Transaction Synchronization for XML Data in Client-Server Web Applications

Stefan Böttcher , Adelhard Türling
University of Paderborn
Fachbereich 17 (Mathematik-Informatik)
Fürstenallee 11 , D-33102 Paderborn , Germany
email : stb@uni-paderborn.de, Adelhard@Tuerling.de

Abstract:

Whenever database centered client-server web applications have to be used by multiple web clients on different platforms, then recently XML has been considered as an important data exchange format. If multiple web clients modify their XML copy of some server side database data, then writing this modifications back to the server side database requires transaction synchronization. We present a transaction synchronization protocol for database centered web applications. It is optimistic during the possibly long browse phase of web clients. However, in the commit phase, the scheduler of a given DBMS is integrated. Furthermore, our synchronization protocol reduces transaction conflicts for browsing transactions, which occur often in our typical applications, i.e. market places for web learning.

1. Introduction and problem description

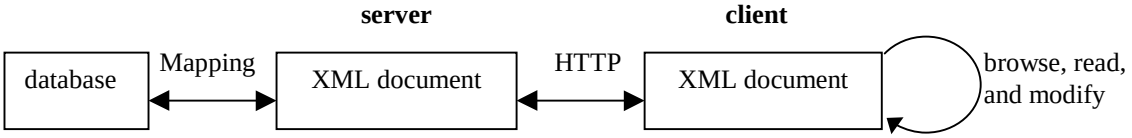
1.1. Problem origin and motivation

There is a growing interest in XML as a data exchange language for the web, even for legacy data which is already stored in a database. The integration of XML with databases with respect to storage, retrieval and update is considered a major challenge [7], [27]. Whenever XML data is modified by multiple web clients, then synchronization of the clients' access to XML data is an important topic [26],[28],[15].

Our work on transaction synchronization is motivated by the development of an application, an electronic market place for web learning, which includes searching for course material, booking courses, constructing exams and performing online tests etc.. The application requires that users work on different types of web clients that can exchange data in a portable data format (as e.g. XML) with a central server-side database, that is also accessible by legacy database applications. Furthermore, the client side of the application shall be portable, independent of the end user system, and flexible in the way information is presented. Finally, whenever multiple clients read or modify local XML copies containing the same data of the server-side database and transfer the changes of their local XML copies back to the server, we require transaction isolation for each client's transaction.

In order to meet these requirements, we developed the Axon Web Database System[1], which uses a web server that is connected to a database and communicates with web clients via HTTP using XML as a data exchange format.

From the clients' point of view, the server side of the Axon Web Database System is a so called web service [3], which allows to browse, to read and to write data that is stored in a database driven web XML server. However, on the server side, XML is mapped to a central database that can also be accessed through its other interfaces, e.g. ODBC. Both, traditional use of the database and access through the web service, have been



[1] Implemented at aXon Informationssysteme mbH in Paderborn, Germany

taken into consideration for the implemented transaction synchronization protocol.

In the market place application built on top of the Axon Web Database system, typical users browse a lot of data from their web client, find a small subset of relevant data, and eventually write small pieces of data. In order to isolate write transactions which are initiated by clients on the web, it is unnecessary overhead to lock all the data which has been accessed by the web client only for browsing purposes. Therefore, in order to increase parallelism, we suggest to distinguish between *browse access*, i.e. access to data which is not relevant to human decisions or write operations in our transaction, and *committed read access*, which is relevant to our transaction. Furthermore, when a client reads or modifies his XML copy of server-side database data, locking that database data during the whole client transaction would decrease parallelism and rise new problems when the web client looses the connection to the server. Therefore, in our protocol, the web client works optimistically on its local XML copy until it commits its operations.

1.2. Relation to other work and our focus

Recently, there has been a lot of research in the domain of storing XML documents in databases for which at least four major approaches can be distinguished. First, develop a specific XML database system, such as Rufus [25], Lore [14], Natix [16], Strudel [10] or Tamino [22]. Second, use an OODBMS to store XML or hypertext documents, e.g. [5]. Third, interpret XML documents as graph structures which are stored in a relational database, e.g. [11], [13]. Fourth, map XML documents to relational databases, e.g. [4], [6], [9], [23]. Our implementation follows the fourth approach, not only because RDBMS are widely used in practice and are fairly efficient for large amounts of data [17],[24], but also because we want to integrate legacy data from relational databases. We take a database-centric view, i.e. we consider a legacy database to be given, and use XML as a data interchange language with web clients. In contrast to other approaches which store server side copies of the transferred XML documents, our approach stores XML documents only on the client side, thereby reducing the server storage load.

Whereas a lot of recent work has been done in the field of efficient storage of XML in databases [21],[8], as well as querying such XML [2] and the development of XML schema languages [18], a new upcoming field is updating XML. For example Clock [28] and Updating XML [26] describe how to propagate changes of an XML document to a database in order to increase update performance. We follow this approach and use an update language as introduced in [26] and a DOM client in our transactions.[2]

Beyond this, the focus of our paper is on synchronization of multiple client transactions that access XML data structures which are generated from a server-side database by a flexible mapping. However, we require that the mapping is reversible, i.e. that we can map modifications of the XML structure back to modifications of the underlying database.

Concurrency control could be implemented as in distributed database systems, e.g. based on a two phase commit protocol, on top of a federation of database systems [20], or within a client/server environment. In contrast to another approach [15] that implements update services for XML document elements as two level transactions, we follow the client/server approach, because a lean synchronization protocol is sufficient for our application and avoids problems with lost connections between client and server.

Within client/server systems that ship data to be modified by clients, concurrency control is coupled with cache consistency algorithms, that are classified to *avoid* or *detect* cache inconsistencies [12] and to be *synchronous*, *asynchronous* or *deferred* [19]. Our approach uses adaptive optimistic concurrency control [1] which can be classified as *detection-based deferred*, i.e. we *detect* read access to stale data[3] instead of avoiding it, and we *defer* informing the server about the client's write operations until commit time. According to [1] and [19] this outperforms all other approaches to cache consistency in a client/server architecture like ours where the application processing is performed at the web client. Although caching XML documents on the server is compatible with our transaction protocol, this is not useful in our application which generates XML fragments from a database that is also accessed by legacy applications.

Furthermore, our transaction synchronization protocol is tailored to the specific needs of web-clients in database centered applications, i.e. it not only avoids

[2] We do not need to round-trip a whole XML Document in a transaction. A client is allowed to send only relevant, smaller fragments with the update information back to the server.

[3] Stale data is data, that has been updated and committed into the database by a different client.

problems with lost connections from web clients to a server side database, but also improves the concurrency of browse transactions. A key point of our approach is to distinguish two kinds of read operations. Reading relevant data has to be synchronized with write accesses of parallel web transactions, whereas read access to browsed data will need no synchronization, i.e. allows more parallelism of concurrent web transactions. The same optimization is extended to write operations by distinguishing two kinds of update and delete operations. Finally, our protocol is compatible with conventional (e.g. locking based) schedulers in the server side DBMS.

1.3. Requirements to web transactions working on XML documents

Transactions have to meet the following requirements, which are different from standard DBMS transaction requirements, because our transactions are initiated from web clients:

- Transactions shall provide the isolation property of a database system for read and write operations initiated by each client on its copy of an XML document.
- A rather high degree of parallelism should be reached. For example a client transaction which uses an XML fragment to modify a database tuple should run in parallel with a second transaction of a different client using another XML fragment to read or modify a *different attribute* of the *same* database *tuple*. Note, that in a conventional locking protocol the first transaction may lock the whole tuple and thus prevent the second transaction to access this tuple concurrently.
- Transactions shall support a long browse phase of the web client. During this phase, it is not acceptable to block large portions of the database for several reasons, e.g. because the connection between client and server can be lost.
- Furthermore, a simple integration of web transactions with existing DBMS transactions would be preferable, because data integration with existing databases is important.

In contrast to a distributed database system with a web interface at each node, we need only a centralized database system, and we do not require a two phase commit decision. These reduced requirements are an advantage in applications with web transactions on mobile partners, because other transactions are not blocked, when a client looses a connection to the server's database.

The next section summarizes mapping of database content to an XML document. Thereafter, we discuss client operations on XML documents, whereas the fourth section concentrates on their synchronization.

2 Mapping database content to XML

This section describes *maps* as a way to select data from a database and transform the data to an XML document which can be sent from the web server to a client, and all the way back. Although *maps* have in common with views, that maps describe only a specific part of the data in a database, the difference between maps and views is that a *map* maps between database data and a structured XML document.

For example, for a relational database containing a relation `Customer` with `customer_id` as key

Customer(**customer_id ,** customer_name ,
phone , …)

and with some tuples stored in the relation, a *map* defines, which subset of the database data is extracted and how this information is represented as XML document.[4] For example, a map can extract the customer_id and the phone number only, which could result in the following XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AxonXMLServer SYSTEM "customer.dtd">
<AxonXMLServer>
   <customer customer_id="101" phone="79797979"/>
   <customer customer_id="102" phone="27272727"/>
</AxonXMLServer>
```

The map generates the *structure* of such an XML document (i.e. its *elements* `AxonXMLServer` and `customer` and its *attribute names* `customer_id` and `phone`) according to a given DTD, but it reads actual data from the database in order to fill the *attribute values* (like `"102"`) in the XML document. When the XML document is accessed by a client, it

[4] The map itself is implemented in the Axon Web Database System as an XML document for flexibility reasons. Since these map XML documents tend to be rather long, we omit the map document here and show only the XML structures, which are extracted from the database.

makes a difference with respect to synchronization, whether a client only browses the structure of an XML document, or reads (and eventually modifies) attribute values which are mapped from database data.

Since the client may modify the received XML document and send modified XML fragments back to the server, where these modifications have to be propagated to the common server-side database, each mapping has to be reversible. We call a mapping *reversible*, if every modification of an XML fragment can be uniquely mapped back to a sequence of corresponding database operations.

A reversible mapping can be achieved in different ways, but which way is taken, is not essential to our synchronization approach. For example, the Axon Web Database System includes those database relation keys (like `customer_id`) into XML documents that are needed to identify database tuples that have to be deleted or updated when XML fragments are deleted or updated, and it uses user defined or system generated keys in the database for insert operations on XML fragments.

Both, the used mapping language and the mapping middleware used as part of our implementation, are an extended version of the open source project XMLDBMS [4]. This allows a flexible representation of data, e.g. the map can map database attributes either to XML elements or XML attributes. Furthermore, the map defines, which one-to-one, one-to-many, and many-to-many relationships between database relations are mapped to hierarchies of XML elements and how the element hierarchy looks like. Finally, the map specifies, how the XML representation distinguishes between key and non-key attributes of database relations. Among other things, the middleware allows to extract XML data from a database and to insert XML data corresponding to a given map into the database. We have extended this with update and delete operations on database data corresponding to a given XML fragment that matches a given map.

Such a mapping can be specified for users or for applications. In general a mapping maps only a subset of the accessible data (often referred to as *virtual XML Document*). Furthermore, different users generally use different maps to access the server side database, i.e. they may access different parts of the database, and the same data may be represented in different XML structures. Generally, for one database on the server there may exist several different maps. This is needed, if different types of clients need access to different subsets of the database data, or because different types of clients may need a different XML representation of the same subset of database data.

3. Client side operations on XML documents

Corresponding to the DOM, an XML document can be modeled as a tree where XML elements and attributes are represented as *nodes*. The web client can perform the following operations on the database values stored in these nodes, i.e. on any XML element or attribute of the virtual XML document defined by the map: *read committed*, *browse*, *insert*, *delete this*, *delete any*, *update this* and *update any*. For some of these operations, the client collects the accessed nodes in the transaction's *read set*, the transaction's *update set*, etc.. This seems to be a lot of work, but e.g. in our DOM implementation, the client only *marks* the accessed node as *deleted*, *inserted*, *updated* or *read*, and stores a *new value* for each insert or update operation, and stores pointers that refer to these changes. The client picks the modifications up, when the web application or its user decides to complete a transaction.

3.1 Committed read

Whenever data contained in an XML document was generated from database data (by the mapping), then the originally read database data may be relevant for further processing steps of the transaction. Therefore, this data is by default marked as *committed read* as soon *as it is shown to the client application*. This means, that a commit of the transaction requires, that a repeated read of this data at commit time of the transaction yields the same data values.

Typically only a small part of the XML data really has to be read committed, i.e. the value of that data influences decisions of the user or transaction updates. Only this subset of the XML data has to be synchronized with write operations of concurrent transactions.

Our protocol requires that the client automatically marks and collects those nodes of an XML document, the values of which have been read by the client application in the committed read mode. These nodes are added to the so called *read set* of the client transaction. Note that on commit request, the client sends the transaction's read set back to the server, where it will be checked against the actual content of the

database.

3.2. Browse access to parts of an XML document

In contrast to reading committed, *browse access* to parts of an XML document means that the original content of the database, from which the XML document was constructed, is not important for this operation.

There are two kinds of browse access. First, we say that a client application accesses data of a node *in browse mode*, if no data which originally was mapped from database data is shown to the client application (e.g. the client application *browses* only element names and attribute names which are specified in the DTD). Typical browse operations occur for example, when a tree structure is opened step by step, but without displaying internal values.

Second, a node is considered to be accessed *in browse mode*, if the client application explicitly declares the data *to be considered as* accessed *in browse mode*. The idea is, that a client application can distinguish between data which is relevant to its transaction and other data which is not relevant. Whenever a lot of displayed data is not really relevant to the user (which may often be the case in web applications[5]), then the application developer can declare the non-relevant data used in the client application to be accessed in browse mode only.

The key point is to *distinguish* between *committed read* of a node's data (for important information that was originally generated from database data) and *browsed* nodes (containing data which is either structural information or reading them is not considered to be relevant to the client application). This distinction allows us to increase parallelism, because there is no need to synchronize browse access to data with write operations of other transactions on the database. For the same reason, in the following, we also distinguish between *update this* and *update any* and between *delete this* and *delete any*.

3.3 Updates

We distinguish two kinds of update operations on nodes on the client side: *update this* updates the actual value found in the node, i.e. the value was shown before the update to the client application. This may be used e.g. to increase the given price of a product by 5 percent. In contrast, *update any* updates any information in this position without looking at the special information or special values stored here before the update. This may be used e.g. to assign a new phone number (which in general does not depend on the old phone number).

In the first case (*update this*), the information previously stored in the modified node is added to the transaction's read set. However, in the second case (*update any*), the information or value previously stored in this position is not relevant, and therefore it is not added to the transaction's read set, which reduces conflicts with concurrently writing transactions. In both cases, the new version of the modified node is added to the transaction's *update set*.[6]

3.4 Deletes

As with update operations, there are two kinds of delete operations on nodes on the client side. *Delete this* deletes the actual values found in the node, i.e. the value was shown to the client application before the delete operation was executed. In contrast to this, *delete any* deletes any information a node contains without looking at the special information or special values stored here.

In the first case (*delete this*) the information previously stored in the deleted XML fragment is added to the transaction's *read set* (that also contains the committed read fragments), whereas in the second case (*delete any*) the information or value previously stored in this fragment is not relevant, and therefore it is not added to the transaction's read set. For both delete operations, the node is collected in the *delete set* of the transaction.

Finally, we mention two minor implementation decisions regarding the deletion of XML nodes: First, we support per-row deletion on the database with the following consequences: A client that wants to delete a single XML attribute (which is not a database key) has to *update* the field to a null value, and, our *delete* operation is restricted to XML nodes containing database key-values. Second, deleting a node in a natural sense means to delete the whole sub tree. This can be done recursively by marking all children as deleted, producing overhead in time and space. Beside

[5] In our application, many users interactively browse much content, that they later on consider to be not relevant for their transaction.

[6] Our implementation even allows to change XML attribute values that contain primary keys. In that case on the database side, the mapping also has to change foreign keys in other relations that refer this modified key value.

this possibility the Axon Web Database System offers an alternative operation called *cascading delete*. In this case a cascading delete is simulated on the server side only considering key-foreign key pairs defined in the map.

3.5 Inserts

Whenever a client transaction inserts a new XML fragment into an XML document, then this new fragment is added to the transaction's *insert set* (recursively all children of the root node of the inserted fragment will be added to the *insert set*). An insert into an XML document does not modify the transaction's read set.[7] However, insert operations like all other write operations can depend on the transaction's read set. Of course, the new XML document including the inserted XML fragment must be valid according to the given DTD.

3.6 A typical operation mix

Generally, browsing an XML document takes a long time and accesses a considerably large amount of its data. A much smaller number of visited nodes will be read and considered to be relevant (although the amount may vary a lot, a typical amount is between 0% and 10% of the number of nodes which are browsed – depending on the client's role in the application). This relevant data is read committed, i.e. it will occur in the transaction's read set. In most applications, even fewer nodes are written (a typical value is between 0% and 2% of the nodes which have been browsed or read committed). That is why the focus of our transaction synchronization, i.e. to reduce the synchronization overhead for data which is only browsed, is an important improvement for browsing-intensive web applications.

4. Transaction synchronization for XML documents mapped from databases

4.1 Transaction phases

Each client's transaction is performed in two phases: a browse phase and a subsequent commit phase. Within the browse phase, the client receives XML documents from the server[8], which are filled with snapshot data from the database, and may decide to perform write operations (or committed read operations) that are collected in local write sets or local read sets.

The browse phase terminates with the web client's request to commit the transaction. This request starts the commit phase. During the commit phase, the local write sets and local read sets of the client are checked against actual values found in the database.

In the case of a successful check and commit on the server-side database, the client transaction is committed safely. Otherwise, the client gets a transaction-aborted response from the server and has various options how to proceed, e.g. abort the client transaction or retry the transaction commit phase with reading actualized stale data.

4.2 The browse phase

Within the browse phase of a transaction, the client accesses one or more XML documents, each of which is mapped on the server from a subset of the server's database data.

The client application may browse, read committed or write on its copy of the transferred XML document as described in section 3.– However the parts of the document being browsed, read and written are usually different from each other. As a side effect of each committed read or write operation on a locally stored XML document, the client transaction manager collects the actual values of those XML fragments which are read committed or written, i.e. it collects the following:

```
read set = { } ; insert set = { } ;
delete set = { } ; update set = { } ;

For each operation on the local XML document
on the client:
    Case operation of
        read committed (node) :
                read set = read set  U  { node }  ;
        insert (fragment) :
                insert set = insert set  U  { fragment } ;
        delete any (node) :
                delete set = delete set  U  { node }  ;
        delete this (node) :
                read set = read set  U  { node }  ;
                delete set = delete set  U  { node }  ;
        update any (node) :
```

[7] Key management on the server side database system is solved by our server implementation, but this is not outlined here, as it is no subject to our transaction synchronization protocol.

[8] In general subsets of the virtual XML document described by the map.

```
                update set = update set  U  { node } ;
        update this (node) :
                read set = read set  U  { node } ;
                update set = update set  U  { node } ;
    end Case ;
end For ;
```

At the end of the browse phase, the client submits a commitRequest call to the server and passes read set and write sets as parameters:

```
serverResponse =
commitRequest( read set , insert set ,
                    delete set , update set ) ;
```

If the server returns *transaction committed*, then the web client transaction is committed too. The server-side action for commitRequest is as follows.

4.3. Commit phase (and server side transaction protocol)

The server implements commitRequest as a server side conventional database transaction. It gets with a call of commitRequest all relevant data that has been read or written by the client. The map is used to map the read or written XML fragments to database operations. For the nodes of the read set, it is checked whether or not they are mapped to the same values as the actual values found in the database. If not, the server-side conventional transaction is aborted. Otherwise, for the values of the write sets, the changes are propagated to the database. In case of server side transaction failure, e.g. caused by an integrity constraint violation or a deadlock, the server aborts this conventional database transaction, otherwise it commits the transaction. Finally, the server signals either commit or abort to the client.

```
CommitRequest ( read set , insert set , delete set ,
                update set )
{
   tBegin;
   if ( hasChanged( read set ) )  tAbort ;
   else
   { use the map and perform database updates
     ( insert set , delete set , update set ) ;
     if ( integrity violation or other failure )  tAbort ;
     else  tCommit ;
   }
   tell client about the transaction-commit-status ;
}
```

In case of a server side commit, the client web transaction can also be committed.[9]

In most other schedulers for relational database systems which lock (or validate) the access to complete tuples, two operations on the same database tuple are in conflict, if at least one of them is a write operation. In our approach however, during the (rather long) browse phase, only a subset of these pairs of operations are in conflict, i.e. only those pairs of operations where the read or written XML fragments are mapped from *the same attribute* values *of the same tuples*. Therefore, and because we distinguish between browsed and read committed read information, we can expect a higher degree of parallelism compared to a traditional synchronization approach.[10]

4.4. Possible client reactions on failure

In case of a server side abort, the client has three different options: First, the client may decide to withdraw its requested changes, i.e. to abort and quit. Second, the client may decide to actualize its read set, i.e. refresh the read data, and retry a commit request, using these actualized data. Finally, e.g. in case of multiple failures, a client may decide to reduce its read set in order to increase its chance to commit. This can be done by declaring more input information as browse information, i.e. not to be relevant for the intended update transaction.

5. Summary and Conclusions

We presented a synchronization protocol for transactions on web clients that communicate via XML data exchange with a server, where the XML documents are mapped to a server-side database. In contrast to pure locking schedulers, our web client works optimistically in the browse phase until it switches to the commit phase. In the commit phase, transaction validation and commit are performed on the server by the given database system's scheduler, which might use locking. Doing part of the synchronization at the web client's side, reduces the server's load for cache management

[9] On demand of the client application, XML fragments may now be refreshed with actual database data and sent again from the server to the client.

[10] Since in the commit phase, the read committed part of the data is checked against the actual server database data, it is even safe to access dirty data in the browse phase. Therefore, during the browse phase, the server's database could be accessed via the map without synchronization.

and synchronization. Our transaction synchronization protocol offers a high degree of parallelism for long web transactions, if clients access a high percentage of the XML nodes in browse mode, i.e. the access to the values stored in these nodes must not be committed. This is very common in our application and occurs e.g. when the client application browses element names or attribute names or declares some attribute values as browse data.

Furthermore, the synchronization strategy does not limit us to use a single specific kind of XML database mapping. Instead the mapping function from databases to XML can be chosen free, as long as it is reversible. Even more, it seems to be a promising next step to adapt our web service's communication protocol to SOAP.

References:

[1] Adya, A., Gruber, R., Liskov, B., Maheshwari, U.: Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks, ACM SIGMOD Int. Conf. on Management of Data, 1995

[2] Bonifati, A., Ceri, S.: Comparative Analysis of Five XML Query Languages. SIGMOD Record 29(1): 68-79 (2000)

[3] Bosworth, A.: Developing Web Service, ICDE, 2001.

[4] 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 Web-based Information Systems (WECWIS), San Jose, California, June, 2000

[5] Böhm, K., Aberer, K.: HyperStorM - Administering Structured Documents Using Object-Oriented Database Technology. Proc. of the ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada, June 1996

[6] Carey, M., Florescu, D., Ives, Z., Lu, Y., Shanmugasundaram, J., Shekita, E., Subramanian, S.: XPERANTO: Publishing Object-Relational Data as XML. Int. Workshop on the Web and Databases (WebDB), Dallas, May, 2000

[7] Ceri, S., Fraternali, P., Paraboschi, S.: XML: Current Developments and Future Challenges for the Database Community. Proc. of the 7th Int. Conf. on Extending Database Technology (EDBT), Springer, LNCS 1777, Konstanz, March, 2000

[8] Deutsch, A., Fernandez, M., Suciu, D.: Storing Semistructured Data in Relations. Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Jan., 1999

[9] Fernandez, M., Tan, W-C., Suciu, D.: SilkRoute: Trading between Relations and XML. 9th Int. World Wide Web Conf. (WWW), Amsterdam, May, 2000

[10] Florescu, D., Levy, A., Mendelzon, A.: Database Techniques for the World Wide Web: A Survey. ACM SIGMOD Record, Vol. 27, No. 3, September, 1998

[11] Florescu, D., Kossmann, D.: Storing and Querying XML Data Using an RDBMS. IEEE Data Engineering Bulletin, Special Issue on XML, Vol. 22, No. 3, September, 1999

[12] Franklin, M., Zdonik, S.: A Framework for Scalable Dissemination-Based Systems, OOPSLA, 1999

[13] Gardarin, G., Sha, F., Dang-Ngoc, T.-T.: XML-based Components for Federating Multiple Heterogeneous Data Sources. Proc. of the 18 th Int. Conf. on Conceptual Modeling (ER), Paris, Nov., 1999

[14] Goldman, R., McHugh, J., Widom, J.: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. Proc. of the 2nd Int. Workshop on the Web and Databases (WebDB), Philadelphia, June, 1999

[15] Grabs, T., Böhm, K., Schek, H.J.: Scalable Distributed Query and Update Service Implementations for XML Document Elements, RIDE-DM 2001

[16] Kanne, C.-C., Moerkotte, G.: Efficient Storage of XML Data. Proc. Of the 16 th Int. Conf. On Data Engineering (ICDE), San Diego, March, 2000

[17] Kappel, G., Kapsammer, E., Rausch-Schott, S., Retschitzegger, W.: X-Ray - Towards Integrating XML and Relational Database Systems, ER 2000

[18] Lee, D., Chu, W.W.: Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema, ER 2000

[19] Öszu, M.T., Valduriez, P.: Distributed Database Systems, 2nd Ed., Prentice Hall, 1999

[20] Schenkel, R., Weikum, G.: Taming the Tiger: How to Cope with Real Database Products in Transactional Federations for Internet Applications. GI-Workshop Internet-Datenbanken 2000

[21] Schmidt, A., Kersten, M.L., Windhouwer, M., Waas, F.: Efficient Relational Storage and Retrieval of XML Documents, WebDB (Informal Proceedings), 2000

[22] Schöning, H., Wäsch, J.: Tamino -- An Internet Database System. Proc. of the 7 th Int. Conf. on Extending Database Technology (EDBT), Springer, LNCS 1777, Konstanz, March, 2000

[23] Shanmugasundaram, J., et al.: Relational Databases for Querying XML Documents: Limitations and Opportunities. Proc. of the 25 th Int. Conf. On Very Large Data Bases (VLDB), Edinburgh, 1999

[24] Shanmugasundaram, J. et al.: Efficiently Publishing Relational Data as XML Documents. VLDB 2000

[25] Shoens, K., et al.: The Rufus system: Information organization for semi-structured data. Proc. of the Int. Conf. On Very Large Data Bases (VLDB), Dublin, Ireland, 1993

[26] Tatarinov, I., Ives, Z.G., Halevy, A.Y., Weld, D.S.: Updating XML, ACM SIGMOD Int. Conf. on Management of Data, 2001

[27] Widom, J.: Data Management for XML - Research Directions. IEEE Data Engineering Bulletin, Special Issue on XML, Vol. 22, No. 3, September, 1999

[28] Zhang, X., Mitchell, G., Lee, W.C., Rundensteiner, E.A.: Clock: Synchronizing Internal Relational Storage with External XML Documents, RIDE-DM 2001.