International Journal of Database Theory and Application
Vol.8, No.2 (2015), pp. 215-222
http://dx.doi.org/10.14257/ijdta.2015.8.2.20

FOCCX: An Optimistic Concurrency Control Protocol over XML

Weifeng Shan and Husheng Liao

College of Computer Science, Beijing University of Technology, Beijing 100124, China
Department of Disaster Information Engineering, Institute of Disaster Prevention, Sanhe 065201, China
shwf@163.com, liaohs@bjut.edu.cn

Abstract

XML concurrency control protocol (CCP) is used to guard the consistence and isolation of transactions in Native XML databases. Experiments show that locking overhead of existing approaches based on locking may be huge, especially in the applications with few or without conflicts. Optimistic concurrency control (OCC) is an alternative to locking. This paper presents a new optimistic approach for concurrency control over XML documents named FOCCX (Forward oriented Optimistic Concurrency Control over XML) facing XPath-based API. FOCCX increases the degree of transaction concurrency. This is achieved by aborting the current transaction when a potential UPDATE-UPDATE conflict taking place as early as possible, and reduces comparison times by checking a small write set against read set of a limited number of concurrent transactions. Experimental results show that our protocol has superior performance to approaches based on Backward Oriented mechanism (BOCC).

Keywords: *XML Databases, Optimistic Concurrency Control, Transaction, XPath*

1. Introduction

As a general markup language, XML has self-description, cross-platform features, and has been widely used in data exchange and data representation fields. An XML document is usually represented as a label tree, and accessed by XPath, XQuery languages. If an XML document is concurrently accessed by many users in web applications or Native XML databases, some unpredictable problems such as lost update, dirty read, or phantoms may occur.

Concurrency control mechanism is a key component of database systems, and provides data consistency in multi-user environment. Although these protocols have been researched and used in many relational database systems such as Oracle, SQL Server and MySQL, etc.,, they are not ideal approaches for XML because of its hierarchical structure character. In the past decade, many concurrency control protocols have been proposed to deal with XML. Most of them are locking-based where a transaction can proceed if the lock on the target node is compatible with locks held by other transactions on the same node[1-3].

Due to the pessimistic nature of locking, experiments of [4, 5] have shown that locking overhead may be huge, especially for applications with few conflicts. Furthermore, locking protocols are not deadlock-free. Optimistic concurrency control[6] scheme is an alternative to locking when the conflict rate is low, and can get rid of the locking overhead. [7] proposes two optimistic concurrency control mechanisms based on snapshot technology over XML document: OptiX and SnaX. In OptiX, all nodes read and written by transaction T_i are recorded in read phase, denoted by $RS(T_i)$ and $WS(T_i)$ respectively. A transaction T_i passes validation if for each concurrent transaction T_j that already validated, $WS(T_j) \cap RS(T_i)=\emptyset$. SnaX provides the isolation level snapshot isolation. Different from OptiX, SnaX does not keep track of reads. Instead, it only deals with UPDATE-UPDATE conflicts. That is, a transaction T_i passes validation if for each concurrent transaction T_j that already validated, $WS(T_j) \cap WS(T_i)=\emptyset$. An important point is that no two transactions can be concurrently in validation phase for both OptiX and SnaX. In order to improve the validation phase duration, [8] presents a novel optimistic path-based approach where most conflicts can be detected by analyzing XPath expressions instead of XML nodes. However, it has to check the conflict at node level when XPath contains predicates or wildcards such as '//' and '*'. [9] discusses another for valid XML, where READ-UPDATE and UPDATE-UPDATE conflicts are effectively detected when the operations are specified using XPath expressions according to the scheme information of XML such as DTD.

Experimental results show that OptiX, SnaX, [8] and [9] protocols have better concurrency than those based on locking under low contention. However, they also have several disadvantages. First, OptiX has to validate a potentially large read set against a large number of old write sets since it uses the backward oriented validation strategy(BOCC). Secondly, although SnaX needs not to keep track of read set of transactions, it does not guarantee the serializability, and it could lead to a lost update problem. Finally, Both [8] and [9] use XPath and scheme of XML to reduce the duration of validation, but they only support a small set of XPath expression.

Forward oriented optimistic control protocols(FOCC) [10] is another approach that it checks during the validation phase of T_i its write set $WS(T_i)$ intersects with any of the read sets $RS(T_j)$ of all transactions T_j having not yet finished their read phases. It is clear that the write set is often a small subset of the read set in query-dominated transactions. So FOCC usually has less comparison times than BOCC and offers multi choices in handling and optimizing conflict resolution.

In this paper, we consider a new optimistic concurrency control protocol over XML, named FOCCX, based on FOCC. It has all advantages of FOCC. In addition, when a potential UPDATE-UPDATE conflict is checked, it aborts the transaction as early as possible to avoid some unnecessary "wasted work". We discuss in detail the design issues and conflict detection algorithm. Experimental results show that it has better performance than OptiX and SnaX.

The rest of the paper is organized as follows. Section 2 presents the XML memory data model, operations and transaction model of FOCCX. Section 3 discusses the implementation issues, especially the conflict detection algorithm of FOCCX. Section 4 describes the experimental results. Finally, Section 5 concludes the paper.

2. FOCCX Protocol

In this section, we introduce the XML memory data model, XPath operations and transaction model used in FOCCX.

2.1. XML Memory Data Model

Generally, an XML document is represented as a labeled tree in memory, and everything in an XML document is viewed as a node, including document node, element node, attribute node and text node, etc…A valid XML document is a well-formed document that confirms to the stricter rules specified in a DTD or an XML Schema.

In this paper, we use *MemXMLTree* as the memory data model in our protocol which is a simplification of the standard XPath data model.

An *MemXMLTree t* is a tuple*(N,E,r)* where *N* is set of nodes, $E \subseteq N \times N$ is a binary relation representing the directed edges of the tree *t*, and $r \in N$ is the root node of *t*.

2.2. XPath Operations

Since most of XML languages are based on XPath expressions such as XQuery, we use XPath API to access XML in our protocol. An operation over XML document may be a query, modification, insertion or deletion. All these operations execute over the *MemXMLTree*, and travel the *MemXMLTree* from top to bottom to locate the target node(s) to read and(or) write. We distinguish them two kinds of operations:

a) Read operation: The operation doesn't change the content of nodes and the structure of *MemXMLTree*.

query(p): The operation returns all nodes located by the XPath expression *p*.

b) Write operations: A write operation changes the nodes' content or the structure of *MemXMLTree*. The possible write operations are deletion, insertion, updating, replace, etc.,. In order not to overburden the discussion, we only consider to delete and insert operations in this paper. Other complex write operations, such as replace or update, can be defined by combining the preceding operations.

delete(p): The operation removes the nodes(s) and its(their) sub tree located by the XPath expression *p*.

insert (p, n ,q): The operation inserts the XML fragment *q* as the *n*th child node of *p* located by the XPath expression *p*.

Where, *p* is a XPath expression, and it is used by query engine to locate the target nodes that satisfy the expression *p*. It supports axis such as child, descendant, descendant-or-self, parent, preceding, preceding-sibling, following, following-sibling and ancestor, ancestor-or-self and self. Now, only position predicates are allowed in *p* in our implementation. Obviously, it is easy to extend it to support other predicts such as value predicate. If predicates contain path constraint, all the nodes satisfied it are also added to NS read set as follows. In this paper, we omit it.

2.3. Transaction Model of FOCCX

Like those optimistic concurrency control protocols, our concurrency control mechanism also has three phases: a working phase, a validation phase and a write phase.

Working phase: When transaction T_i starts, it receives a unique identifier *TS*(T_i). Transaction T_i only access the version that was most recently committed version as of the time T_i started, i.e., it should not see the new added nodes made by a concurrent transaction.

Validation phase: Once a transaction T_i has finished its working phase and wants to commit, it goes into validation phase. Only one transaction can perform validation at a time in order to ensure the serialization order. In FOCCX, validation checks, whether the write set *WS*(T_i) of T_i intersects with any of the read sets *RS*(T_j) of all transactions T_j having not yet finished their working phases. Once a conflict occurs, the current transaction T_i will be aborted.

Write phase: If the validation phase is successful, the modification carried out by the transaction become visible to other transactions, otherwise the transaction is aborted, its temporary space freed.

3. Implementation of FOCCX

There are two main challenges when adjusting optimistic concurrency control protocols to XML documents for its hierarchical structure. Firstly, we have to identify the read and write sets over XML. Secondly, we must decide when conflicts occur. The ancestor/descendant relationship between nodes makes difficult to detect conflict among transactions.

Firstly, We defined READ-UPDATE and UPDATE-UPDATE conflicts. Then the read set and write of transaction are discussed. Finally, we talk about the conflict detecting algorithm.

3.1. What is a Conflict?

In traditional optimistic concurrency control protocols, two transactions T_i and T_j are not conflict if they are no read dependency, that is T_i does not read data modified by a concurrent transaction T_j and vice versa, and no overwriting, i.e. T_i does not overwrite data, which has been written by a concurrent transaction T_j and vice versa[10].

Read dependency means a READ-UPDATE conflict, overwriting means a UPDATE-UPDATE conflict.. An XML document is modeled as a labeled *MemXMLTree t*, where each node has a label from an infinite alphabet Σ. The set of all trees over Σ will be denoted as T_Σ. We use $R_i(t)$ and $U_j(t)$ to indicate that transactions T_i read the XML document t and T_j update t such as delete or insert a node[9].

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

It means that if the scope of a READ operation includes that of an UPDATE operation, then the two operations are a READ-UPDATE conflict.

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

If the scope of UPDATE operation U_i includes all parts or some parts of that of U_j, or vice versa, the two operations are UPDATE-UPDATE conflict.

3.2. Read Set and Write Set

In an XML document, when a transaction is reading the node, the other concurrent transaction may be deleting this node's ancestor node, this is not allowed. So we must define Read Set and Write Set of a transaction. In order to provide quick conflict detection, we differentiate different subsets within the read set $RS(T_i)$ of a transaction T_i. $RS(T_i)= RR(T_i)\ NS(T_i)$.

In order to explain how to maintain the read set and write set of a transaction, we use the following transactions. For the sake of simplicity, we only consider transactions composed of a single operation.

T1: query(/site/regions/asia/item[x]);
T2: query(/site/regions/asia/item[x]/payment);
T3: query(/site/regions/asia/item[x]/mailbox);
T4: delete(/site/regions/asia/item[x]);
T5: delete(/site/regions/asia/item[x]/incategory);
T6: delete(/site/regions/asia/item[x]//mail[y]);
T7: insert(/site/regions/asia/item[x],0, <incategory category="computer" />);
T8: insert(/site/regions/asia/item[x]/mailbox,0,
<mail><from>beijing</from><to>shanghai</to><date>01/02/2014</date><text>book</text></mail>);
T9: insert(/site/regions/asia/,x, <item>…..</ item >);

***RR(T_i)*:** The read return nodes of a transaction T_i . These nodes are the roots of the subtrees returned as part of query operation. In T1, item[x] will be added to RS(T1).

NS(T_i): This set contains all ancestors of target nodes of query, delete and insert operation in a transaction T_i. In T2, site, regions, asia and item[x] belongs to NS(T2). Consider the transaction T6, site, regions, asia, item[x], mailbox will be added to NS(T6).

The write set of a transaction contains nodes that are modified in the transaction, like delete or insert operations. $WS(T_i) = D(T_i) \cup I(T_i)$.

D(T$_i$): This set contains all nodes that were deleted by the transaction T_i. In T5, all incategory nodes of item[x] are considered part of D(T5). Similar to $R(T_i)$, although deletion changes entire subtrees, only the roots are added in D(T) in order to keep the set smaller.

I(T_i): This set contains all the immediate parents of any nodes inserted in the XML tree by the transaction T_i. Let's take T7 as an example, item[x] will be added into I(T7). In T8, mailbox node of item [x] is considered part of I(T8).

3.3. Snapshot Technology

Like OptiX and SnaX, we use the same snapshot technology proposed in [7] to implement a multi-version system. Each XML memory node *n* in *MemXMLTree* has a valid timestamp *V* to identify the transaction that created this node. *n* also has an invalid timestamp *IV* that is the identifier of the transaction that deleted this node. If no transaction has deleted *n* so far, then it's *IV=NULL*.

In order to add the predecessor of target nodes to the $NS(T_i)$, we keep the reference of the parent node in each node in *MemXMLTree*.

When a *query* operation coming, it will be executed over the *MemXMLTree* directly. All return nodes are added to *RR* set and their ancestors insert into *NS* set. If the operation is *insert* operation, the new subtree will be inserted into the *MemXMLTree* immediately. However, it is invisible to other concurrent transactions.

If it is a *delete* operation, it will check whether another transaction has modified it before. If modified, it will cause the current transaction rollback. This is different from traditional OCC, because in the implementation of FOCCX, we use one IV flag in MemXMLNode to indicate which transaction has modified it. If this flag is not null, there may have a possible conflict with other concurrent transactions (UPDATE-UPDATE conflict). There are two benefits of this approach, one is it avoids unrecoverable schedule while two conflicting concurrent transactions fail, the other is it increases the throughput of transactions since it detects potential conflicts as early as possible. Obviously, it may cause unnecessary abort and make the abort rate increased.

3.4. Conflict Detection

After had defined the conflict over XML tree, read and write set of a transaction, we now have to adjust the traditional conflict detection algorithm to work with the XML tree model. Similar to concurrency control protocols based on locking, we use conflict matrix to detect the potential conflicts between two transactions, as shown in Table 1. If there is a ✓ in the matrix, operations are compatible. If there is a ×, the two operations conflict, and lead to an abort of T_i.

Assume T_i and T_j are two simultaneous transactions, node *p* is the predecessor of node *q*. When T_i enter into validation phase, we have to check write set of current transaction T_i with read sets of other concurrent transactions to find where there is a conflict.

We first have a look at a node *p* (resp. *q*) that is both in $RS(T_j)$ and $WS(T_j)$.

Table 1. Conflict Matrix for FOCCX

T_i		T_j			
		p		*q*	
		NS	*RR*	*NS*	*RR*
p	*I*	✓	×	✓	✓
	D	×	×	×	×
q	*I*	✓	×	✓	×

D	✓	×	×	×

- $p \in I(T_i) \cap p \in NS(T_j)$: There is no conflict. This is because T_j only reads the node *p* and not its descendants and so the insert of a new subtree in *p* by T_j does not cause a problem.
- $p \in I(T_i) \cap p \in RR(T_j)$: There is a conflict. Because T_j returns the whole subtree with root node *p*, so the added new subtree in *p* by T_i will be visible to T_j.
- $p \in D(T_i) \cap p \in \{NS(T_j), RR(T_j)\}$: There is a conflict. Once T_i deletes the node *p*, T_j cannot read and travel the node *p*, so they conflict.

Now, we turn to considering the condition when two transactions operating on two different nodes with ancestor/descendant relationships.

- $p \in I(T_i) \cap q \in \{NS(T_j), RR(T_j)\}$: No conflict occurs. Although T_i inserts a new subtree in *p*, it does not affect T_j for it only reads the descendant node *q* of *p*.
- $p \in D(T_i) \cap q \in \{NS(T_j), RR(T_j)\}$: A conflict occurs. Because T_i deletes the node *p*, T_j cannot read or navigate the descendant node *q* through *p*.
- $q \in I(T_i) \cap p \in NS(T_j)$: No conflict occurs. Although T_i inserts a new subtree in *p*, it does not affect T_j for it only reads the descendant node through *p*.
- $q \in I(T_i) \cap p \in RR(T_j)$: A conflict occurs. T_j reads the new inserted child node *q* of *p*, which is added by T_i.
- $q \in D(T_i) \cap p \in NS(T_j)$: No conflict occurs. Although *q* is deleted by T_i, it does not affect T_j to travel other nodes by *q*'s ancestor node *p*.
- $q \in D(T_i) \cap p \in RR(T_j)$: A conflict occurs. T_j misses the node *q* that deleted by T_i.

4. Experimental Evaluation

We implemented a simple memory database in Java to test the performance of different XML concurrency control protocols. For our experiments, we used a PC with Intel Core i5-2520 CPU (two cores, 2.5G Hz) and 12G RAM running Windows 7 Ultimate x64.

We use a standard XMark [11] tool to generate a well-formed, valid XML document with about 100Mb. However, it is too big for our experiment machine, so we select part of the document with about two thousand 'item' nodes under path '/site/regions/Asia/', about 5Mb, to evaluate the performance of different protocols. According to the benchmark DTD definition, the height of the XML document is 7. Evaluation transactions are T1 to T9 listed above.

As optimistic concurrency control protocols are ideal for environment with few or without conflicts, we set query transaction occupy 90%, insert and delete transaction is 5% respectively.

Figure 1 and Figure 2 show the performance and abort rate evaluation results for varying the number of concurrently running transactions. Clearly, an increase in the number of concurrent transactions leads to high throughput and more conflicts. When there is only one running transaction, all protocols have the same throughput since no conflict occurs. In all cases, FOCCX has better performance than OptiX and SnaX protocols. Because FOCCX aborts the transaction in working phase when a potential conflict is checked, it has a higher abort rate than OptiX and SnaX in most cases. SnaX has a lower abort rate for it only checks the UPDATE-UPDATE conflicts in validation phase. However, it cannot ensure the serializability of transactions.

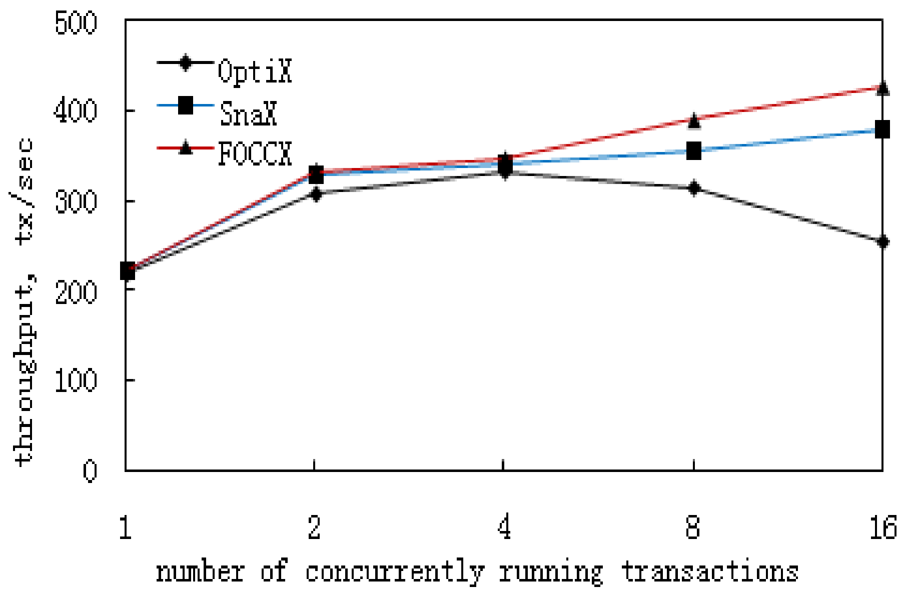

Figure 1. Throughput of Protocols

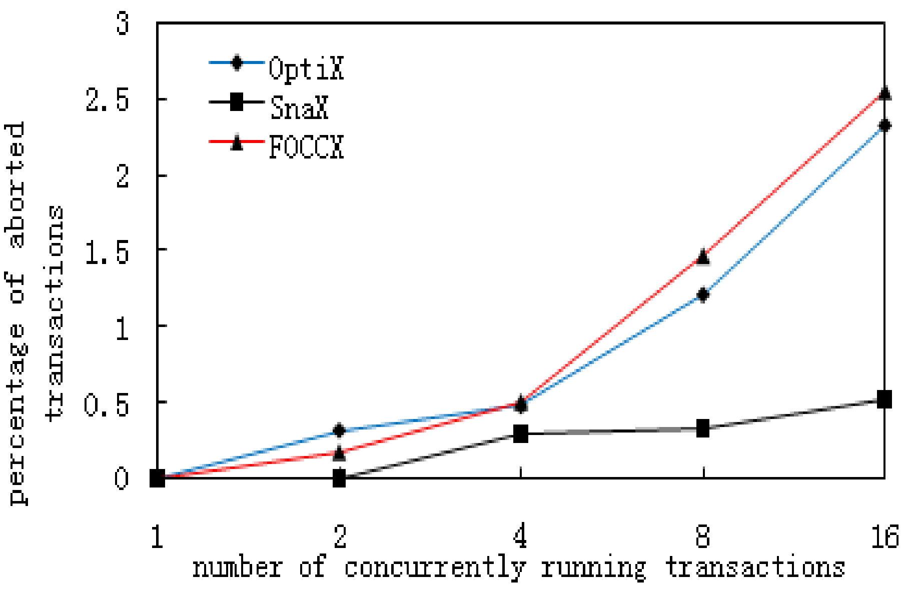

Figure 2. Abort Rate of Protocols

5. Conclusion

In order to isolate read and modifications over XML document and guarantee serializability, we introduced a new optimistic concurrency control protocol—FOCCX. Unlike others adopt backward validation strategy, FOCCX uses forward validation solution. As FOCCX allows read-only transaction committed directly without validation, it has better performance in experiments.

Since the transactions to be checked during validation have not yet committed, FOCCX offers more flexibility in handing a detected conflict than OptiX and SnaX. For example, the current transaction can be deferred instead of aborting or abort the conflict transaction.

Like OptiX and SnaX, we use a simple snapshot technology in FOCCX. However, it does not allow two concurrent transactions to modify the same node, so we abort the later transaction in the current version of FOCCX. We plan to implement a real multi-version FOCCX based on snapshot isolation technology to improve its performance.

Acknowledgements

This work was supported by the National Science Foundation for Young Scholars of China (61202074) and Beijing Natural Science Foundation (4122011).

References

[1] S. Helmer, C. C. Kanne and G. Moerkotte, "Evaluating lock-based protocols for cooperation on XML documents", SIGMOD Record, vol. 33, no. 1, **(2004)**.
[2] K. F. Jea and S. Y. Chen, "A high concurrency XPath-based locking protocol for XML databases", Information and Software Technology, vol. 48, no. 8, **(2006)**.
[3] S. Helmer, C. C. Kanne, G. Moerkotte, *et al.*, "Lock-based protocols for cooperation on XML documents", 14th International Workshop on Database and Expert Systems Applications, **(2003)** September 1-5, Prague, Czech Republic.
[4] M. P. Haustein and T. Harde, "Adjustable transaction isolation in XML database management systems", Second International XML Database Symposium(XSym 2004), **(2004)** Auguest 29-30, Berlin, Germany.
[5] M. Haustein, T. Harderand and K. Luttenberger, "Contest of XML lock protocols. Proceedings of the 32nd international conference on Very large data bases, **(2006)** September 12-15, Seoul, Korea.
[6] H. T. Kung and J. T. Ronbinson, "On Optimistic Methods for Concurrency Control. ACM Transactions on Database Systems," vol. 6, no. 2, **(1981)**.
[7] Z. Sardar and B. Kemme, "Don't be a pessimist: Use snapshot based concurrency control for XML", Proceedings of International Conference on Data Engineering, **(2006)** April 3–7, Atlanta, GA, USA.
[8] D. Berrabah, S. Gancarski, S. K. Chikh and C. L. Pape, "Optimistic path-based concurrency control over XML documents," 5th International Conference on Soft Computing as Transdisciplinary Science and Technology, **(2008)** October 27-31, Cergy-Pontoise, France.
[9] C. Byun, I. Yun and S. Park, "A New Optimistic Concurrency Control in Valid XML", Journal of Information Science and Engineering, vol. 25, no. 1, **(2009)**.
[10] T. Hardert, "Observations on Optimistic Concurrency Control Schemes. Information Systems", vol. 9, vol. 2 **(1984)**.
[11] A. Schmidt, "X Mark-An XML Benchmark Project, http://www. xml-benchmark. org . **(2009)**.

Authors

Weifeng Shan, He is a PhD student at the Beijing University of Technology with research interests in XML and parallel computing. He has received his master's degree in Software Engineering from YunNan University, China.

Husheng Liao, He received his M.S degree from Tsinghai University in 1981. Now, he is a professor of Beijing University of Technology. His main research interests include compiler, program languages and XML.