Don't be a Pessimist: Use Snapshot based Concurrency Control for XML

Zeeshan Sardar　　　　Bettina Kemme
McGill University, Montreal, Canada
{zsarda,kemme}@cs.mcgill.ca

1 Introduction

As native XML database systems (e.g., [3, 7, 8]) get increasingly popular, fine-granularity concurrency control becomes imperative in order to allow different clients to concurrently access the same documents. Existing concurrency control approaches for XML are mainly based on locking [2, 3, 4, 6, 5]. However, the experiments of [5] have shown that the locking overhead, especially for read operations, can be tremendous. In this paper, we present two snapshot based concurrency control mechanisms that avoid locking. Instead, transactions access a committed snapshot of the data. OptiX is a variation of optimistic concurrency control adjusted to use snapshots and work on XML data. SnaX provides the isolation level of snapshot isolation [1] and has similar semantics as the concurrency control mechanisms implemented in, e.g., Oracle or PostgreSQL. Both protocols are optimized to XML data in several ways. Firstly, we implement snapshots using an efficient node-based multiversion mechanism. Secondly, we take ancestor/descendant relationships into account when we identify what a transaction reads and writes. Lastly, we use the semantics of the update operations to provide better concurrency.

2 XML

XML documents consist of elements which have a name, can have text and/or elements as children, and can be owners of attributes. In the tree representation of a document, each element, text and attribute is a node of the tree. Element and text siblings are ordered while attributes are not ordered in regard to sibling nodes. Query languages such as XQuery provide mechanisms to express constraints on the structure of the tree as well as on the content. Path expressions (e.g., */site/*/region*) select nodes at specific locations in the tree, and predicates are filters on text or attribute nodes (e.g., *region* = *'africa'*). [9] suggested to extend XQuery to support various types of updates. An update statement identifies via standard path and predicate constraints one or more target nodes for the update. Let p be such a node, and let $tree(p)$ be the subtree rooted at p. (i) *delete(p)* deletes $tree(p)$. (ii) In *replace(p,tree(q))*, if p and q are elements, then $tree(p)$ is replaced with $tree(q)$. For text nodes the content is changed. For attributes, the value of the attribute is changed. (iii) *rename(p,pname)* changes the name of the element or attribute p to *pname*. (iv) *insert-into(p,tree(q))* inserts $tree(q)$ as a subtree below element p. There is no requirement on the position of q in regard to already existing children of p. (v) *insert-after(p,tree(q))* inserts q as a new right sibling of p. Since attributes are unordered, p and q can either be element or text nodes. *insert-before(p,tree(q))* is defined similarly, and hence, omitted from the further discussion.

3 Snapshots

We implement snapshots via a multi-version system. When a transaction T_i starts, it receives a unique identifier $ID(T_i)$. Each node p in an XML tree has a *valid* timestamp $V(p)$ which is the identifier of the transaction that created this node. p also has an *invalid* timestamp $IV(p)$ identifying the transaction that invalidated this node. If no transaction has invalidated p so far, then $IV(p) = NULL$. Inserting a node or subtree creates valid timestamps at the inserted nodes, deleting a node or subtree adds invalid timestamps to the nodes. The replace operator has the effect of a combined delete and insert. For rename, we maintain a stack of names that have been given to the node and associate valid and invalid timestamps with these names.

In order for a transaction T_i to access a snapshot of the tree as of start of transaction, we maintain $EB(T_i)$ as the set of all $ID(T_j)$ such that T_j committed before T_i started plus $ID(T_i)$ itself (T_i can see its own changes). Then, T_i can access p if $V(p) \in EB(T_i) \wedge IV(p) \notin EB(T_i)$.

4 Snapshot based Concurrency Control

In both concurrency control mechanisms that we propose, transaction execution is split into working phase, validation phase, and update phase. Two transactions T_i and T_j are concurrent if T_i started the working phase before T_j finished the update phase and committed, or vice versa. During the *working phase* of transaction T_i all read and write operations are executed on the snapshot accessible by T_i. The read set $RS(T_i)$ denotes the nodes read by T_i and the write set $WS(T_i)$ denotes the nodes written by T_i. Once T_i wants to commit, it goes into the *validation phase*. No two transactions can be concurrently in validation phase. If T_i passes validation, the *update phase* writes the changes performed by T_i into the database and commits T_i. Update phases of different transactions can be executed concurrently. If T_i fails validation, it must abort: new nodes created by T_i are removed, and any invalid timestamps it has set are undone. Since we use snapshots, read-only transactions do not need to perform validation but can commit immediately. Hence, read-only transactions never abort.

OptiX and SnaX differ in their validation of update transactions. OptiX follows traditional optimistic concurrency control. A transaction T_i passes validation if for all concurrent transaction T_j that already validated, $WS(T_j) \cap RS(T_i) = \emptyset$. The motivation is that if T_i passes validation then the execution should be equivalent to a serial execution where T_j executes completely before T_i. In this case, if $WS(T_j) \cap RS(T_i) \neq \emptyset$, then T_i would read changes done by T_j but it does not do so in a concurrent execution.

SnaX provides the isolation level *snapshot isolation* as now offered by many relational systems. Snapshot isolation conforms to the ANSI SQL definition of serializability but is not conflict-serializable in the traditional sense [1]. The strength of snapshot isolation is that it can be implemented without keeping track of reads. Instead, it only handles write/write conflicts. That is, a transaction T_i passes validation if for all concurrent transaction T_j that already validated, $WS(T_j) \cap WS(T_i) = \emptyset$. Since many applications are read-intensive, ignoring reads reduces not only the concurrency control overhead, but also the number of conflicts because only write/write conflicts are considered.

Read and Write Sets One challenge is to identify read and write sets given the nested structure of XML and the various operation types that exist. In order to provide smart conflict detection, we split read set $RS(T_i)$ of transaction T_i into subsets $RR(T_i), ER(T_i)$, and $ER_A(T_i)$. $RR(T_i)$ contains the roots of subtrees returned as part of a query. $ER(T_i)$ contains the nodes that are explicitly read as part of a predicate or path constraint, but that are not returned within the query result. For instance, assume a statement contains the path expression */site/regions*. Then all nodes that fulfill this path expression and their ancestors are in $ER(T_i)$. These nodes belong to the read set, because if someone else changes them, the statement might have a different result. If a path expression contains a wildcard, e.g., */site/regions/*/items*, then nodes identified by '*' are not part of $ER(T_i)$. Finally, $ER_A(T_i)$ includes the nodes that are read for the insertion of a sibling after them.

The write set $WS(T_i)$ of a transaction T_i is split into subsets $D(T_i), Rn(T_i), I(T_i)$ and $I_A(T_i)$. $D(T_i)$ contains the roots of subtrees deleted or replaced by T_i. $Rn(T_i)$ contains the nodes renamed by T_i. $I(T_i)$ contains the immediate parents of any nodes inserted by T_i. $I_A(T_i)$ contains the same nodes as $ER_A(T_i)$. We need this set because transactions using the insert-after operation might require nodes to be placed at a certain position. For example, if elements are sorted in an alphabetical order, then insert-after operations on the same node by two concurrent transactions might lead to inconsistencies, and should be disallowed.

Table 1. Conflict Matrix for OptiX

T_i validating		T_j already validated				
		on p				on q
		D	Rn	I	I_A	WS
on p	RR	-	-	-	$\surd$	-
	ER	-	-	$\surd$	$\surd$	$\surd$
	ER_A	-	-	$\surd$	-	$\surd$
on q	RS	-	$\surd$	$\surd$	$\surd$	

Validation in OptiX Assume T_i wants to validate, and concurrent transaction T_j validated before T_i. Assume a node $p \in RS(T_i) \cap WS(T_j)$. Instead of immediately inducing a conflict, we look at the subsets of $RS(T_i)$ and $WS(T_j)$ of which p is a member and check the conflict matrix of Table 1. $\surd$ indicates compatibility, while - shows a conflict leading to an abort of T_i. A conflict means that if T_i had executed serially after T_j, then it would have possibly read a different value for p than in the concurrent execution. Compatibility means that in a serial execution T_j before T_i, T_i would have read exactly the same value for p. Since $RR(T_i)$ and $D(T_j)$ only contain the roots of affected nodes, the matrix also considers ancestor/descendant relationships whereby p is assumed to be an ancestor of q.

Generally, delete, replace and rename operations conflict with most read operations while insert operations conflict only with few read operations. For space reasons we only discuss the case $p \in I(T_j)$ in detail. If $p \in RR(T_i)$, there is a conflict, because T_i's read-only operation returns the subtree rooted at p. If T_j executed before T_i, then T_i would read the data inserted by T_j, the concurrent execution, however, does not. If $p \in \{ER(T_i), ER_A(T_i)\}$, T_i only reads the node p and not its descendants, and hence, does not conflict with T_j's insert of a new child of p. $q \in RS(T_i)$ does not cause a conflict because T_j's insert of a new child of p has no effect on reading a descendant q of p.

Validation in SnaX For SnaX, we ignore the read sets and only consider write/write conflicts. Table 2 shows the corresponding conflict matrix. Recall that OptiX checks whether in a serial execution T_j before T_i, T_i would read the same values as in the concurrent execution. We cannot perform

Table 2. Conflict Matrix for SnaX

T_i validating		T_j already validated				
		on p				on q
		D	Rn	I	I_A	WS
on p	D	-	-	-	√	-
	Rn	-	-	√	√	√
	I	-	√	√	√	√
	I_A	√	√	√	-	√
on q	WS	-	√	√	√	

such reasoning here since snapshot isolation does not guarantee that there is an equivalent serial execution. Instead, we check whether there is really a write/write conflict. Note that the matrix is symmetric since it does not matter which write operation was performed first.

Generally, delete and replace conflict with most other update types, while inserts conflict only with few. Again, we only look closely at the case where $p \in I(T_j)$. If $p \in D(T_i)$ we have a conflict. A delete or replace spans the entire $tree(p)$ and hence, implicitly conflicts with a concurrent insert into this $tree(p)$. If $p \in \{Rn(T_i), I(T_i), I_A(T_i)\}$, there is no conflict. There is no insert/rename conflict because the insert changes only the subtree below p while the rename only changes p. There is no insert/insert conflict because order does not matter. There is no insert/insert-after conflict because the subtrees are inserted at different locations. Finally, if $q \in WS(T_i)$, there is no conflict since T_j's insert into p does not affect descendant q of p.

5 Evaluation

We integrated both OptiX and SnaX into the native XML database systems McXML [10]. McXML supports a subset of the XQuery language and all XQuery update statements proposed in [9]. For space reasons we only provide a summary of one of the many experiments we performed. This experiment used a narrow and deep tree (each inner node has two to three children and tree height is nine), and analyzed response time and abort rate with increasing throughput for two different workloads. When the workload submitted to the system contained only update operations (0% read-only statements), SnaX performed slightly better than OptiX. Although there are only updates, they include predicate evaluations and path evaluations which read nodes. This leads to higher abort rates for OptiX than for SnaX. However, abort rates are generally small, and hence, the response time of OptiX is only slightly worse than the one for SnaX. With a workload that contains 50% read-only statements, response times are much higher for OptiX than for SnaX. This is due to conflict behavior. Since queries usually read more than one node, the abort rate with OptiX is much higher with 50% reads than with 0% reads. In contrast, SnaX only considers write/write conflicts. Hence, the abort rate is much lower with 50% reads than with 0% reads.

Our other experiments confirmed that SnaX performs generally better than OptiX. However, SnaX does not provide conflict-serializability in the traditional sense. If an application has read/write patterns that lead to many anomalies under SnaX, OptiX might be the better choice.

We also run the experiments with no concurrency control in place (only the multi-version system was active and all transactions committed despite conflicts). The response times for this baseline experiment were less than 3% lower than the response times for SnaX. This shows that our snapshot based protocols have little overhead.

6 Conclusions

This paper presented two snapshot based concurrency control protocols for XML. OptiX extends traditional optimistic concurrency control while SnaX provides snapshot isolation avoiding to keep track of reads. Our protocols consider a wide range of operation types and work on the granularity of individual nodes in order to achieve low conflict rates. Our performance evaluation shows that our protocols have low overhead. Furthermore, SnaX outperforms OptiX due to the simplified handling of read operations.

References

[1] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[2] S. Dekeyser, J. Hidders, and J. Paredaens. A transaction model for XML databases. *World Wide Web*, 7(1), 2004.

[3] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.

[4] T. Grabs, K. Böhm, and H.-J. Schek. XMLTM: efficient transaction management for XML documents. In *CIKM*, 2002.

[5] M. P. Haustein and T. Härder. Adjustable transaction isolation in XML database management systems. In *XSym*, 2004.

[6] S. Helmer, C.-C. Kanne, and G. Moerkotte. Lock-based protocols for cooperation on XML documents. In *DEXA*, 2003.

[7] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *VLDB Journal*, 11(4), 2002.

[8] X. Meng, D. Luo, M.-L. Lee, and J. An. OrientStore: A schema based native XML storage system. In *VLDB*, 2003.

[9] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, 2001.

[10] J. Wu. Updating and indexing XML data. Master's thesis, McGill University, Canada, 2004.