XML Fragment Caching for Small Mobile Internet Devices

Stefan Böttcher, Adelhard Türling

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

Abstract. Whenever small mobile devices with low bandwidth connections to a server execute transactions on common XML data, then efficient data exchange between client and server is a key problem to be solved. However, a solution should also consider client- side cache management for the XML data, synchronization of concurrent access to the XML data, and lost connections during transaction execution. In order to reduce data exchange between client and server, our protocol reuses data stored in the client's memory instead of reloading data into the client's memory wherever possible. A key idea is that the server keeps a 'living copy' of the XML fragments in the client's memory for efficient cache management. Furthermore, our protocol integrates well with a validation based scheduler in such a way that offline work and transaction completion after lost connections are supported. Finally, we present some optimizations that further reduce client-server communication.

1. Introduction

1.1. Problem origin and motivation

There is a growing interest in XML as a data representation language and data exchange format for different web based enterprise applications. Whenever complex applications include small mobile devices in distributed transactions, then resources like communication bandwidth and available memory are known to be bottlenecks that make mobile transactions slow or even impossible.
Our work is motivated by a web-database application [2], that now shall be ported to small mobile devices, which communicate via low bandwidth connections with a server. Within our application, several client transactions read or modify (their local copy of) a virtual XML document, which is stored on a server.
Our approach optimizes the data communication between client and server needed to support client transactions, i.e. to keep the client's copy of the server's XML document up to date.

1.2. Relation to other work and our focus

Our work is related to contributions in different fields ranging from XML databases, over cache management in distributed databases and transaction management for mobile devices, and focuses on optimization of client-server communication over low bandwidth connections.
Recently there has been much research in the area of XML databases, e.g. [8,17], with an increasing interest in query optimization for XML data (e.g. [13]) and efficient storage of XML data (e.g. [11]). While other contributions on cache management and query optimization assume, that there is a fixed and fast connection from the main memory used by the client application to the XML database, we focus on special optimizations needed for

small bandwidth connections between a client application and a server side XML document.
Additionally, many contributions have proposed ideas for cache management in distributed DBMS, e.g. [4,7,12]. While these contributions integrate the cache in the query processing environment by caching the results of queries, other contributions improve cache management by the storage of deltas, e.g. for OODBMS [6] and for XML [14]. Common to the last approach, we compute deltas, however, we compute delta XML fragments on the server side and our focus is to restrict the communication between server and client to the exchange of delta XML fragments.
Furthermore, mobile transactions are considered to be an additional challenge due to limited bandwidth and frequent disconnection [16,21]. Compatible to related contributions on mobile transactions (e.g. [3,5]), we assume an optimistic approach to transaction synchronization. However additionally, we integrate optimized cache management with validation.
Finally, while there has been much excellent work on scaling up XML database technology to large databases, our work can be characterized as downsizing database technology, which is considered to be very important but quite difficult [1,9]. But different from other work that downsizes database technology, e.g. [10,15,18], we ”downsize” the exchange of XML fragments which leads to different requirements.
Different from all other contributions, we focus on cache management and reduced client-server data exchange tailored to the specific needs of XML databases which are accessed from mobile clients via small bandwidth connections.

2. Underlying concepts and problem description

We offer two possible ways for clients in our transaction approach to access fragments of an XML document stored on a server: *direct requests* use XPath [20] expressions to access XML fragments, whereas *indirect requests* are transparently initiated by calling DOM (document object model) [19] methods of our client API. We summarize the underlying techniques (XPath, DOM) used for these requests, before we describe the problem.

2.1 Direct requests use XPath

As mentioned, XPath, the query language of XML, is used to describe which fragment of the server-side document shall be transferred to the client’s memory. A read operation, e.g.:

```
doc.read( ”/BookingService/Connections/Connect[ @id=’3’ ]//* ” )
```

has as argument an XPath expression that describes a set of nodes of the underlying XML document. The path ‘/BookingService/Connections/Connect’ starts at the root ‘/’ of the XML document and uses the child axe, i.e. it identifies ‘Connect’ elements under ‘Connections’ elements under a ‘BookingService’ element. The predicate filter [@id=’3’] restricts the set of selected ‘Connect’ nodes to the node with the attribute ‘id’ set to the value of ‘3’. Finally, ‘//*’ expresses that all descendent nodes of the node(s) selected by ‘/BookingService/Connections/Connect[@id=’3’]’ are included in the request.
XPath expressions are used to communicate, log, and analyze not only the client’s read operations, but also for the client’s write operations (e.g. in the context of transaction validation), i.e. all XML fragments read or written by clients are described by XPath expressions on the server side.

2.2 Indirect access through the DOM interface

In addition to direct requests based on XPath, we offer the access through a mobile DOM interface ($MDOM_{tr}$). This interface supports the main methods of the DOM [19] like reading or writing node values, appending or deleting nodes, and navigating on the document object model using inter-child and parent-child relations. Because we focus on data exchange, we report and set node names as defined[1] for the document and differ between node-types of the DOM (like attribute, element, or comment), but leave the document validation to the server.

2.3 Problem description

Within our application scenario, several clients share access to a *virtual XML document*, i.e., the client's view on the data source is the complete XML document no matter what is the real representation on the server - and ignoring that the client only works on parts (fragments) of the XML document and concurrently with other clients. The virtual XML document is stored or made available on a server and XML fragments are transferred to clients, where they are read or modified. Clients are expected to use a DOM API for their access to the server side XML document and to use XPath expressions to preselect or focus an interconnected fragment of the XML document they want to access in a transactional context.

Our clients face two partially contrary main problems: First, since these clients use low bandwidth connections to a server, data exchange shall be reduced to a minimum. Second, efficient use of the client's memory is a key requirement, because the client's data memory may be too small for the data needed for client applications.

More specific, we expect that most but not all of the client operations access rather small XML fragments that fit into the client's memory. We further expect that a considerable part of a single client's transaction use the same (personal) subset of the XML data again and again. For these transactions, it makes sense, to reuse data already stored in the client's memory wherever possible, even across the boundaries of a single transaction. Therefore, our focus is to optimize memory management of DOM applications running on small internet devices not only beyond the boundaries of a single operation, but also beyond the boundaries of a single transaction.

Since clients are mobile, they may loose their connection to the server, but transactions should get the opportunity to complete after the connection is reestablished in order to be as smart on resources as possible. Furthermore, each transaction involves only a single client (and the server) and each client runs at most one transaction at a time. While synchronization of such concurrent transactions can be done on the server as described e.g. in [3], here we discuss, the transaction's benefit of reused data from previous transactions left in the client's memory.

[1] E.g. defined by a server side DTD or XML schema

3. Overview and key concepts

In this section, we outline $MDOM_{tr}$ a mobile client's DOM interface which allows the client to interact with a server side XML document concurrently with other clients. Our approach allows the client to operate on arbitrary parts of a DOM of the server's XML document, while data loading form the server to the client and transaction management are performed by our protocol transparently to the client application. The presented protocol uses available client memory to cache data and optimizes data exchange between client and server. For this purpose, our approach extends the server-side XML database system with server-side memory areas for each client, where the server keeps an identical physical copy of each client's data memory and logical representations of each client's transactional operations. In Section 4, we discuss the displacement strategy used on both, the client's memory and the server's copy, and the achievable communication reduction. Section 5 presents the transaction management adapted to the needs of our mobile clients. We end up with Summary and Conclusions in chapter 6.

3.1. The data structure for $MDOM_{tr}$

As mentioned in Section 2.2, we implement our approach as a small mobile-client DOM API ($MDOM_{tr}$) that accesses a server side XML document in a transactional context. We reserve a fixed amount of the client's memory (called the *client's cache*) at application initiation time and use it for the storage of the XML fragment the client is currently working on. The cache size usually differs from client to client (it may be e.g. 256 KB) and often represents the client's maximum available (data-) memory, i.e. that part of the client's memory that is not needed for application program code, etc..

The $MDOM_{tr}$ data structure consists of a pointer connected tree[2] that represents the XML fragment of accessed data and associated inter-connected access time stamps which are used for displacement. The cache is organized in memory blocks with a fixed precalculated[3] size. Such a memory block is able to hold any expected node making it easy to organize and displace it. Possible overflows of memory blocks caused by arbitrary input values are handled by using additional memory blocks. If no such block is available, other nodes have to be displaced as described in chapter 4. When a client accesses the DOM tree, it is transparent to the client application, whether data is already available in the client's cache or it has to be downloaded from the server when a node-object is accessed. The API achieves this transparency by using additional pointers to indicate whether siblings or children of a node are missing and have to be downloaded or are already available in the cache.

3.2 Our client-server communication protocol

Our protocol supports two main strategies for clients to access data. As introduced for read operations in chapter 2, we call these strategies *direct* and *indirect*. Whereas an indirect request initiated through the MDOM is possible for all operations (read, update, insert, delete) on the virtual XML document, direct requests are only supported for read and delete operations.
Direct requests use XPath expressions to identify the fragment the operation accesses. For example, a direct read access of the client transfers an XPath expression to the server,

2 Within our implementation, each node object is represented by 10 pointers, the data value and a time stamp. When each pointer, time stamp and decision criteria consumes 4 byte and each date value consumes 32 byte, we need 80 bytes/node, i.e. we can store 3200 nodes in 256 KB cache.

3 E.g. a memory block should be able to hold 90% of all existing nodes in the document

where the corresponding interconnected XML fragment is extracted from the virtual XML document. Only those parts of the fragment that are not yet stored in the client's cache are transferred to the client in order to save communication bandwidth. When the data is stored in the client's cache, the client can use $MDOM_{tr}$ operations in order to navigate within the retrieved fragments or to initiate any indirect request. - For the delete operation, it is sufficient to send the XPath expression to the server, because the XML fragment to be deleted can be extracted from the server's cache copy. Both, read and delete operations have in common that they are often applied to fragments and that the fragment itself is the only parameter that can be represented as a short (XPath) expression. This makes a direct request the best choice for fragment based read and delete operations.
On the other hand, indirect requests are used by iterated read, insert, update or delete operations on single nodes. Here, the strategy is to only transfer the physical data that have been changed back to the server. In the case of individual insert and update operations, transferring the new values for inserted or updated data is an optimum w.r.t. the amount of data that must be transferred[4], because the new data has to be exchanged anyway.
The server's response to a client's indirect write request or the direct delete request is a simple acknowledgement to synchronize the protocol if the request was well formed and no constraints where violated. Otherwise, the server refuses the write operation and repairs inconsistent data.
The protocol can follow different strategies (heuristics) to respond to indirect read or write requests. One strategy, optimizing the low bandwidth bottle-neck, is that the server returns each accessed node immediately and the client sends written data for each operation immediately. Other strategies allow the server or the client to send fragments containing multiple nodes for read operations at once or data of multiple write operations at once. Notice that strategy may depend on different optimization goals like: *small bandwidth* or *minimal online time.*

3.3 The server's copy of the client's cache

One of our key ideas is to hold an identical physical copy of the client's cache on the server (called the *server's cache copy*) and to duplicate the client's operations on it. This is possible because any operation of the client changing the cache is communicated from the client to the server and because we use an identical prearranged displacement strategy[5] and an identical fixed prearranged cache size for both, the client's cache and the server's cache copy. Therefore, the server's *cache copy* is able to displace data exact the same way the client cache does without any additional client-server interaction. Because keeping the server's cache copy up do date doesn't involve any additional communication, its costs are only server memory and CPU time. Since the client's memory is rather small, the server's load on memory resources and CPU time to keep track of the client's operations is acceptable in our application[6]. As we'll see in chapter 4 the server uses its *cache copy* to optimize the data communication with the client.

[4] We focus on single data updates here and do not discuss mass updates that could be done by stored procedures.
[5] e.g. LRU or FIFO, limitations are discussed later.
[6] e.g. this would be one displacement-step per client interaction and 256 Mbytes of memory for 1000 clients

4. Efficient memory management

Small bandwidth for the communication between client and server requires to avoid unnecessary exchange of large XML fragments or to exchange the same data repeatedly. Instead, reduced data exchange containing only client requests and cache modifications (on client or server) is preferred. Therefore, still available client side data belonging to older and actual transactions should be reused whenever possible.

4.1 Flexible displacement strategy

As mentioned, the basic idea in order to reduce the data exchange between client and server is, that client and server have identical copies of the client's cache and use local, identical, and independent displacement strategies for their copy. Note that we do not require the client and the server to use a specific displacement strategy (e.g. LRU or FIFO), but we only require that they use an identical strategy during a transaction. In fact, the client can even choose the displacement strategy from a set of predefined displacement strategies – depending on the application.
In principle, a displacement strategy has to obey the following rules:

1. Nodes not accessed in the actual transaction have to be displaced before nodes accessed in the actual transaction.
2. Only displace as much nodes as needed.
3. Displacement always starts at leaf nodes, i.e. it is not allowed to displace an inner node as long as its successor nodes are not displaced.
4. The displacement strategy has to obey locks.

4.2 Locks and override rules

Additionally, client and server are allowed to lock and unlock fragments of nodes in the cache. Such nodes are not allowed to be displaced. This override rule is used by the client-side $MDOM_{tr}$ in order to prevent misleading pointers. Since the client side MDOM decides locally to lock (or unlock) a node, it informs the server of its new locks (or unlocks) together with the next client request. Similarly in the *delta refresh scenario* (as described in section 4.3), the server informs the client that some nodes are locked, i.e. can not be displaced.
Be aware that although locks have to be communicated and locks decrease the cache available for displacement, locks can be used to enhance the offline workability of a client, because the application can be sure that the needed data is not displaced and no further read request to the server is needed for this fragment. As we will see in the *delta refresh scenario* described in section 4.3, a temporary lock initiated by the server is needed for a proper working protocol.

4.3 The benefit of caching former operations' data

The goal of caching former operations' data is to reuse the data instead of reloading it wherever possible. First, read operations are performed locally (we will call this *zero refresh scenario*) whenever the client knows, that it has already read all the required data in the same transaction. If not, the client contacts the server.
Whenever a read or insert request is initiated by a mobile client, the limited capacity of the cache may require data replacement which leads to two additional scenarios which we call the delta refresh and the overflow scenario. Whereas update requests only affect time stamps and data values, they do not displace any nodes but should increase the chance of

the updated nodes to stay in the cache. A delete request of the client only releases memory that can be used by data of the next operation.

Zero refresh scenario

As mentioned before, the client cache associates access time stamps with each stored node of the XML fragment, such that the client can compute whether the data has been loaded in the current transaction or in a previous transaction. In order to avoid unnecessary read requests for fragments that are completely in the client's cache, the client API first searches data in its local cache. If the needed fragment is completely in the client's cache and has a time stamp that belongs to the current transaction, then no read request is submitted to the server[7]. Note however, if the data found in the client's cache has a time stamp that belongs to an older transaction or the client does not contain all data it needs in its cache, then the client asks the server for actual data.

Delta refresh scenario

What we expect to occur most often in this protocol is the *delta refresh scenario*. This scenario takes place whenever a client requests to read a fragment, a part of which is already stored in the client's cache (called *existing fragment*). Then the server only has to send the missing data (called the *delta fragment*).

If the server now would send the *delta fragment* without any additional information, then the displacement strategy used for server and client might displace some nodes of the *existing fragment* and the client's processing might fail. To avoid this, the server applies the *delta fragment* first to the server's *cache set,* monitoring if any node of the *existing fragment* is effected. If an *existing fragment* node would be displaced, then the server overrides this by a lock on this node. All locks are collected and assembled to a temporary lock list representing as a lock-fragment. This temporary lock fragment is sent to the client in the *delta update* response. The client now can follow the server's steps by first locking the same nodes and thereafter applying the *delta fragment* to the cache. After that, the client automatically unlocks the temporary locked fragment (as the server does after sending the temporary lock fragment) and client and server have the same state in their caches again.

Note that the delta refresh scenario not only covers the situation that a previous query of the same transaction left actual data in the client's cache, but also that data from a query of a previous transaction left valid data in the client's cache.

If the client requests to read a fragment of the XML document and none of the fragment's nodes is available in the clients cache, then obviously the complete fragment has to be transferred. Note that only in this special case the cache is no advantage or disadvantage w.r.t. the attempted reduction of client-server communication.

4.4 Treatment of large results (the overflow scenario)

Whenever the result to a single XPath query of the client will not fit into the client's cache, there are several alternative ways to iterate over the query result.

A first approach is, that the server follows heuristics to fill the client's cache with data of the queried XML fragment that is expected to be used by the client's application first. If the client application wants to access data of the requested fragment that is not available in the cache, then the $MDOM_{tr}$ API transparently loads it on demand (indirect load request). Such a heuristics can be driven by parameters provided by the client, e.g. a preferred access sequence (e.g. depth first traversal) and a preferred fragment size to be transferred at once. This allows an application to *implicitly* iterate over a fragment of any size without implementing any additional code that takes care of such a situation. The price for this advantage is that reload and displacement volume depends on whether or not the displacement and overflow heuristics are appropriate for the given application.

[7] A list of updated time stamps is sent to the server with the next communication step.

In the special case where it is known at design time of the client application which amount of data a single client operation will access, a second approach is to implement an *explicit* iterator. Explicit iterators are stored at the server side and optimize data access for this query, such that the client application only has to pass a reference to the appropriate iterator with the query. Notice that the protocol is still able to fall back to indirect iteration (on each iteration fragment), if the client's cache size is too small for a whole explicitly defined iteration fragment. If more than one fragment has to be iterated, the available space in the client's cache is computed and the server sends as many fragments as the client can hold.
After such a large iteration has been processed, the client cache will be used by the protocol and chosen replacement techniques as before. Since typical read operations in our applications currently use an XML fragment that is considerably smaller than the client's cache, we consider the need for the management of large results to occur not so often.

4.5 Advantages of our memory management

The main goal of our approach is to allow for transactions on small mobile devices to access a server-side large XML document over a low bandwidth connection. Our approach reduces (the costs for) client-server communication in several ways. First, read operations are performed only locally (zero refresh scenario) whenever the client knows that it has read all the required data before and within the same transaction. Second, the delta update scenario reduces the size of returned fragments. Third, the ability of the client to use an XPath expression and to retrieve a larger XML fragment at once reduces the number of further client requests. Forth, the reuse of old transaction's data still kept in the client's cache instead of reloading the data is an additional advantage. Finally, since in the *zero refresh scenario* the protocol needs no server interaction, the client has the opportunity to work offline for a long time.

5. Transaction management

5.1. Integration of our approach with validation

As stated before, each client access to the XML document must be done in the context of a transaction. Like most transaction synchronization protocols for mobile clients [5,16], we use an optimistic approach to transaction synchronization on the server-side, because it is not acceptable to block XML fragments for a mobile client that may loose the connection to the server. Since our approach to optimistic transaction synchronization is described in [3], we do not outline it here, but instead mention the following advantages.
Our approach to validation needs no extra communication between the client and the server - except that the client signals that it wants to commit a transaction, and the server informs the client about the commit status of this transaction. The reason is that for direct read or delete operations, our validation protocol [3] uses exactly the same XPath expression as communicated from the client to the server, i.e. validation applies them to modified XML fragments of older concurrent transactions. Furthermore, indirect read or write access can be translated on the server side to the corresponding XPath expression, i.e. this also needs no extra communication between client and server for the purpose of validation.
Note that this communication optimization for the purpose of validation is compatible with the reuse of data that was loaded by old transactions into the client's cache for the following reason. Before old data in the client's cache can be reused for a new XPath query, the client asks the server whether or not the data is still up to date (delta refresh scenario), i.e. submits the XPath expression to the server. This XPath expression is used in the validation phase, i.e. validation has to consider only the actual transaction's XPath

expressions (even if data exchange is reduced to those XML fragments that have changed from older transactions to the actual transaction).

5.2. Offline work and lost connections

Whenever the client uses an XPath expression that covers all the data it needs in a transaction and the corresponding XML fragment fits into the client's cache, then the client needs no further communication with the server (zero refresh scenario) until the end of the transaction, i.e. it could work offline. Furthermore, when the client-server connection gets lost during transaction execution, the client can continue transaction execution as long as it has all needed data in its local cache. Thereafter, it has to wait until the connection is reestablished, and can continue its transaction. Finally, if the connection gets lost during a data exchange step between client and server, client and server can recover by repeating the data exchange operation during which the connection got lost.

5.3. Further advantages of our approach after a failing validation

If validation fails, the server sends an abort response to the client. After such an abort, the client can decide to redo the whole transaction. Especially for the frequent case of personal transactions that work only on a small (individually different) fragment of the XML document that fits in the client's cache, we expect that most of the client's cache will contain data that is still up to date. In such a case, submitting the same query as in the previous transaction will require that only a small part of the client's cache has to be refreshed by the server, i.e. again we can reduce the data transfer between client and server. The same advantage holds, if a transaction has to be restarted, because it lost a connection.

6. Summary and Conclusions

A key requirement for transactions on small mobile devices with a low bandwidth server connection is to reduce (the costs for) data exchange and communication between client and server. Our approach contributes to this goal in two ways, i.e. by reusing cache data instead of reloading cache data, and by reducing the communication steps needed in order to check whether client and server cache are up to data.
In order to reuse the client's cache data, we keep a server-side cache copy, that is used to compute those (smaller) delta fragments that are needed to update the client's cache. We transfer only the delta fragments and avoid to reload existing fragments of the client's cache.

In order to reduce communication steps between client and server, we have contributed the following ideas. First, using XPath expressions to read (or to delete) a larger fragment at once needs fewer data request and data transfer steps between client and server. Second, we use an identical replacement strategy in the client's and the server's cache that avoids extra communication steps to synchronize both caches, i.e. all information needed to synchronize the caches can be taken from the read or write operations that the client performs on the server-side XML document. Third, the zero refresh scenario avoids a server call, whenever the client finds all the required data in its cache and knows that the data has been read from the server (or written by a client operation) within the same transaction. Forth, transaction synchronization (in our case implemented by validation) needs no extra communication between client and server (except that the client asks the server to complete the current transaction and that the server responds with the commit status). Fifth, even in the case of a failed validation, our approach may profit from the data of the failed transaction that is kept in the client's cache, whenever most of this data is still up to date and only a small delta refresh fragment is needed. Finally, our approach integrates well with lost connections.

We considered the presented approach to be a significant step towards new application areas which use XML on small mobile devices. Further research on optimized heuristics for both, replacement and prefetching of XML fragments, with a focus either on small bandwidth connections or on minimized connect times, may be a challenging direction to extend our approach. We are currently implementing a prototype to prove our protocol and to investigate the efficiency of different strategies for different application behavior.

References:

[1] Bobineau, C., Bouganim, L., Pucheral, P., Valduriez, P.: PicoDBMS: Scaling down Database Techniques for the Smartcard. Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000.

[2] Böttcher, S., Türling, A.: Transaction Synchronization for XML Data in Client-Server Web Applications. Informatik 2001, Jahrestagung der GI, Wien, 2001.

[3] Böttcher, S., Türling, A.: Optimized XML Data Exchange for Mobile Concurrent Transactions. Workshop MDBIS, Jahrestagung der GI, Dortmund, 2002.

[4] Dar, S., Franklin, M., Jonsson, B., Srivastava, D., Tan, M.: Semantic data caching and replacement. In Proc. 22nd VLDB, Bombay, 1996.

[5] Ding Z., Meng, X., Wang, S.: O2PC-MT: A Novel Optimistic Two-Phase Commit Protocol for Mobile Transactions. DEXA 2001: 846-856

[6] Doherty, M., Hull, R., Rupawalla, M.: Structures for manipulating proposed updates in object-oriented databases. In SIGMOD 1996.

[7] Franklin, M., Jonsson, B., Kossmann, D.: Performance tradeoffs for client-server query processing. In *Proceedings of the ACM-SIGMOD Conference on Management of Data* (Montreal, Que., June). ACM, New York, NY, 1996.

[8] 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

[9] Graefe, G.: The New Database Imperatives. *Int. Conf. on Data Engineering (ICDE)*, 1998.

[10] IBM Corporation. *DB2 Everywhere – Administration and Application Programming Guide*. IBM Software Documentation, SC26-9675-00, 1999.

[11] 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

[12] Kossmann, D., Franklin, M.J., Drasch, G.: Cache Investment: Integrating Query Optimization and Distributed Data Placement. ACM ToDS, Vol. 25, No. 4, Dec. 2000.

[13] Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Expressions. Proc. of the 27th VDLB, Roma, 2001.

[14] Marian, A., Abiteboul, S., Mignet, L.: Change-Centric Management of Versions in an XML Warehouse. Proc. of the 27th VDLB, Roma, 2001.

[15] Oracle Corporation. *Oracle 8i Lite - Oracle Lite SQL Reference*. Oracle Documentation, A73270-01, 1999.

[16] Rasheed, A., Zaslavsky, A.B.: A Transaction Model to Support Disconnected Operations in a Mobile Computing Environment. OOIS 1997: 120-130

[17] 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

[18] Sybase Inc. *Sybase Adaptive Server Anywhere Reference*. CT75KNA, 1999.

[19] W3C: Document Object Model (DOM) Level 1 Specification. http://www.w3c.org/TR/1998/REC-DOM-Level-1-19981001/

[20] W3C: XML Path Language (XPath) Version 1.0 . http://www.w3.org/TR/xpath

[21] Yeo, L.H., Zaslavsky, A.B.: Submission of Transactions from Mobile Workstations in a Cooperative Multidatabase Processing Environment. ICDCS 1994: 372-379