of the lock modes and their compatibility rules is given in Table 2.

In order to show the correctness of our locking behavior, the following explanations are needed to clarify the intuitions behind our locks.

- It is logically possible to have two insert operations under a node by two different transactions[5,9]. So two A_C locks are not in conflict with each other.

Table 2. Compatibility matrix

	IR_C	IC	R	A_C	U	D
IR_C	+	+	+	→	x	–
IC	+	+	–	+	x	–
R	+	–	+	–	–	–
A_C	→	+	–	+	x	–
U	x	x	–	x	–	–
D	–	–	–	–	–	+

+: compatible –: conflict →: conditional
x: impossible
(→) means that those locks are compatible if the C parts of them are different.

- It is not possible to have A_C lock and U lock on the same node because A_C lock is only applicable on non-terminal nodes while U lock is used only for terminal nodes.
- Suppose that two transactions want to lock a node n by A_{C1} and IR_{C2} locks. These two locks are in conflict if C1=C2, because this means that the first transaction wants to add a C1 type node while the second transaction wants to read C1 type nodes and this lead to phantom problem.
- The IR_C, IC and A_C locks are only applicable to non-terminal nodes while U lock is only used for terminal nodes. In other words it is not possible to request U-lock for the node which is locked with IR_C, IC or A_C locks.

2.4 Locking Protocol

In order to demonstrate our locking protocol, first we present the scenario of locking when an operation is scheduled.

- r(p): To perform this read operation all the nodes in the path p, except the destination nodes which are locked by R-lock, are locked by IR_C locks. For example r(/Library/Books/Book) leads to the following locks:
  IR_{C1}(r), IR_{C2} (n1), IR_{C3}(n2), R(n4), R(n5).
  where, C1="Library", C2="Books", and C3="Book". It is clear that before reaching the destinations node n4 and n5, we lock r, n1, and n2 with respect to the child of them.
- a(n, v): To perform this insert operation, all the nodes from the root r till node n are locked by IC lock. Node n is also locked by A_C lock. For example to add a chapter element behind the <Book id="1"> the following lock should be set:
  IC(r), IC(n1), IC(n2), A_C(n4),
  where, C="Chapter".
- d(n): This operation deletes the node n with its corresponding subtree. To perform this delete operation, all the nodes from the root r till node n is locked by IC and node n is locked by D-lock. For example, deleting <book id="2"> needs the following locks:
  IC(r), IC(n1), IC(n2), D(n5).
- u(n, v): This operation updates the content of a terminal node. To perform this operation the existing nodes from root till node n is locked by IC locks and the node n is locked by U lock. For example updating the title of the book with id="1" to "Databases Concepts" needs the following locks:
  IC(r), IC(n1), IC(n2), IC(n4), IC(n8), U(n17)

Now we are ready to present our locking protocol. XPLC protocol is based on the following rules:

Rule 1: Before performing any operations, use the proper locking scenario.

Rule 2: Before a transaction acquires a lock its compatibility should be checked with lock compatibility matrix (Table 2).

Rule 3: Each transaction should obey 2PL protocol.

3. Analysis and Comparison

3.1 Serializability Analysis

The combination of locking with correct behavior and observing 2PL protocol ensures serializability of the locking protocol[11]. According to this rule our XPLC protocol is serializable because:

- In section 3.3 we have shown the correctness of our locking behavior.
- Rule 3 of XPLC protocol explicitly express the existences of 2PL in this approach.

3.2 Comparison with the Previous Protocols

We have found seven problems in the previous protocols. These problems have been listed below:

Problem 1: XLP protocol assumes R-lock and I-lock are compatible because the insert operation modifies the structure rather than the content of a node but the read operation locks the content of the node. This assumption may not be true and may lead to phantom problem when a transaction inserting a child under a node is read by another transaction.

Problem 2: In XLP Protocol Read lock (R-lock), write lock (W-lock), insertion lock (I-lock) and deletion lock (D-lock) are applied to destination nodes and the pass-by nodes in path from root to destination nodes are temporarily locked in P mode. This may lead to integrity defect when a transaction wants to delete a subtree while another one performs a reading operation in lower levels in the subtree.

Problem 3: Some of these models like Dekyser and Hidder[5,8,9] and Lightweight multigranularity locking[10] do not define any independent update operation. It implies that they need to simulate update operation by a delete and insert command.

Problem 4: Sedna[7] has ambiguity in its concurrency control method, since it uses a labeling algorithm but does not define its mechanism for assigning labels to the nodes of XML document.

Problem 5: XPath-based concurrency control[6] uses three copies of a document to control concurrency. This is an expensive method and has high overhead.

Problem 6: Having proper granularity levels for locks according to their types lead to reduce the number of the needed locks. However most of the previous protocols except[10] do not consider this fact.

Problem 7: All of the proposed protocols prevent concurrent executing of some operations which can logically commute. For example in document Figure 1 assume that transaction T1 wants to add a new chapter to the book with id=2 and transaction T2 wants to read Title of this book, so T1 should acquire a lock on Book node with id=2 for its writing operation and T2 should acquire a lock for its reading operation on the same node. In all of the previous protocols, these two operations cannot be performed concurrently because one of them needs a read lock on a node while the other needs a write lock on it. These two locks are in conflict according to the rules expressed by the previous protocol. But performing these two operations logically is possible and does not result in any defect since these two operations work on different types of the Book node's children and do not interfere in the execution process of each other.

In our approach regards to our locks and the locking protocol none of the above problems could occur. As a result we could claim that our protocol have higher concurrency control level without any defect.

4. Conclusion and Future Works

XML has become the most important technique to exchange and store data in the Web. Managing huge data stored in XML documents, emphasizes the need for native XML databases. While XML is hierarchal and represented by tree models, its flexible structure makes the previous tree locking protocols inadequate. Providing a high degree of concurrency in XML databases is crucial in many applications. In this paper we have proposed XPLC protocol for XML concurrency control. It is a novel approach which introduces the notion of child consideration in XPath locking protocols. This protocol unlike the pervious ones allows two transactions to have concurrent update operations e.g. (insert, delete, update) below the same nodes. Using A_C and IR_C lock modes allows a transaction to read a subset of a subtree and at the same time it allows another transaction to insert a node of different type under the root of that subtree. These enhancements provide higher level of concurrency to XPLC protocol.

As our locking protocol is a logical protocol, our future work is adapting our logical locking protocol to a proper physical locking protocol to have a tradeoff between concurrency and disk access.

5. References

1. Bourret R. Going native: use cases for native XML databases. [Internet]. Available from: http://www.rpbourret.com/xml/UseCases.htm
2. Feinberg G. Anatomy of a native XML database, XML 2004; 2004.
3. Jagadish HV, Al-Khalifa S, Chapman A, Lakshmanan LVS, Nierman A, Paparizos S, et al. TIMBER: a native XML database. VLDB Journal. 2002; 11:274–91.
4. Jea KF, Chen SY, Wang SH. Concurrency control in XML document databases: XPath locking protocol. Proceedings of the Ninth International Conference on Parallel and Distributed Systems (ICPADS'02); 2002 Dec 17–20; 2002; p. 551–556.
5. Dekeyser SS, Hidders J. Conflict scheduling of transactions on XML documents. Fifteenth Australasian Database Conference (ADC 2004). 2004; Dunedin, New Zealand: Australian Computer Society Inc. 2004. p. 93–101.
6. Hye Choi E, Kanai T. XPath-based Concurrency Control for XML Data. Proceedings of the 14th Data Engineering Workshop (DEWS 2003) Japan; 2003. pp. 302–313.

7. Pleshachkov P, Novak L. Transaction isolation in the sedna native XML DBMS. Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYR CoDIS. St.-Petersburg, Russia; 2004.
8. Dekeyser S, Hidders J. Path locks for XML document collaboration. Proceedings of the Third International Conference on Web Information Systems Engineering; 2002. Dec 12–14; Singapore, 2002. p.105–14.
9. Dekeyser S, Hidders J. A transaction model for XML databases. World Wide Web: Internet and Web Information Systems. 2004 Mar; 7(1): 29–57.
10. Choi Y, Moon S. Lightweight multigranularity locking for transaction management in XML database systems. J Syst Software. 2005; 78:37–46.
11. Weikum G, Vossen G. Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann; 2002.
12. Haustein M, Härder T, Luttenberger K. Contest of XML lock protocols. Proceedings of the 32nd International Conference on Very Large Databases (VLDB 2006), 2006 Sep 9; VLDB Endowment Inc.; 2006; p.1069–80.