A Transaction Synchronization Protocol for XML Database in Web Architecture

Morshed Osmani and Syed M. Rahman
Department of Computer Science, North Dakota State University
258 IACC Building, Fargo, North Dakota 58105, USA
Email: {Morshed.Osmani, Syed.Rahman}@ndsu.edu

Abstract

Transactions synchronization in high throughput database demonstrates two important characteristics: consistency is maintained and semantic knowledge can potentially be added in order to enhance concurrency. In this paper, we have proposed a synchronization protocol for transactions on client-server model using XML database and some associated enhancement to existing databases. In contrast to other pessimistic locking algorithms, our client works without creating a lock in the database and the transaction requires lock only at the commit phase in the server side.

We create a wrapper around the DBMS to facilitate this synchronization process. This wrapper helps to reduce abort rate, deals with loss of connection, and works with various DBMSs with XML support as well as legacy databases. In this protocol, a transaction is started optimistically at the client side. At the commit phase the wrapper works closely with the DBMSs scheduler and transaction manager. Our proposed protocol enhances concurrency and reduces conflicts.

1. Introduction

Whenever XML data are modified by several clients over the web, transaction synchronization requires some extra attention because traditional synchronization protocols create frequent abort and takes too long. In this paper, we have acknowledged this problem and proposed a protocol dealing concurrent access efficiently. Let us explain the problem with an example: an online server is used to store course materials, constructing exam papers, taking test etc. Here several clients use the central server from different parts of the world at the same time. They request documents or data from the server, modify or update the data, and send back to the server for update and storage. Some of the clients may work on the same document but in different parts. So synchronization is one of the vital requirements. An important characteristic of this system is that client requests for data at the beginning of the transaction and submits the data to the server for update at the end of the transaction. This may cause a transaction to be in progress for a long period which may be inappropriate for most of the databases. Another problem may arise. In a typical web system, lose of connection is a common scenario. Thus a client may lose the connection with the database in the middle of the transaction which may force the transaction to abort. To minimize the effect of all these problems we propose the new method of transaction synchronization. It helps us to reduce the transaction failure rate, as well as makes possible the use of XML database in such a scenario. Our approach creates a wrapper on existing DBMS to support our protocol without lessening the power of DBMS. Rather it increases the transaction commit rate and parallelism of the server and transfers some load from server to client.

2. Related works

Though XML is a relatively new area, a great number of researchers are working on it. Several researchers convert the existing relational database system to XML documents as well as converting XML data to relation model[1][2][3]. Other approaches include using a RDBMS to store and retrieve XML data with a limited support of XML built into the DBMS. Some are working to develop a full-featured Native XML Database [4][5][6] to be built from scratch to support all the required feature of a database as well as the all aspects of XML. XML as a data exchange format is also a research topic for many researchers. Our approach can use either Native XML Database or a RDBMS (with XML support) as database.

A few other researchers use XML as data exchange format as well as storage [8] and to synchronize the transaction in web architecture [7]. However, our protocol differs from others in several ways. First of all, Böttcher and Türling [7] use XML as only data exchange format. It generates XML document from the legacy database when users requests for data and at the commit phase updated XML document is transferred to the relational database values again. But our protocol uses Native XML database which uses XQuery [10], can ship XML data, and can update its XML values using XUpdate [9] document. The client behaves almost similarly with a few modifications; however, the server side is very much different than Böttcher and Türling approach. Our approach

gives a more flexible method to use XML databases in that situation as well as legacy database with some additional supports.

3. The Model

3.1 Important features of the proposed model

Our proposed model is based on Native XML Database primarily. We use XUpdate to send update information to server, and DOM for managing XML data at the client. A transaction is divided into three major parts; client side, top level transaction, and transaction at the underlying database. An important key feature of this protocol is to distinguish between *browse* data and *committed read* data. This feature eliminates the need for locking the whole data/document requested by the client. Other important features include the short time of actual transactions which greatly improves the failure rate of transactions which are caused by long duration of the transactions.

3.2 The server

In our proposed protocol, the server is built on an existing XML database. It has two parts; one is a DBMS which may be a Native XML Database or any RDBMS with XML support and the other one is a wrapper that consists of a top level transaction manager and a Read-Log. Native XML database provides a higher degree of concurrency which may not be found in other RDBMS in using XML data. The top level transaction manager is very simple compared to the DBMSs transaction manager. The top level transaction manager creates a new transaction at its level when client request a document; however, it initiates a transaction at DBMS when it gets a commit request from the client.

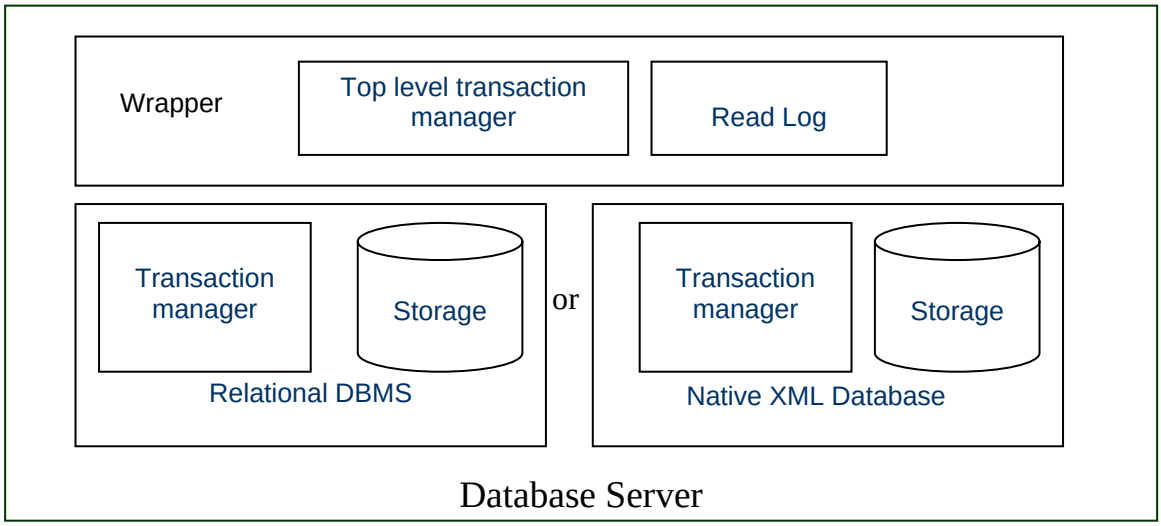

Figure 1: Schematic view of the server

3.3 Transaction Protocol

Our proposed transaction protocol of the model is divided into the following three parts:

1. Client side.
2. Server side-top level.
3. Server side-lower level.

In this model, majority of work is done in the client side. The client sends query to the server, receives the result-set/document from the server, performs operations on the XML document, and finally submits the transaction to the server to commit. The server tries to commit the transaction and then sends the result indicating commit status. Thus, the client plays a major role in committing the transactions. On the other hand the server maintains the original database, ships data to the client as XML documents, and also maintains a read-log. The server has a Native XML DBMS (or a RDBMS with XML support) with a second top level transaction manager sitting on the database. Server side transaction is performed in two steps. The top-level transaction manager decides whether it will send the transaction to the DBMS's transaction manager to commit. If the decision is positive, a transaction at lower level is initiated. Client is informed about the status of the transaction.

3.4 Server side Operations at the beginning of the protocol

When a client requests data from the server through XQuery (or possibly in other query language that supports XML), the server returns an XML document containing the query result. At the same time, the top level transaction manager initiates a transaction at its level, does necessary book keepings, and updates it Read-Log. Query operation is accomplished by a *RequesData* call to the server with the client-id, query and query-type as parameter. The server executes the following procedure in response to the *RequesData* call:

```
RequestData (Client-ID, Query, Query-Type)
{
        Result = ExecuteQuery(Query, Query-Type) ;
        TrID = CreateTransation(Now, Client-ID)
        WriteReadLog(Client-ID, TrID, Result ) ;
        SendQueryResult (Client-ID, Result) ;
}
```

Here *Result* is a valid XML document which is sent to the client for modification. Before sending data to client, the server updates its Read-Log by calling WriteReadLog(Client-ID, TrID, Query). The server maintains the Read-Log to identify current read sets of different clients. It does not lock those data present in *Result*, but only keeps necessary information about them. When a client submits a transaction and the server attempts to commit it, the update (delete, update) sets are compared with the data in Read-Log. If there are some matching elements in the two sets, the server sends a message with the

updated data to the corresponding clients. Read-Log is also consulted when the client requests for the transaction to commit.

3.5 Client side operations on XML documents

The client may perform the following operations on XML elements or attributes of the XML document returned by the server: *browse*, *read committed*, *read data changed (insert*, *delete*, and *update)*. The client uses a DOM to hold the XML document in memory. This implementation has a flexible approach. We can just set status of the node to reflect the corresponding operation. Later at the time of commit, we derive an XML file containing XUpdate information from the DOM.

The synchronization protocol can be viewed diagrammatically as:

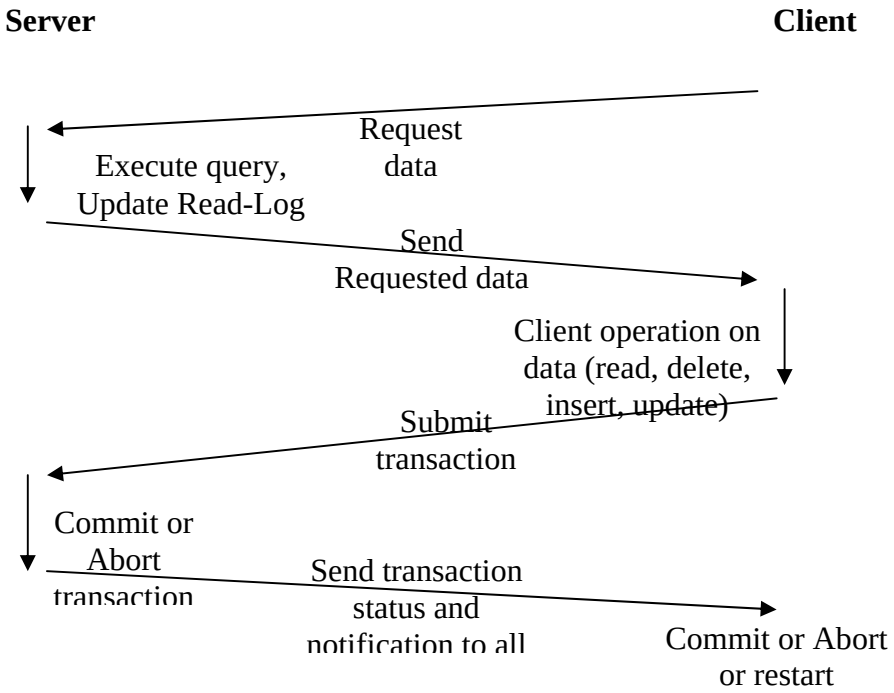

Figure 2: Synchronization Protocol

3.6 Committed read

Committed read data are those data that influence other update information. Consider X and Y are two elements (or attributes) in the XML document. If *Y* value is changed to 3**X* then we consider *X* in committed read set and *Y* in update set. If *Y* is set to *Y +3*X* then we put both *Y* and *X* in the committed read set and also new value of *Y* in update set. This committed read set is compared with the Read-Log of the top level transaction manager of the server for possible new values updated between the period of transaction

start and transaction commit request. Typically only a small part of the XML document is changed and these changes may depend upon only a small section of the document. Committed read set helps the transaction manager to check for inconsistency at early stage and thus helps to abort a transaction without reporting the DBMS. Read set is generated by client and used by the server.

If a part of the XML document is not relevant for any modification and is not used for any other processing, then it belongs to the browse set. We do not need to consider this set further.

3.7 Changes to the document being made

In case of updating a value Y, if new value of Y does not depend on any other node, Y is marked as updated in DOM. Otherwise all the value on which Y is dependent is added to the committed read set.

Similarly if Y is deleted without any condition, it is marked as deleted. However if deletion of Y depends on some values of other elements, they are added to the committed read set.

If there is a conditional insertion, all the values on which this insertion depends are added to the committed read set. Otherwise no change is made to the committed read set. Just the data is inserted to the DOM and it is marked as inserted.

3.8 Transaction phases

Each client's transaction is performed in two phases: a browse phase and a subsequent commit phase. The browse phase terminates with the web client's request to commit the transaction. This essentially starts the commit phase.

3.9 The *browse* phase

In the browse phase of a transaction, the client accesses XML elements (or attributes) in the document returned by the server. The client application may 'browse', 'read committed' or 'write' on the XML document. The client's transaction manager marks the element in DOM with corresponding tags (updated, inserted, deleted) and collects the committed-read set.

When the transaction needs to be completed, it first transforms its updates in DOM to an XUpdate XML document and committed read set to another XML document named Read-document. Then client sends request to server to commit the transaction using a *commitRequest* call with committed read set and XUpdate document as parameters.

serverResponse = commitRequest(Client-ID, Read-document, XUpdate-document);

If the server returns *success,* then the client transaction is committed too.

3.10 The server-side action for *commitRequest*

The server implements *commitRequest* as a two stage operation. The Read-document is checked with the Read-Log to see if there is any modification to these values since the inception of the current transaction. If database values do not conform to Read-document values, the transaction is aborted. Otherwise, a transaction is created in the low level, i.e. at the DBMS. Changes in XUpdate-document are passed to the DBMS using corresponding interface. In case of DBMS's transaction failure, (which may be caused by an integrity constraint violation or a deadlock) the top level transaction manager aborts the transaction, otherwise the transaction is committed. Finally, the top level transaction manager signals either commit or abort to the client. After committing the transaction the server checks it's update-set with Read-Log and sends notification to the corresponding clients with the updated values. It is up to the client (application) whether it will proceed its current transaction with updated value or abort current transaction and start a new one with updated values.

3.11 Algorithm for server side commits operation

```
CommitRequest (Client-ID, Read-document , XUpdate-document )
{
    trID = getTrID(Client-ID); // get the transaction id created previously
    if (getTransactionTime(trID)> TIMETOLIVE)
      if (hasChanged(trID,Read-document))
      {
            abortTransaction();  // aborts the transaction
            return Failure; //return status as failure
      }
      else
      {
            dbTrID = createTransactioninDBMS();  // create a
                        //transaction at the low level DBMS
            sendUpdates(dbTrID, XUpdate-document)
                  // send the updates to DBMS to perform the transaction.
            if ( integrity violation or other failure )
            {
                  abortTransaction();  // aborts the transaction
                  return Failure; //return status as failure
            }
            else
            {
                  commitTransaction();  // commits the transaction
                  Compare XUpdate-document with Read-Log and sends
```

```
                                notifications to the corresponding clients.
                        }
                    }
                return success;   //return status as success
            }
```

In case of a server side commit, the client transaction is also committed.

4. Evaluations of the Protocol

Our model differs from Böttcher and Türling approach [7] in several ways. Our approach uses a Native XML Database or other relational database with XML support. This enhances the concurrency level of the transactions at lower level. The participation of client in transaction also release loads from the server. Isolation level is high as each client works on its data independently of other client.

Another great advantage of our approach is that it does not need to acquire lock at the start of the transaction when initiated by client. Rather high level transaction manager only request lock from the low level DBMS at the time of commit at low level. Transaction time period at low level database is very low as compared to total transaction time and we believe that the range would be around 0.001% to 5% of total transaction time. Thus our model only requires lock for a very short period as compared to other protocols. Since the low level transaction manager is not aware of the web transaction time at high level, the long transaction time at high level does not affect the corresponding low level transaction and this reduces the abort rate to a lower value which may be higher in other model due to only long transaction time.

In any web architecture the connection loss is a typical scenario and our model can handle it efficiently. At the start of transaction at top level, the manager allocates maximum allowed time period for the client to submit the transaction for commit. If a client fails to submit a transaction within that period, it is assumed that the connection is totally lost and transaction is aborted. However, there is no need to continue the connection between query-request and commit-request. Any stateless protocol like HTTP can be used to communicate between the client and the server.

The wrapper part of the server which comprises only a top level transaction manager and a Read-Log. The wrapper does not need to be complex to handle all the situations arise in a database. Even it does not need to concern about the ACID properties of a server. These properties are handled by the low level DBMS. The wrapper part only need to deal with creating a transaction at top level, to perform some book-keeping tasks on transaction, to maintain a Read-Log, to send appropriate commands to the DBMS, to get output or response from the DBMS, and act accordingly. This works are not too complex as compared with a DBMS implementation. Whether we want to use Native XML Database or a RDBMS with XML support, we only need to modify a small section of the wrapper. This wrapper can be used in a wide selection of the DBMS.

More interestingly our model can work with a legacy database with no support for XML. In this case, we need to add a middleware to convert XML data into relational data and vice versa. We need to add some interfaces with which the middleware can communicate with the wrapper.

5. Conclusions

In this paper, we have presented a synchronization protocol for transactions on client-server model using XML database and some associated enhancement to existing databases. In contrast to other pessimistic locking algorithms, our client works without creating a lock in the database and the transaction requires lock only at the commit phase in the server side.

We have used a wrapper around the database to facilitate the transaction initiated from web client. This wrapper helps to reduce the locking problem, low commit rate and keeps the transaction alive in case of connection loss. The wrapper also provides interface to work with any Native XML Database, RDBMS with transaction support or any other middleware that works with a legacy database. Our proposed protocol's wrapper gives a high level view with a consistence interface to all clients irrespective of underlying database.

The transaction may fail for several reasons such as the server can abort it or client can abort it itself. A long time gap between client's query request and commit request exceeding maximum allowed time may cause the transaction to abort. In all those cases, the client is responsible for taking appropriate steps. In case of a server side abort, client may either decide to withdraw its transaction or may decide to refresh the document and then retry a commit request. In case of multiple failures, the client may want to reduce its committed read or write part and declare more data to be in browse set and try to commit again. We believe that our proposed protocol improves the performance of the system compared to other existing implementations. However, we do not have any simulation result to support our claim which is considered as our future work. We would like to also develop a common framework for the database wrapper so that it can easily be integrated with other DBMS.

6. Reference

1. Gerti Kappel, Elisabeth Kapsammer and Werner Retschitzegger: Integrating XML and Relational Database Systems, World Wide Web: Internet and Web Information Systems, 7, 343–384, 2004.
2. Bourret, R., Bornhövd, C., Buchmann, A.P.: A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases. 2nd Int. Workshop on Advanced Issues of EC and Webbased Information Systems (WECWIS), San Jose, California, June, 2000

3. Fernandez, M., Tan, WC.,Suciu, D.: SilkRoute: Trading between Relations and XML. 9th Int. World Wide Web Conf. (WWW), Amsterdam, May, 2000
4. SALMINEN, A. AND TOMPA, F. 2001. Requirements for XML document database systems. In Proceedings of the ACM Symposium on Document Engineering 2001 (DocEng '01, Atlanta, GA).
5. T. Fiebig et al. Anatomy of a native XML base management system. VLDB Journal, 11(4):292{314}, 2002.
6. Sipani, S., Verma, K., Chandrasekaran, S., Zeng, X., Zhu, J., Che, D., and Wong, K. (2002)' Designing an XML Database Engine: API and Performance', in Proceedings of the 40th Annual Southeast ACM Conference, Raleigh, NC, pp. 239-245.
7. S. Böttcher, A. Türling: Transaction Synchronization for XML data in Client Server Web Applications, GI-Jahrestagung, Wien, 2001.
8. Manirupa Das, Pamela B. Lawhead : Information storage and management in large web-based applications using XML, Journal of Computing Sciences in Colleges, Volume 18 , Issue 6, 2003
9. XML:DB initiative (http://xmldb-org.sourceforge.net/), web retrieve on March 10, 2006
10. XQuery 1.0: An XML Query Language (http://www.w3.org/TR/xquery/), web retrieve on March 10, 2006