Cooperative Data Management for XML Data

Katja Hose and Kai-Uwe Sattler

Department of Computer Science and Automation,
TU Ilmenau, Germany
{katja.hose|kus}@tu-ilmenau.de

Abstract. Emerging non-standard applications like the production of high-quality spatial sound pose new challenges to data management. Beside the need for a flexible transactional management of complex hierarchical scene descriptions a main requirement is the support of cooperative processes allowing a group of authors to edit a scene together in a distributed environment. Based on previous work on cooperative and non-standard transactions we present in this paper a transaction model and protocol for XML databases addressing this issues.

1 Introduction

In recent years, XML has been widely established as a data exchange format but also as a native data format for (semi-)structured data. XML data management is particularly well suited in application domains where a fixed structure of data is too restrictive and where hierarchical structures have to be represented. In this way, XML data management can be seen as a successor of object-oriented database technology. An example of an emerging application domain from which we derive the motivation for our work presented here is media production, e.g. the production of movies, sounds, and graphics. The authoring process in this domain is typically characterized by

(1) the incremental construction of scenes which are often represented as graphs with varying structures.
For example, sound production in high quality spatial sound systems, like the IOSONO system[1], is based on an object-oriented modelling of scenes. The scene is rendered at runtime to compute the signals for a large number of loudspeakers installed in the listening room. In such a system, a scene consists of several audio objects with properties as well as spatial and temporal relationships describing position, movement, start time and duration of the sound [1]. Scenes are organized in several layers such as dialog, foley, effects, atmosphere and music.
(2) the duration and cooperativeness of the construction process.
In bigger projects we already encountered situations where up to 60 sound designers in up to 8 different studios – located all over the world – are

[1] www.iosono-sound.com

working simultaneously on one sound production project. Each designer is responsible for a part of a scene, e.g. modelling special effects, while others model the actors' dialogs. Both groups must know the current state of the other group's work because for example the volume of a speaker has to be set with respect to the background noise.

Basically, (1) can be addressed by representing a scene as an XML graph stored in a database. Today, commercial DBMS provide support for XML data management, either by treating XML documents as CLOB objects or by "shredding" the document into a set of tables. In both cases, the transactional support is restricted to the native (relational) structure and is not adopted to the special characteristics of (hierarchical) XML data.

The alternative of using a native XML DBMS promises a better handling of XML data, e.g. wrt. querying, but regarding transactional support these systems are more or less still in their infancies.

Orthogonal to the scheme of storing the scene data we can look at the issue of supporting cooperative work (2) in different scenarios:

Single User: The simplest and in fact non-cooperative case is the situation where a single author (sound designer) is working on the sound production. Though, basic database functionality like persistence and recovery is required, advanced techniques for distributed and cooperative operation are not needed.

Workgroup: When multiple designers work on the same scene together in a workgroup, their updates have to be synchronized. In this scenario a central repository is needed but can be extended by local caches or databases for better response times.

Workspace: In a large-scale scenario multiple studios (maybe situated on different continents) work on the same scene data. Here, permanent connections to a central repository (as in the case of the workgroup scenario) may not be assumed. Therefore, the different users work on their own workspaces requiring a decoupled synchronization technique similar to version control systems like CVS.

In our work we focus on the workgroup scenario because it is the most interesting one for the intended application of sound production. In any case, the notion of transaction is required and because of the characteristics of work we need long running, nested transactions. Furthermore, in order to allow a cooperative work on the same scene (e.g. on different layers of a scene) modifications should be visible to other authors before the end of the global transaction.

Based on these observations and the above mentioned challenges of data management in media authoring processes, we present in this paper an approach using an open-nested transaction model for XML data supporting cooperative processes. The remainder of the paper is structured as follows. In Section 2 we give a brief survey on related work. Based on this, we introduce in Section 3 our cooperative transaction model. Next, we describe the protocol implementing this model by a combination of locking and notification in Section 4. Finally, we conclude the paper and point out to future work.

2 Related Work

For many years transactions have been used to enable concurrent access to a database where each transaction adheres to the ACID properties. A very common means to enforce serializability are lock protocols. They have advantages like low complexity and disadvantages like limiting concurrency. Thus, many variants have been developed; 2PL [2] and hierarchical locks [3] to mention only a few.

In order to overcome the disadvantage of limited concurrency advanced transaction models have been developed by relaxing some of the ACID properties. One principle is to divide a transaction into several smaller ones. This concept is for example used by nested transactions [4,5] and multi-level transactions [6,7]. With the advent of CAD systems with typically long transactions chained transactions and sagas [8] have been proposed. Whereas both of these concepts require the definition of subtransactions in advance, split/join transactions [9, 10] can be used to determine subtransactions at runtime.

As XML became more important DBMS were extended by XML modules. At first, it was sufficient to support XML as input and output format. But since this limits querying abilities, first concepts for native XML databases have been developed. In general, classic lock protocols are too restrictive in terms of concurrency for use in XML DBMS. One possibility is to use path based locks as proposed by several groups [11–14]. Natix [15] is a native XML DBMS that recognizes that transaction management needs a non-traditional approach. However, the authors focus more on recovery and isolation, and use a lock manager supporting multi granularity locking and strict two-phase locking.

Another possibility to realize concurrent access are protocols based on the taDOM model, e.g., [16, 17]. They use multi granularity locking, apply the concept of intensional locks on the path from the root node to the context node, and provide lock conversion. taDOM models attributes and text content as special nodes. By this, attributes or text can be locked without locking the original XML node as a whole. Lock granularity and lock escalation can be adapted according to the users needs.

As indicated in the introduction we need to allow multiple users to work on the same XML document simultaneously. The approaches on XML databases mentioned above do net (yet) pay attention to the requirements imposed by our scenario where we have deal with long transactions adhering to an open nested transaction model.

3 Transaction and Cooperation Model

The basic architecture of our solution is a client-server-model where the server holds the latest version of the data and coordinates the clients. An arbitrary number of clients can be connected to the server. Both client and server have a DBMS to manage their data. The server maintains the global copy ensuring data consistency. The client uses its local DBMS to manage its local copy of the portion of the data that it downloaded from the server. Since we are working

with XML data, we chose Berkeley DB XML[2] for both client and server. For the local DBMS on client side, we can use any standard transaction model since only one user is working on the local copy. Using the DBMS, however, the client may also benefit from database functionality like persistence and recovery.

In the following we at first identify a transaction model that fulfills the requirements stated in the introduction. Then, we describe how notifications can be used to make non-committed updates visible to other users.

3.1 Transaction Model

As motivated in the introduction transactions may endure a rather long time period. In order to support concurrent access to the common global database we need a transaction model that efficiently supports long transactions but still allows multiple users to work concurrently on the same data. This suggests the use of a nested transaction model [4, 5] where a global transaction is divided at runtime into several subtransactions. In our model a subtransaction is not vital for the global transaction. Thus, when a subtransaction is aborted not all changes have to be undone but only those that have been made by the aborted subtransaction. The global transaction goes on until the global commit or the global abort.

	Client	**Server**
user-level primitives	query	checkout
	update	checkin
	savepoint	subcommit, publish, subbegin
	revert	subabort, subbegin
	begin, abort, commit	begin, abort, commit
system-level primitives	lock, unlock	lock, unlock
	subscribe, unsubscribe	subscribe, unsubscribe

Table 1. Primitives at Client and Server

Having started a transaction the user issues commands on the client side. The client communicates with the server to initiate corresponding actions. Table 1 lists the main primitives that are supported by client and server instances. They can be divided into two groups: user-level primitives and system-level primitives. The former are issued more or less by the user himself, the latter by the system transparently to the user.

Figure 1 illustrates a sample sequence of primitives that are issued in a typical transaction. When the user starts to work a global transaction is started (*begin*). Since we are using a nested transaction model, the first subtransaction starts at the same time (*subbegin*). The *query* primitive allows the user to read elements of the XML document that are identified using XPath. Hence, this primitive corresponds to the classic read operation. Its counterpart, i.e. a write operation, is represented by the *update* primitive. The *query* primitive results in a *checkout* at the server retrieving the latest version of the read data. The *update* primitive of the client is realized by the *checkin* primitive at the server. This, however,

[2] http://www.sleepycat.com/products/bdbxml.html

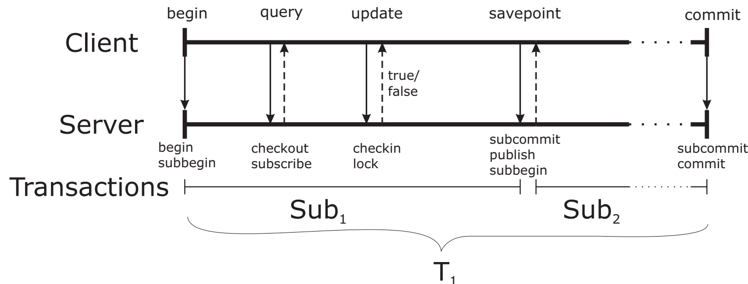

Fig. 1. Basic Sequence of Actions for Transactions

requires another step: first the server is trying to *lock* the data that is to be updated. If the lock has been acquired the data is updated, if not the attempt to update is refused.

When the user decides for a *savepoint* the currently running subtransaction commits and the next subtransaction is implicitly started. All changes that have been made by the first subtransaction are now visible to other users – but may not yet be changed by them. In case a subtransaction is aborted, all changes that have been made since the beginning of the subtransaction are undone without any effect on the global transaction. Finally, when the global transaction commits, the current subtransaction is committed and all locks are released.

3.2 Cooperation Model

As mentioned above other users might need to see the changes of committed subtransactions before the global transaction submits. In order to achieve such a cooperative transaction model we use the concept of notifications. For this purpose the server maintains a list of *Listeners*. Each entry is defined by two pieces of information: the identifier of the corresponding client and an XPath expression that indicates a subtree of the XML document. Whenever a subtransaction of another client commits, all clients that have registered for the affected data are notified using the *publish* primitive (see Figure 1). When the client reads data in a transactional context using the *query* primitive the client is implicitly registered at the server for updates concerning the read data.

Figure 2 illustrates this concept in a situation where two transactions are working on the same version of the data. Both transactions start with checking out version $V1$ from the server, they implicitly register for updates concerning the retrieved data. Both transactions ($T1$ and $T2$) are now starting to work concurrently. When subtransaction $Sub1.1$ of $T1$ commits, the changes are propagated to the client that $T2$ is running on. The client updates the affected portion of local data. Afterwards, the data of both clients corresponds to subversion $V1.1$. By the use of our locking protocol that we use for synchronization (Section 4) it is not possible that both transactions changed the same data records.

In case another transaction $T3$ would now checkout the current version from the server it would retrieve the basic version $V1$ and all updates of already committed subtransactions so that the local version of $T3$'s data would correspond to subversion $V1.1$.

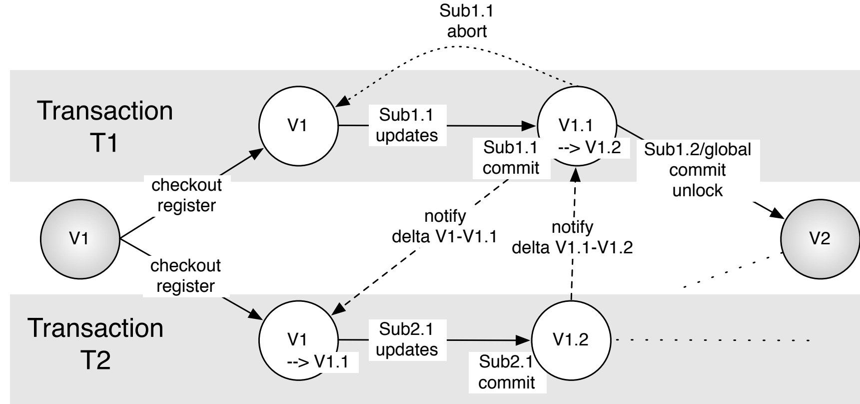

Fig. 2. Example for Versioning Using Listeners

After subtransaction $Sub1.1$ of $T1$ has committed assume that now subtransaction $Sub1.1$ of $T2$ commits. Then, the client that $T1$ runs on retrieves the updates made by $Sub2.1$ of $T2$. Both clients now have the same subversion $V1.2$. Finally, the global transaction $T1$ commits. The server now retrieves the changes, releases the locks still held by $T1$, and converts the global version to $V2$. After having notified the client that $T2$ runs on, the corresponding client updates its local data, and now works on version $V2$. In case $T2$ now aborts, then the changes made by the already committed subtransactions of $T2$ are undone and the registered clients are notified. In case $T2$ submits the the server creates version $V3$ and notifies the registered clients.

4 Transaction Protocol

In this section we at first identify operations that need to be supported and their demands on a suitable locking protocol. Then, we present such a protocol that we use for synchronization. Finally, we show how to increase concurrency by using a special internal representation.

4.1 Operations and Lock Compatibility

The set of relevant operations that we need to support consists of: (i) edit, (ii) delete, (iii) add, and (iv) move. XML documents consist of nodes organized in a hierarchical structure. Each node may contain text and attributes and might have attached child node. To change the content of a node's text or attribute, the *edit* operation is used. Obviously, this requires an exclusive lock so that no other user is able to edit attributes and text at the same time. The *delete* operation can be used to delete attributes, nodes, text, or whole subtrees that are attached to a given node n. Since the deletion of attributes and text directly affects n, we need n to be locked exclusively. In case we want to delete a node n_1 (or a subtree rooted with n_1) that is attached to n as a child, we only need a shared lock on n and an exclusive lock on n_1 (and on all subtree nodes).

The *add* operation can be used to attach attributes, text, child nodes, or subtrees to node n. Since text and attributes are an integral part of n we need

an exclusive lock on n for adding text and attributes. Intuitively, all we need to attach child nodes or subtrees to n is a shared lock on n. Thus, adding a child node n_1 and deleting another child node n_2 (both children of n) by two concurrent transactions is possible at the same time.

The *move* operation can be used to move attributes, nodes, text, and subtrees from node n to node m. We can treat this as a deletion followed by an insertion so that we can use the two operations introduced above. Table 2 shows the lock matrix that results from the considerations discussed so far.

	edit	**delete**	**add**
attribute	XL on node	XL on node	XL on node
text	XL on node	XL on node	XL on node
node	–	XL on node, SL on parent	SL on parent
subtree	–	XL (root and descendants), SL on parent	SL on parent

Table 2. Operations and required locks when attributes and text *cannot* be locked apart from the corresponding node – XL = Exclusive Lock, SL = Shared Lock

This is still rather restrictive. Under the assumption that we can lock attributes and text independently from the corresponding XML nodes, we can achieve a higher level of concurrency: e.g. concurrent operations on attributes and text are now compatible with each other. Furthermore, these operations are compatible with concurrent deletion and addition of child nodes. Table 3 shows the resulting lock matrix.

	edit	**delete**	**add**
attribute	XL on attr.	XL on attr., SL on corr. node	SL on corr. node
text	XL on text	XL on text, SL on corr. node	SL on corr. node
node	–	XL on node, SL on parent	SL on parent
subtree	–	XL (root and descendants), SL on parent	SL on parent

Table 3. Operations and required locks when attributes and text *can* be locked apart from the corresponding node – XL = Exclusive Lock, SL = Shared Lock

Since we have to deal with only two kinds of locks (exclusive and shared), we can apply the tree protocol that has been developed for hierarchical databases. Thus, the remainder of this section first presents the tree protocol. Then, we point out how to manage locks on attributes and text without locking the whole node.

4.2 Tree Protocol

In contrast to most lock protocols the tree protocol [3] does not imply two phase locking (2PL). It has been designed for use in hierarchically organized data structures and thus can be used for XML data as well. The basic variant of this protocol only knows one kind of locks: exclusive locks. The advanced variant – that we consider – also knows non-exclusive locks. Any transaction T_i that adheres to the following rules satisfies the advanced tree protocol and leads to a serializable schedule:

1. At first, T_i locks any node of the hierarchy – provided that any existing locks are compatible
2. T_i locks each node at most once – there is no lock conversion
3. locks may be released at any time – in contrast to 2PL protocols
4. T_i may lock node u if and only if it is currently holding a lock on a predecessor (father) of u
5. $LS(T_i) \subseteq \mu(L(T_i))$, where $LS(T_i)$ is the set of shared locks held by T_i, $L(T_i)$ the set of all locks (shared or exclusive) held by T_i, and $\mu(W) = \{v \in W | \textit{there exists at most one } w \in W \textit{ such that } v \textit{ and } w \textit{ are neighbors}\}$

Note that if T_i satisfies conditions $1-4$ it fulfills the basic tree protocol [18]. The fifth condition assures that deadlocks cannot occur when allowing non-exclusive locks.

With respect to Section 4.1 using the tree protocol means that locks are acquired based on the tree protocol and may be released before the commit. However, in our implementation those locks that are required by the supported operations (Table 3) need to be held until the global transaction commits. Only then can be guaranteed that a rollback is possible without side-effects on other transactions.

4.3 XML Representation

In order to improve concurrency on XML nodes especially with respect to editing attributes and text, we apply the taDOM concept [19] and adopted it to our needs.

Representing XML Documents for Fine Grained Locking taDOM [19] has originally been designed for supporting a fine grained lock granularity for documents that are accessed by the DOM API. In contrast to our application and access methods the DOM API knows functions like *getAttributeNode()*, *getValue()* etc. In order to support these functions efficiently, the authors split up XML nodes into several parts, e.g. each XML node is represented by one element node n_e, one text node n_t (as child of n_e) and one string node n_s (as child of n_t). If the original XML node contained attributes then an additional attribute root node n_{AR} is inserted as child of the n_e. For each attribute, n_{AR} has one attribute child node n_{a_i} where each n_{a_i} has a string node that contains the attribute value.

Since we do not aim to support such APIs we do not need such a large number of nodes. Thus, we reduce the overhead by simply splitting up an XML element node into one element node n_e, one child for the contained text n_t, one attribute root node n_{AR} with one attribute child node n_{a_i} for each attribute – n_{a_i} contains all information about the attribute. In short, we keep text nodes and string nodes together in one node. The separation of attributes from the original node has two advantages: first, locking attributes apart from their nodes is possible and second, locking all attributes at once promises to be low effort.

Figure 3 illustrates these concepts with an example. It shows a small extract of a sample XML file and its corresponding representation where we distinguish

between *element nodes*, *text nodes*, *attribute root nodes*, and *attribute nodes*. Inner nodes of XML documents are represented by element nodes.

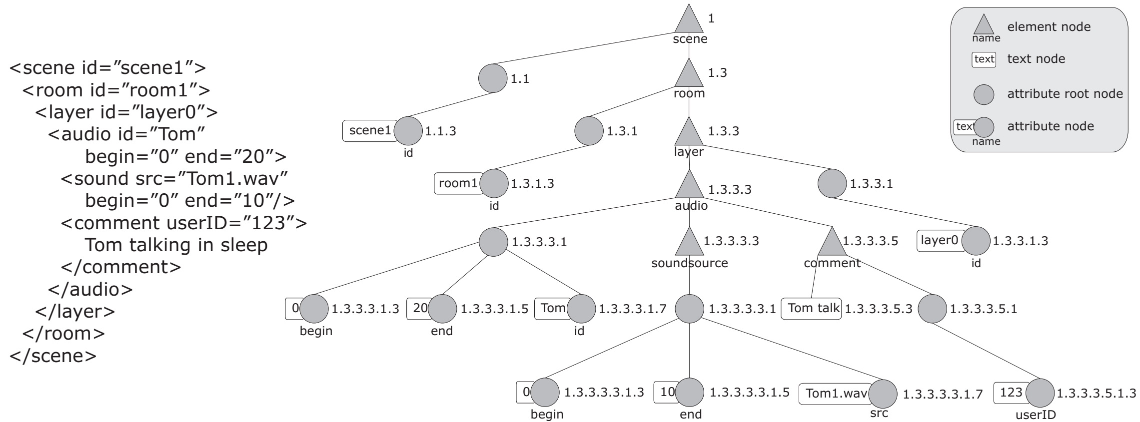

Fig. 3. Internal XML Representation

DeweyIDs By using DeweyIDs [20] we assign a unique identifier to each node and thus enable an efficient management of read and write sets. As we have borrowed the idea of separating attributes from elements and using an extra attribute root node to group them, we also adopt most of the adaptations of DeweyIDs that were made to support taDOM. DeweyIDs are based on the decimal classification and serially number all nodes in the same level with odd numbers. With the exception of the document root node, the number 1 is only assigned to attribute root nodes. This makes finding and identifying attributes easy. The ID of a node is defined as the conjunction of the parent-ID and the assigned number separated by a point. Thus, the prefix of each ID reveals its level and unambiguously identifies the parent node and all ancestor nodes. Since initially only odd numbers are used new nodes may be added at any position. For instance, between DeweyIDs 1.3 and 1.5 we may add nodes with the following IDs: 1.4.3, 1.4.5, etc. Figure 3 gives an example for the DeweyID numbering scheme that we use in our implementation.

5 Conclusion

In this paper we have addressed the problem of concurrent access and modification to XML documents. The application scenario demands that users may see changes of other users whose global transactions have not yet committed. To solve this problem we proposed an open nested transaction model that uses the advanced tree protocol for synchronization. Notifications take care of propagating recent updates to registered clients, so that they are always up-to-date. Future work will consider to further increase concurrency. One possibility to achieve this is releasing locks already with the commit of a subtransaction so that other transactions may update the same elements before the global transaction submits. Introducing compensating transactions might be a solution to the

problem. However, it remains a task for future work to define such compensating transactions and maybe determine their actions without extensive interaction with the user.

References

1. Heimrich, T., Reichelt, K., Rusch, H., Sattler, K., Schröder, T.: Modelling and streaming spatiotemporal audio data. In: OTM Workshops. (2005) 7–8
2. P. A. Bernstein and V. Hadzilacos and N. Goodman. In: Concurrency Control and Recovery in Database Systems. Addison-Wesley Publishing Company (1987)
3. Silberschatz, A., Kedem, Z.: Consistency in hierarchical database systems. Journal of the ACM **27** (1980) 72–80
4. Härder, T., Rothermel, K.: Concepts for transaction recovery in nested transactions. In: SIGMOD '87. (1987) 239–248
5. Moss, J.E.: Nested transactions: an approach to reliable distributed computing. Massachusetts Institute of Technology, Cambridge, MA, USA (1985)
6. Weikum, G.: Principles and realization strategies of multilevel transaction management. ACM Transactions on Database Systems **16** (1991) 132–180
7. Weikum, G., Schek, H.J.: Concepts and applications of multilevel transactions and open nested transactions. Database transaction models for advanced applications (1992) 515–553
8. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD '87. (1987) 249–259
9. Kaiser, G., Pu, C.: Dynamic restructuring of transactions. In: Database Transaction Models for Advanced Applications. (1992) 265–295
10. Pu, C., Kaiser, G., Hutchinson, N.: Split-transactions for open-ended activities. In: VLDB '88. (1988) 26–37
11. Choi, E., Kanai, T.: XPath-based Concurrency Control for XML Data. In: DEWS 2003. (2003)
12. Dekeyser, S., Hidders, J.: Path Locks for XML Document Collaboration. In: WISE '02. (2002) 105–114
13. Dekeyser, S., Hidders, J.: Conflict scheduling of transactions on XML documents. In: ADC '04. (2004) 93–101
14. Jea, K., Chen, S., Wang, S.: Concurrency Control in XML Document Databases: XPath Locking Protocol. In: ICPADS '02. (2002) 551
15. Fiebig, T., Helmer, S., Kanne, C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Natix: A Technology Overview. In: Web, Web-Services, and Database Systems. (2003) 12–33
16. Haustein, M., Härder, T.: Adjustable Transaction Isolation in XML Database Management Systems. In: In Proc. 2nd Int. XML Database Symposium. (2004) 173–188
17. Haustein, M., Härder, T., Luttenberger, K.: Contest of XML Lock Protocols. In: VLDB 2006. (2006) 1069–1080
18. Kedem, Z., Silberschatz, A.: Controlling Concurrency Using Locking Protocols (Preliminary Report). In: FOCS. (1979) 274–285
19. Haustein, M.P., Härder, T.: taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In: ADBIS. (2003) 88–102
20. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: insert-friendly XML node labels. In: SIGMOD '04. (2004) 903–908