Fine-Grained Management of Natively Stored XML Documents

Michael Haustein Theo Härder

University of Kaiserslautern
P. O. Box 3049
D-67653 Kaiserslautern, Germany
{haustein,haerder}@informatik.uni-kl.de

Abstract

Processing XML data in relational database systems (RDBMSs) requires a sophisticated and application-specific mapping of XML data to relational database tables and columns. Future database systems should provide—in addition to the relational processing capabilities—native storage and processing mechanisms for XML data which allow for abstraction and isolation from the internal representation at higher system layers.

In this paper, we present the architecture and implementation of our prototype system which gives the user full relational and XML functionality based on native storage structures and processing capabilities for each side. Our XML Transaction Coordinator (XTC) supports such a combined query processing via declarative and navigational interfaces. For this purpose, the XTC architecture extends an existing (object-) relational database system by a native XML storage engine using mature techniques proven and tested in various DBMS implementations.

1 Introduction

The use of XML for electronic data interchange often leads to an enormous number and size of files keeping semi-structured XML data. Special application domains (e. g., mapping UML diagrams to XML structures or maintaining large tagged documents) lead to management of voluminous XML data in centralized repositories. Collaborative workflows in these application domains require concurrent read as well as write access to such XML data.

Currently available relational or object-relational database management systems ((O)RDBMSs) only manage structured data well. There is no effective and straightforward way for handling XML data. This is obviously true when simple CLOB types have to be used. In particular, searching of XML documents becomes prohibitively slow. But also more refined mappings do not lead to good solutions per se: An innumerable number of algorithms [11, 17, 24] has been proposed for the mapping of semi-structured XML data to structured relational database tables and columns (the so-called "shredding"). Especially the isolation of concurrent transactions in RDBMSs is tailored to the relational data model and does not take the semi-structured data model and corresponding interfaces of XML into account. "Shredded" mappings of XML documents to relational tables may cause disastrous locking behavior, in particular, if relational systems lock entire pages or even entire tables in order to prevent the phantom anomaly.

On the other hand, native XML database systems enable tailored processing of XML documents, but most of the systems published in the DB literature are designed for efficient document retrieval and not for frequently concurrent and transaction-safe document modifications. This primarily results from the numbering schemes used to identify XML components. These schemes allow for very fast computation of structural dependencies, but modifications of the document structure often lead to re-numeration of large document parts. As opposed to these retrieval-oriented systems, Natix is a rare example that is designed to support concurrent modifications [10].

In any case, there are no specific provisions to process concurrent transactions guaranteeing the ACID properties and using typical XML document processing (XDP) interfaces like SAX [4], DOM [17], and XQuery [3] simultaneously. For this reason, we implemented our prototype system, called *XML Transaction Coordinator (XTC),* with native XML storage structures as a testbed for the evaluation of XML storage and searching efficiency and, later on, synchronization and logging/recovery algorithms. The detailed architecture of our system is presented in this paper. Our *XTCserver* is a database management system which supports combined processing capabilities for relational and XML data and is strictly designed in accordance to the well-known five-layer database architecture [14]. In this way, the XTCserver implementation also illustrates the idea of extending existing (object-) relational database sys-

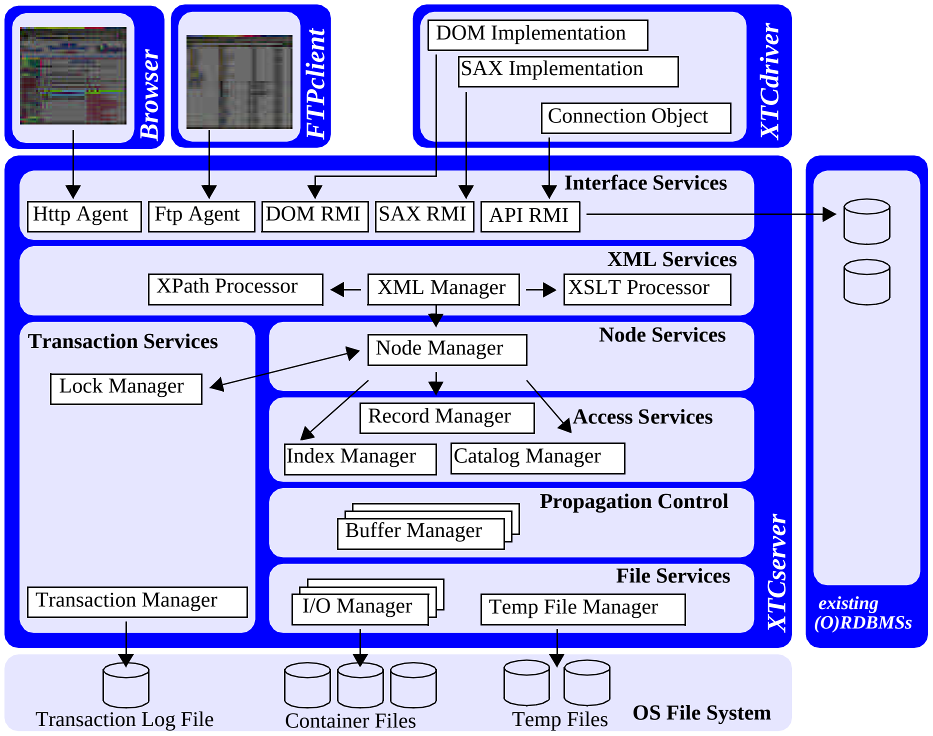

Fig. 2. System architecture overview

tems with native XML storage using techniques already available in these systems.

XML documents can be composed of components of different types including elements, attributes, texts, processing instructions, and comments. These components of XML documents embody externally visible nodes identified by unique IDs, that is, they can be directly referenced at the client side. Our XTCserver is responsible to store such documents as physically ordered records in fixed-length pages which can be adjusted to the space requirements of XML objects (documents or elements). This approach leads to a new kind of XML storage-level management.

```
<book>
  <title>Introduction to DBS</title>
  <content>
    <chapter title="An Overview...">
      <keyword>Database</keyword>
    </chapter>
  </content>
</book>
```

Fig. 1. XML fragment

To motivate our approach, we refer to Fig. 1. At the user level, XML documents consist of trees whose nodes represent the typical XML types elements, attributes, and texts. Our native XML storage structures are designed to achieve effective storage utilization and efficient search to any node in a document tree, while arbitrary modifications of the tree structure and the node contents remain possible. This requires a flexible node identification scheme which is also a mandatory requirement for fine-grained locking protocols on XML documents. To further enhance concurrent operations, we have introduced for the internal XML tree representation two additional node types called *string* and *attribute root* [15] which do not need special consideration here. Transparent at the user level, a string node, for example, keeps the actual value of an attribute or text node which avoids locking of such a node unless its value is explicitly fetched. While instances of the node types text and string may be very large thereby challenging our storage mechanism, the remaining nodes are typically short.

In the following section, we outline the system architecture of the XTCserver. Section 3 discusses the database engine architecture in detail. Section 4 summarizes our performance evaluation results for bulkloading documents and querying simple path expressions. Finally, in sections 5 and 6, we review the related work concerning native XML storage and wrap up with conclusions and some aspects of future work.

2 System Architecture

In this section, we introduce the system architecture embedding our XTCserver in an existing (object-) relational database environment. This approach is in accordance with the questions raised in *ROX: Relational over XML* [13] to support SQL APIs as well as XDP interfaces. Questions controversially discussed so far are "Will the DBMSs of the future be hybrids, storing both relational and XML?" or "Will everything be stored in XML format?" making myriads of SQL systems "legacy applications". While, in this paper, we primarily focus with our XTCserver on the support of XDP operations on native XML structures, we can use/extend our testbed system to future DBMS architectures representing mixed SQL and XQuery systems to run SQL applications on native XML or on hybrid structures concurrently. Therefore, the coarse architectural overview in Fig. 2 accentuates the XML internals. It visualizes a client/server environment, where several client applications can communicate with the XTCserver.

After starting the XTCserver, the remote method invocation agents (RMI) in the interface services layer are instantiated. To execute queries by the XTCserver, we have developed a command line processor enabling the user to connect to the server via the *XTCdriver* and to send queries whose results are written back to the screen or into a file. The XTCdriver is responsible for the connection establishment of client applications. At the server side, the *API RMI* thread handles the processing of the queries passed on by the XTCdriver.

The application programming interface (API) provided by the command line processor may be considered as an interface providing for SQL and XML processing functions. This require a certain kind of preprocessing of the submitted statements. For this reason, the API RMI parses each statement sent by the corresponding command line processor and processes, on the one hand, the recognized XML parts invoking the XTCserver, on the other hand the SQL parts invoking a connected RDBMS. Although a description of such a combined processing is not a focus of this paper, we give a first impression of the XML-enriched SQL interface in Fig. 3.

```
CREATE TABLE myschema.xmldata
(
  filename VARCHAR(50),
  data     XTCXML
)

SELECT data/book/content
      /chapter[keyword=$relatedTerm]/@title
FROM   xmldata, keywords
WHERE  keyword = 'semi-structured'
```

Fig. 3. Combined processing of relational and XML data

The first statement creates the table *xmldata* which contains a column of type *XTCXML* that natively stores XML documents (performed by storing the document into the XTCserver and maintaining a reference to the document in the column *data*). In the second statement a SELECT query is shown which directly evaluates an XPath expression on column *data* of table *xmldata*. Note, the expression *relatedTerm* with the prefixed $-character is a column of the conventional relational table *keywords* (also referenced within the FROM clause) and, in that way, enables directly a join of relational and XML data inside a path expression.

The *XTCengine* is responsible for native XML processing and serves all clients connected to the XTCserver via the *interface services*. The refined architecture and algorithms enabling XML processing inside the *XTCengine* are further discussed in Section 3.

As an additional feature, the XTCserver provides an ftp and http daemon which allow for accessing XML documents via a usual ftp client or a web browser. For these tasks, an *ftp agent* and an *http agent* can be defined which deliver the connection handles of the connected ftp client or web browser at the server side.

3 Database Engine

So far, we have sketched essential aspects of XML processing at the database user level. In this section, we present the XTCserver database engine and the corresponding algorithms to store XML documents. For this reason, the database engine components organized by five hierarchical layers are described in more detail.

3.1 Engine Architecture

To implement a centralized database management system, Härder and Reuter have proposed a mapping model consisting of five layers [14]: *file services* (file management), *propagation control*, *access services* (record and access path management), *node services (*record-oriented, navigational access), *XML services* (non-procedural or algebraic access).

The design of our XTCengine (depicted in Fig. 2) strictly adheres to this widely used reference architecture. Its mapping hierarchy embodies the major steps of dynamic abstraction from the level of physical storage up to the user interface. At the bottom, the database consists of huge volumes of bits stored on non-volatile storage devices, which are interpreted by the DBMS into meaningful information on which the user can operate. With each level of abstraction (stepping upwards), the objects become more complex, allowing more powerful operations and being constrained by a growing number of integrity rules. The uppermost interfaces both support the XML data model, in our case by providing navigational and declarative data access.

File Services. The bottom layer operates on the bit pattern stored on some external, non-volatile storage device and is implemented by the *i/o managers* and the *temporary file manager*. In collaboration with the operating system's file system, this layer copes with the physical characteristics of each type of storage device. Each i/o manager is responsible for a so-called *container file* with a uniform block size.

Propagation Control. This layer implemented by our *buffer managers* provides for pages which are fixed-length partitions of a linear address space and mapped into physical blocks of the system layer below.

Access Services. The next layer implements mapping functions much more complicated than those provided by the two subordinate layers. For performance reasons, the partitioning of data into pages is still visible at this layer. It has to maintain all physical object representations, that is, data records, fields, etc. as well as access paths structures, such as lists, B-trees, and B*-trees, and internal catalog information. These tasks are provided by the *record manager*, *index manager*, and *catalog manager*. Especially tailored to the XML data model, the record manager is responsible to prevent the records from loosing their physical order among each other.

Node Services. The navigational access layer maps physical objects to their logical representations and vice versa. At this interface, the user (or the modules at the next higher layer) navigates through hierarchies of XML nodes. This task is achieved by the *node manager*.

XML services. This layer provides logical data structures with declarative operations or a non-procedural interface to the database. It provides an access-path-independent data model with descriptive languages and contains the *XML manager* which is responsible for declarative document access, e. g., evaluation of XPath queries [6] or XSLT transformations [7].

At the top of our architecture, the agents of the *interface layer* make the functionality of the XML and node services available to common internet browsers, ftp clients, and the *XTCdriver* thereby achieving declarative / set-oriented as well as navigational / node-oriented interfaces.

Transaction management is performed by the *transaction manager*. It provides facilities for logging transaction operations onto disk to enable recovery after a system crash. Furthermore, the transaction manager includes a lock manager which isolates transactional accesses on XML documents using fine-granular node locks [15].

In the following, we will sketch important design issues for the layers which are performance-critical for processing XML data structures.

3.2 I/O Manager

An i/o manager makes read and write operations of fixed-length blocks available for the next higher layer. The blocks are identified by a unique number and read from resp. written into a variable-length container file.

Each container file may have a different block size specified at container creation time. A specific block size is primarily chosen to obtain high storage utilization when storing and maintaining XML documents. In particular, the block size should be somehow adjusted to the storage requirements of the XML objects, that is, neither small XML documents fitting entirely into a block nor records representing XML elements should cause too much fragmentation in a block.

The first block called the index block keeps the block size in two bytes, followed by additional administration data (AD). When an i/o manager opens a container file, it reads the block size from the index block and calculates the number of blocks within the container file by dividing the container file size by the block size.

If the container file is exhausted, we can flexibly extend the container size (extension size adjusted in AD) without any displacements of already allocated blocks.

3.3 Buffer Manager

A buffer manager is assigned to each i/o manager. At instantiation time, it requests the i/o manager assigned to figure out the characteristic block size. Using this parameter and a buffer-size configuration parameter of the XTCserver, the buffer manager dynamically allocates the buffer memory space for the current session. After initialization, the buffer manager is able to service requests from modules residing at the next higher layer.

The pages provided by the buffer manager are typically mapped one-to-one to the blocks of the non-volatile storage. However, the conceptual separation of blocks and pages allows for other flexible mappings, e.g., to provide for shadowing.

A page request is specified by a page number which is a 4-byte integer value. It encodes in the highest byte the container number and in the lowest three bytes the block number where the page is mapped to. A page request causes the buffer manager to search the requested page in the buffer. For this task, a hash table is implemented which uses the page number as the hash key and delivers the position of the page inside the buffer as the hash table value. If the hash table probing fails, the page must be loaded from the container file by the i/o manager. If the buffer is already warmed up, a page—a so-called victim—has to be determined to make room for the newly requested page. Then, page replacement can be performed, that is, the victim page can be freed—if it was modified, it must be written back to the container file before—and the requested page can be loaded. Finally, the buffer manager returns the position of the page inside the buffer to the requesting module after having issued a *fix* command which prevents replacement of the page during its use.

When the requestor has finished its processing within the fixed page, it sends an *unfix* call to the buffer manager which makes the page eligible for replacement or for a further request (a new fix-unfix interval).

To reduce the read and write operations of XTCserver pages on the container file to a minimum, an adequate page replacement strategy should be applied. In our prototype approach, we have chosen to implement the LRD-V2 algorithm [9], where both reference density as well as reference frequency (combined with some aging mechanism) of a page are taken into account. An individual buffer manager for each i/o manager also allows for applying individual page replacement algorithms to XML documents semantically grouped and stored within a single container file.

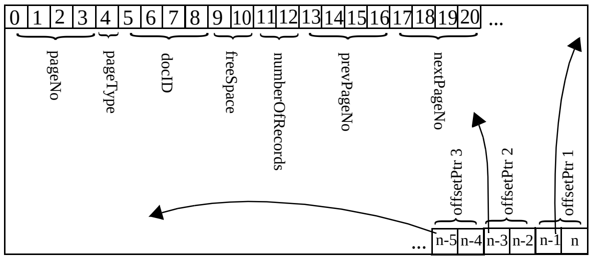

Fig. 4. Record page layout

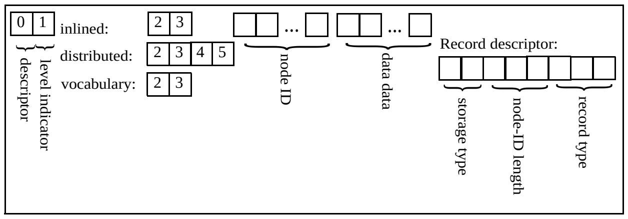

Fig. 5. Record format

3.4 Record Manager

The interface provided by the buffer manager corresponds to a main memory DB with a potentially infinite address space with visible boundaries of fixed-length pages. The record manager referring with the fix-unfix mechanism to these pages is responsible for storing records and access path data into these pages. Some of this data is later transformed into XML document nodes by the node manager (Section 3.7). Because the record manager embodies the core of our XML processing approach, its record storage algorithm also takes the structure of XML documents into account to allow for their optimized processing.

3.4.1 Page and Record Formats

Each node of an XML document is represented as a physical record. Such a record is stored in a page which is flagged by a page type byte as a record page. The left-most depth-first order of the nodes (logical document node order or tree order) within the corresponding XML document is assured by the physical order of the records within the record pages together with so-called level indicators. This is a very important aspect. The reconstruction of subtrees is a very frequent operation in XML database systems; in this way, it can always be done on a clustered set of records. The order of the pages—needed to reconstruct the entire XML document—is preserved by two pointers in each page which reference the previous and next record page. The records within a record page are addressed by offset pointers which are located at the end of the page (corresponding to the record identifiers (RID concept) in [2]). Both together the page number and the offset pointer index constitute the physical address of a record which allows for displacing a record inside a page without changing its externally visible address. The layout of a record page is depicted in Fig. 4.

Before storing a record which corresponds to an XML document node, a new page of type *document catalog* is allocated which keeps the reference to a record page containing the first record of the document. The specific buffer manager allocates a new page for the document catalog before the actual storage procedure is initiated. Because the entire XML document is analyzed before it is stored, the record manager is able to choose an adequate container file to optimize storage utilization (see also Section 3.4.2).

The record manager operation *storeRecord* is used to write a physical record into a record page. The structure of a physical record is shown in Fig. 5.

The first byte of a record represents the record descriptor. The descriptor encodes in the first two bits one of the storage modes *inlined*, *distributed*, or *vocabulary*. *Inlined* materializes the record data as the tail of the record structure, whereas *vocabulary* only keeps a reference to the actual data value in an XML vocabulary. This vocabulary is represented by an ordered list of (surrogate, name) pairs where surrogate contains the internal numeric representation of an element or attribute name. In the case of rather small sets of names, a list implementation using sequential access on both pair items obtains satisfactory performance; larger vocabularies could be indexed by B-trees, in addition. Because XML documents often consist of many elements or attributes with identical names, the vocabulary storage option is a less space-consuming method and, hence, allows to keep larger document parts in the database buffer. If the record data length exceeds the maximum available page free space (a so-called *"long" record*), storage mode *distributed* is chosen storing the record data across several pages. Each record has assigned a unique life-time node ID stored together with the record. Three bits in the middle of the record descriptor encode the number of bytes allocated to keep the node ID. The last three bits of the record descriptor encode the record type corresponding to one of the XML node types (Section 1).

The level indicator stored in byte 1 expresses—in addition to the order given by the physical record order—the relationship of a record to its immediate physical predecessor. When records are mapped to XML document nodes, the level indicator is used to specify relationships like *child-of* or *sibling-of*. We discuss this mapping together with typical values for the level indicator in detail in Section 3.4.3.

The bytes following the level indicator depend on the storage mode set in the record descriptor. For mode *inlined*, bytes 2 and 3 store the length of the data at the end of the record. Mode *distributed* keeps in bytes 2 to 5 the number of the page keeping the initial part of the long record, in mode *vocabulary* bytes 2 and 3 deliver address information for accessing the (surrogate, name) pair in the vocabulary.

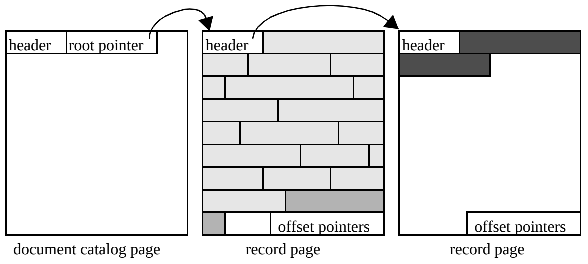

Fig. 6. Initial loading of records

The record description is continued by the bytes for the node ID and possibly the record data (only mode *inlined*). The node ID value allows the node manager supported by the index manager to quickly locate the corresponding physical record (see sections 3.5 and 3.7). Because the node order of an XML document is represented by the physical record order in a page, insert operations of records can cause page overflows and movements of records into another page. In this case, the record manager has to update the physical record addresses for the corresponding node IDs. For this reason, the record manager notifies the index manager when records are moved across page boundaries.

Using our physical record format, the inlined storage of a record of type element or attribute with a typical node-ID length of 3 bytes requires only 7 bytes. This approach allows us to address each node very quickly by its unique life-time ID and, additionally, to store an entire XML document consuming less space than its textual (external) representation (see Section 4).

3.4.2 Inserting Records

Initial loading of entire XML document allocates pages as needed and fills them sequentially. The XML document is parsed and each identified XML node is appended to the so far stored document part. If there is not enough space in the current page, a new page is allocated, the new record is inserted at the first position in the new page, and the next record page pointer of the current page is updated to the just allocated page. This procedure is shown illustrating the insertion of two records in Fig. 6; it guarantees that, if an XML document is distributed across several pages, each page is filled as much as possible.

If an already existing XML document is modified (e.g. by editing the document), a new record is always inserted directly after the record specified as the logical predecessor in document order. In contrast to the loading of an entire XML document, the insert operation may need page splitting. This means, the new record is inserted immediately after the specified record by moving the succeeding records, if any, inside the page. If there is not enough free space in the page for inserting the new record, a new page is allocated (anywhere) and the set of all records in the original page is divided in two parts of approximately the same size. The first part remains in the original page, whereas the

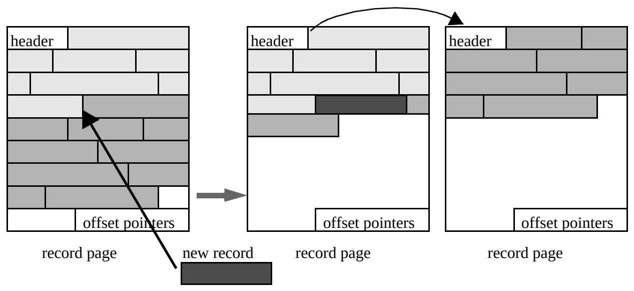

Fig. 7. Insertion causing a page split

second part is moved into the new page. Fig. 7 depicts the insertion of a new record which causes a page split. Because the affected records are distributed equally and free space is left to adjust for future insertions, the growth of the structure only requires a minimum of page splits.

If the size of a long record to be inserted exceeds the fixed-length page size, the long value is divided in parts each stored into a single page and reachable via its representing record using storage mode *distributed* as described in Section 3.4.1. Assume, a new record exceeds the length of one page, then the record is distributed over two pages and inserted as illustrated in Fig. 8.

Our record management algorithms guarantee effective and efficient page utilization. In particular, if entire XML documents are stored at a time, we achieve a very high page occupancy, because we can control the page allocation.[1] Even if the structure of these documents is modified later on, we may expect a degree of page filling of more than 90% in most cases.

3.4.3 Maintaining Record Order

When storing an XML document, the record manager first initializes an analyzing SAX parser which checks the entire document. For each XML node discovered in the document, the SAX parser calculates the expected size of its physical representation. In this way, the analyzing SAX parser can estimate the expected physical size of the entire document and choose the best fitting container file.

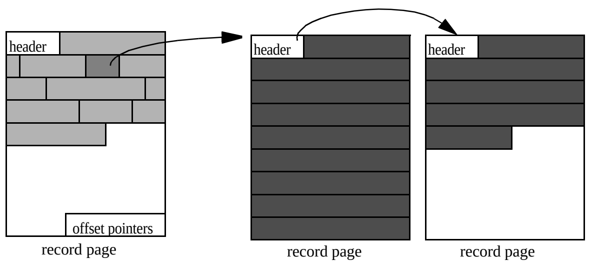

Fig. 8. Inserting long records

1. Using XML documents ranging up to 100MB with structural accordance to the XMark benchmark documents in [23], we have achieved an average page occupancy of 98%.

record type	node ID	level	record data
element	1	0	book
element	2	1	title
text	3	2	
string	4	3	Introduction to DBS
element	5	1	content
element	6	2	chapter
attribute	7	3	title
string	8	4	An Overview...
element	9	3	keyword
text	10	4	
string	11	5	Database

Fig. 9. Parameters for XML document creation - example

At the end of the analyzing phase, the node manager invokes a second SAX parser which guides the actual storage mechanism. For each XML node encountered, the record manager requests a new node ID from the catalog manager and calls the record manager to store the XML node as a physical record. Furthermore, it passes the (node ID, RID) pair on to the index manager which modifies the associated B-tree.

To store a record, parameters for record type, record data, and level indicator have to be supplied. The type of each XML node is mapped to the record type parameter. Both the name of an element node and the name of an attribute node are stored into an XML vocabulary, the information to address the names in the vocabulary is mapped to the physical record as described in Section 3.4.1. The value of the attribute node is stored *inlined* into an additional record of type string. Text nodes do not have a name, they only contain their value which is mapped to a record of string type stored (also *inlined*) after the actual text record.

As already explained in Section 3.4.1, the record manager takes care of the order of the related XML nodes, but does not maintain node relationships like *child-of* or *sibling-of*. These node relationships are expressed using the level indicator which describes the depth of the node position to the XML document root. Starting with level indicator value 0 at the document root node, each new opening element tag increases the level indicator value by 1. In other words: sibling nodes carry the same level value, child nodes have the level value of their parent node increased by 1. Attribute nodes are stored with the level indicator value increased by 1 of their corresponding element node. String nodes are stored with the level value of their owning attribute or text node increased by 1. Using the stored level identification, it is very simple to reconstruct an XML fragment or the entire XML document. Fig. 9 illustrates the parameters of the records maintained for the XML fragment in Fig. 1.

3.5 Index Manager

The index manager provides methods to maintain sequential lists and the well-known B- and B*-trees [8]. An entry present in such access paths consists of a (key, value) pair, both key and value of variable length.

In our approach, we assign stable IDs, valid for their life time, to XML nodes. For this reason, a B-tree is used to map such logical node IDs to the locations (RIDs) of their corresponding records. Note, because physical records can be displaced by document modifications, this mapping guarantees ID stability (also called database key concept). The availability of these stable IDs is critical for direct access to an arbitrary node, update flexibility, and performance of node indexing. Therefore, we represent the node ID and its 6-byte physical address (RID: 4-byte page number and 2-byte offset index) as a (key, value) pair in the B-tree.

If a new XML node is inserted by the record manager, a B-tree entry is created whose key represents the unique node ID and whose RID references its current physical record location. Despite of the indirection, a record specified by the ID of its XML node can be located very rapidly. If the corresponding record is modified or moved inside the page, no additional updates in the B-tree are required, because displacement inside a page does not change the physical address (RID). Only modifications which cause the movement of the record across a page boundary requires the update of the physical record address to keep the B-tree entry consistent.

3.6 Catalog Manager

The catalog manager maintains the catalog data of the XTCserver. Therefore, at the first XTCserver startup the catalog manager creates the database catalog page, a B-tree for managing the node IDs, and a so-called *master document* into which the metadata of the XTCserver is stored. The identifier of the B-tree and the master document are kept in the catalog page which is always resident in main memory from XTCserver startup on. The catalog manager also provides for access information to the structural indexes which are introduced in the next section.

3.7 Node Manager

The node manager implements the navigational access layer and offers methods for node-oriented processing of XML documents. Furthermore, XML nodes can be inserted into an XML document at an arbitrary position and single nodes or subtrees can be deleted, to support flexible document modification. Most of the DOM API methods are implemented with direct invocations of the node manager.

Starting from a context node, specific methods support the navigation to the parent node, the previous or next sibling node, or the first or last child node, and the retrieval of attributes or entire XML fragments. Having additional methods to identify an XML document's root node and to get and set single XML node values, the node manager as part of the navigational access layer provides the essential functionality to support the next higher layer—implemented by the XML manager—to process declarative queries through node-oriented operations.

3.7.1 Node-Oriented Navigation

Because the node manager performs all navigational tasks by applying the methods provided by the record manager, all operations have to be evaluated on the physical record structures. As mentioned in Section 3.4, these tasks can be efficiently performed for operations executed on records which are physically clustered. For example, this can be done very rapidly for accessing a complete XML fragment, the first sibling or an arbitrary attribute of an element, or the value of an attribute or text node.

In contrast, locating at the DOM API distant nodes via one of the four structural relationships parent-of, previous-sibling-of, next-sibling-of, and last-child-of for a given context node is (especially for very large XML documents) an expensive operation, because the entire XML document part between the context node and the requested node has to be fetched. This necessity results from the fact that only the level indicator of a record can be used to determine the corresponding node relationships. Hence, many records may have to be scanned in order to locate the record with the level value searched.

To efficiently support these structural operations, we again resort to the idea of addressing these nodes by B-trees. Hence, we provide so-called structural indexes introduced in the next section.

3.7.2 Structural Indexes on Demand

Structural indexes are implemented by four B-trees which store together with the ID of the context node the corresponding ID of the previous and next sibling, the last child, and the parent for a given context node. To keep the sizes of these B-trees as small as possible, we only index structural relationships that are actually used, that is, pairs of node IDs are inserted on demand into the resp. B-trees. If a node has to be located in the course of query processing via such a structural relationship and if it is not already available in the corresponding index, the record manager has to scan all records of a page and possibly, in the same way, a large set of subsequent pages to identify the requested node by comparing the level indicators of each record. The so determined node ID is finally inserted into the index and can be used in future references. A detailed example for this procedure is given in the now following paragraph.

Assume, the node manager has to return the next-sibling record of the context node specified by its ID, say ID=0815 for our discussion. First of all, the node manager accesses the next-sibling B-tree and looks up the ID attached to ID=0815. In case, this B-tree access is successful, the next sibling of 0815, say 4711, can be immediately located via the node-ID index which delivers the RID for 4711. Otherwise—no (ID, ID) pair present for ID=0815 as a key in the next-sibling B-tree—, the node manager has to search it "the hard way". Accessing the node-ID B-tree for ID=0815 delivers the RID of the context node under consideration. Hence, the referenced page can be directly fetched into the DBMS buffer and, using the offset pointer index at the bottom of this page, the record representing node 0815 can be exactly located. Then, the node manager scans all records succeeding record 0815 until the first record of type element is located having an identical level indicator (this is record 4711). Finally, the located record is transformed into its external representation and returned.

Now the pair (0815, 4711) can be inserted on demand into the structural index for next siblings to avoid performance-critical scan operations on future sibling requests. Because our XTCserver guarantees life-time stable node IDs, the (ID, ID) pairs in these structural indexes only have to be updated on node deletions and insertions of new sibling and child nodes. The more frequent movements of sets of records caused by record insertions and page splits do not affect the index data.

When entire XML fragments can be efficiently reconstructed by local scans of a single page content, it is not necessary to index the next-sibling information. Because structural indexing is performed on demand and only in situations where large subtrees can be skipped using such relationship information, our structural index mechanism accelerates frequently used navigation paths in documents and, at the same time, avoids excessive storage consumption of static schemes. Performance gains are shown in Section 4.

3.8 XML Manager

The XML manager implements the highest layer in the system layer hierarchy—the nonprocedural access layer responsible for set-oriented processing. So far, the node manager makes a number of methods available to store and retrieve documents, document fragments, or single nodes.

For evaluating an XPath expression, the expression is initially separated into single search steps (e. g., */book/content/chapter* is separated into */book*, */content*, and */chapter*). Next, an iterator—keeping a list of nodes—is initialized with the root node of the document for which the expression is evaluated. Then, the nodes contained in the iterator are recursively replaced by their child nodes which satisfy the condition of the step currently processed (e. g., while processing the step */content*, the root node <*book*> will be replaced by its child node <*content*>). Finally, the iterator contains all nodes which qualify for the path expression referenced. A performance evaluation of this algorithm is given in Section 4.2.

4 Performance

In this section, we give a first impression of the overall system performance. The XTCserver is completely implemented in Java using the *Standard Edition 1.4.2*. In order to evaluate the performance of bulkloading and querying entire XML documents, we run the XTCserver on an *IBM eServer xSeries 235* with two *Intel Xeon-A* 2400MHz processors. The machine executes *Microsoft Windows Server 2003 Enterprise Edition*. Both the B-tree index and the XML documents are stored in separate container files, each assigned with a buffer manager providing for 500 pages of 4096 bytes. The container files initially consist of 10,000 pages and are automatically extended by 2,500 pages if a container runs out of space. The bulkload operation for a document is performed transaction safe, this means, it is executed within a transaction which is isolated from other concurrently running transactions and which is completely logged to be recoverable after a system crash. Although client and server are running on the same machine in our performance evaluation environment, each XML document inserted is copied from the XTCdriver to the XTCserver via a local TCP/IP connection to reflect normal client/server cooperation.

We used the *xmlgen* tool of the XMark benchmark for XML database systems [23] to generate the documents which are loaded into the XTCserver. The document size varied from 1MB up to 100MB. These documents generated by xmlgen correspond to a range from ca. 49,000 up to 4.2 million XML nodes (in detail listed in Fig. 11).

4.1 Bulkloading Performance

The storage space required for the XML documents is listed in Fig. 10. The documents generated for the XMark benchmark primarily use short element and attribute names (like person, item, name, id, ...). Although all text values are stored by the XTCserver in two separate records (text record and string record with the actual value) and a unique ID is assigned to each node, our space-saving physical record layout based on vocabulary use and bit encoding of structural information reduces the storage requirements by nearly 10 percent (compared to the original textual representation of the document). For XML documents containing longer element and attribute names (and perhaps additional XML namespaces), we expect even more substantial space savings.

The elapsed times for bulkloading XML documents (varying from 1MB up to 100MB) are shown in Fig. 11. *Storing as document* means that the XML document is parsed and each node is stored as a record and referenced by a key which is inserted into the node-ID B-tree as described in Section 3.4. *Storing as blob* causes the record manager to store the document as a single record distributed across a large number of pages with only one ID in the B-tree addressing the blob.

Of course, storing XML documents and assigning to each record an individual ID takes much longer than just "copying a file into pages" (blob option)—roughly a factor of 10 in our experiments. Nevertheless, the system is scaling with the document sizes.

document size (bytes)	inlined (4096B pages)	inlined (bytes)	vocabulary (4096B pages)	vocabulary (bytes)
1,182,547	285	1,167,360	264	1,081,344
2,622,586	653	2,674,688	582	2,383,872
5,194,897	1294	5,300,224	1135	4,648,960
7,837,417	2021	8,278,016	1750	7,168,000
10,734,292	2744	11,239,424	2366	9,691,136
26,170,648	6990	28,631,040	6012	24,625,152
52.244.461	13943	57,110,528	11950	48,947,200
79.098.184	21658	88,711,168	18475	75,673,600
104.732.949	28474	116,629,504	24203	99,135,488

Fig. 10. Storage requirements for XML documents

document size	number of XML nodes	storing as document	storing as blob
1 MB	49,172	10sec	615msec
2.5 MB	107,446	26sec	2.4sec
5 MB	212,076	53sec	5.2sec
7.5 MB	319,548	1min 22sec	8.5sec
10 MB	435,121	1min 55sec	12sec
25 MB	1,060,178	4min 55sec	28sec
50 MB	2,117,748	10min 40sec	1min 2sec
75 MB	3,194,869	16min 57sec	1min 35sec
100 MB	4,232,422	23min 5sec	2min 12sec

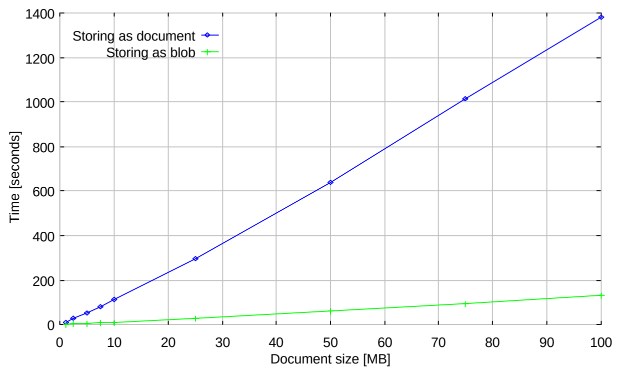

Fig. 11. Bulkloading times for XML documents

4.2 Structural Index

As described in Section 3.8, our iterator-based path evaluation algorithm mainly has to determine the children and siblings of given nodes in order to get the result set. For these tasks, the operations first-child and next-sibling are used. The first-child operation can be efficiently performed because the record manager often locates the first child only a few bytes behind the parent node within the same page. This favorable behavior is contrasted by the next-sibling operation where, due to the tree order, a given context node is typically separated from its next sibling by a (probably very large) subtree stored directly behind this context node.

In order to demonstrate the performance benefits of the structural index (Section 3.7.2), we evaluated the XMark path query Q1 */site/people/person[@id='person1']* on the XML documents generated by xmlgen. Because we did not support any further indexes besides the node-ID and next-sibling B-trees in this experiment, we had to locate and fetch all *person* nodes in */site/people* and test the predicate *@id='person1'* on each node. Fig. 12 (column 'no index') lists the query evaluation times obtained.

The last two columns in Fig. 12 refer to experiments supported by a structural index on demand. Processing the first query, the empty index increases the query evaluation time (compared to 'no index') because the next-sibling relationships are inserted into the index. Further queries of the same type on the 100 MB document profit from the indexed next-sibling IDs and consume on the average only 76% of the initial execution time. In this case, the structural index does not lead to a performance improvement as high as expected. This is caused by the large number (22,439) of person elements distributed across a large set of pages. Because each of the person elements contains only a few child nodes, the subtrees separating siblings in tree order are small such that the index advantage that allows jumping over a large set of pages gets lost. Nevertheless, query processing with or without any additional index scales with the document size.

A more significant gain by the structural index use is achieved in our second experiment. While the results illustrated in both columns 'no index' and 'first query' in Fig. 13 are comparable to their counterparts in Fig. 12, the column 'further queries', however, reveals dramatic gains attributed to the structural index. The index-supported query */site/regions/africa/item[@id='item1']* is executed in 344 milliseconds which is only 3% of the execution time on a freshly inserted document without any index data available. Here, the relevant part of the 100 MB document contains in region *africa* 483 different *item* elements; each subtree of an item element consists of up to 200 nested nodes which can be skipped in the next-sibling operation if matching index entries (in a structural index on demand) already exist. The execution times of the *item* query which

document size	person elements	no index	first query using index	further queries using index
1 MB	254	343msec	500msec	250msec
2.5 MB	560	743msec	938msec	485msec
5 MB	1121	1.3sec	1.6sec	935msec
7.5 MB	1707	1.9sec	2.4sec	1.4sec
10 MB	2294	2.5sec	3.1sec	2.0sec
25 MB	5609	6.1sec	8.0sec	4.7sec
50 MB	11219	12.4sec	15.2sec	8.8sec
75 MB	16956	18.4sec	23.1sec	13.8sec
100 MB	22439	24.6sec	30.6sec	18.7sec

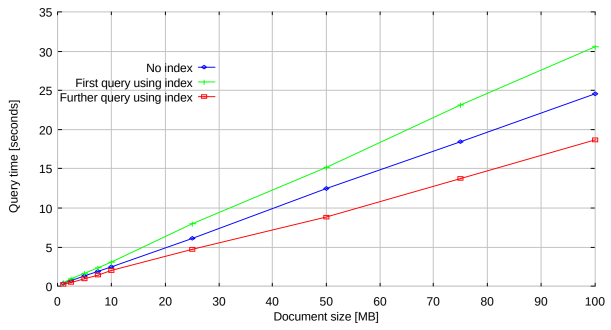

Fig. 12. Querying persons using the structural index

document size	item elements	no index	first query using index	further queries using index
1 MB	4	112msec	141msec	19msec
2.5 MB	11	266msec	313msec	32msec
5 MB	23	547msec	594sec	47msec
7.5 MB	35	797msec	828sec	63msec
10 MB	48	1.1sec	1.15sec	78msec
25 MB	120	2.8sec	2.9sec	110msec
50 MB	241	5.5sec	5.7sec	219msec
75 MB	364	8.5sec	8.8sec	328msec
100 MB	483	11.2sec	11.7sec	344msec

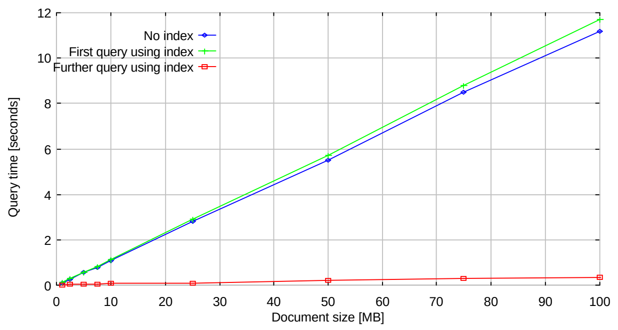

Fig. 13. Querying items using the structural index

clearly owed the performance improvements to the index use are illustrated in Fig. 13.

Considering the query evaluation times and number of elements analyzed in the queries mentioned before, an arbitrary element node and its corresponding attribute are located and tested with the predicate *@id='...'* within 0.8 milliseconds—an impressive performance behavior attributed to the node-ID B-tree delivering the record identifiers for the direct node accesses.

5 Related Work

The detailed design of physical storage structures for XML data and their processing inside a DBMS are hardly discussed in scientific papers. The work closest to our approach is related to the eXist database system [21] which also stores consecutive XML nodes in fixed-length pages and uses a B+-tree to maintain physical record addresses and node IDs. The node IDs are determined by an extension of the numbering scheme proposed in [19] using a complete k-ary tree for each node level. Because the numbering scheme allows for very fast location of, e. g., parent, child, or sibling nodes, eXist is mainly designed for data retrieval. On the other hand, a simple node insertion may lead to nearly the complete renumbering of the entire (possibly very large) XML document. This behavior is not desirable for OLTP applications.

Lore [20] is a database system designed for the management of semi-structured data with a focus on query processing. The data managed is not confined to a schema and may even be irregular or incomplete. As far as storage is concerned, the XML document is mapped to the *object exchange model* (OEM [22]). Each object is stored in a so-called slot within a fixed-length page; the physical order of the objects corresponds to the depth-first traversal of the XML document. For these semi-structured objects, the SQL-like *Lore Language* (Lorel [1]) provides powerful processing capabilities.

Natix [10] splits an XML document tree into several subtrees which are separately stored. The related split algorithm can be guided by the so-called *split matrix*. Finally, each subtree is stored as a record in a fixed-length page. Natix supports full transactional XML processing capabilities with concurrency control as well as logging and recovery tailored to the subtree storage structures. To speed up query execution, XML documents may be indexed by the *XASR* index.

Timber [18] splits an XML document into single nodes and assigns a node ID consisting of start-, end-, and level-position of the node. Reserved gaps in the numbering scheme are supposed to avoid a too frequent renumbering of the document. Single nodes are stored in a clustered mode in document-order by the Shore data manager [5]. But Shore exhibits considerable overhead when dealing with small objects, so several nodes must be packaged in page-size containers. A further problem is that Shore destroys the node cluster if update operations force a page split. Timber implements a mature tree algebra for query evaluation; transaction management is not adapted to the semi-structured data model.

6 Conclusion and Future Work

Native storage of XML documents in database systems is an important practical problem. So far, (object-) relational database systems do not support really native XML storage and concurrency control by their engine. On the other hand, native XML database systems are mostly designed for efficient data retrieval, although transaction-safe processing of XML data is desirable. The relevant ideas for transaction management (e. g., logging and recovery or isolation) are in the majority of cases described in a sketchy way.

In our paper, we have presented the XTCserver, a prototype database system which supports native XML processing. Our XTCserver is able to extend an existing (object-) relational database environment to support native XML data storage and relational tables, that is, hybrid structures, at a time.

The XTCserver is strictly designed in accordance to the well-known five-layer database architecture. For the implementation of file management, any kind of data is stored in block-structured container files. The propagation control is implemented by a buffer manager which offers a potentially infinite page-structured address space to the next higher components. The core mechanisms for native XML data storage are implemented by the access services which consist of the index, catalog, and record managers, taking into account both the level indicator and the record order which represent the logical structure of an XML document. By the use of a B-tree, unique node IDs are assigned to each record resp. corresponding XML node. Because they remain valid for their life time, they can be used to directly access these records from anywhere, for example, using index structures to invert XML documents or node contents, independently of any document modifications. The node manager as part of the node services layer maps the nodes of an XML document to internally managed records. The XML manager as the core part of the XML services representing a non-procedural access layer processes collection-based queries.

Our current steps integrate concurrency control mechanisms for fine-granular collaborative XML document processing. In particular, we take advantage of the update flexibility of and selective access to XML nodes achieved by our stable numbering mechanism (via the node-ID B-tree) in the XTCserver. Because XML elements can be identified and directly accessed, fine-grained lock acquisition at the node-level can be achieved to synchronize multi-user access on server-controlled XML documents [15]. We want to systematically evaluate this new approach in our XTCserver environment; first results are already available [16]. Finally, in future steps, fine-grained concurrency control combined with logging and recovery mechanisms will allow us to perform relational and XML data accesses in a single transaction. More support for processing of collection-based queries with update extensions for XML documents are part of future work in the non-procedural access layer. Additionally, the integration of path-based index structures will support our structural indexes and will allow us to reach even more competitive processing times for typical XML benchmark queries.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener. The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries 1:1, 68-88 (1997)

[2] M. M. Astrahan et al. System R: Relational Approach to Database Management. ACM Transactions on Database Systems 1:2, 97-137 (June 1976)

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon. XQuery 1.0: An XML Query Language, W3C Working Draft (Aug. 2003)

[4] D. Brownell. SAX2. O'Reilly (Jan. 2002)

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin. Shoring Up Persistent Applications. Proc. ACM-SIGMOD 1994, 383-394, Minneapolis, Minnesota, USA (May 1994)

[6] J. Clark, S. DeRose. XML Path Language (XPath) Version 1.0, W3C Recommendation (Nov. 2000)

[7] J. Clark. XSL Transformations Version 1.0. W3C Recommendation (1999)

[8] D. Comer. The Ubiquitous B-Tree. ACM Computing Surveys 11: 2, 121-137 (June 1979)

[9] W. Effelsberg, T. Härder. Principles of Database Buffer Management. ACM Transactions on Database Systems 9:4, 560-595 (Dec. 1984)

[10] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann. Natix: A Technology Overview. A.B. Chaudri et al. (Eds.): Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Erfurt, Germany, LNCS 2593, Springer, 2003, 12-33 (Oct. 2002)

[11] D. Florescu, D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Rapporte de Recherche No. 3680, INRIA, Rocquencourt, France (Aug. 1999)

[12] J. Freire, J. Siméon. Adaptive XML Shredding: Architecture, Implementation, and Challenges. 28th Int. Conf. on Very Large Data Bases, EEXTT Workshop, 104-116, Hong Kong, China (Aug. 2002)

[13] A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, M. Mörschel. ROX: Relational Over XML. Proc. 30th Int. Conf. on Very Large Data Bases, Toronto (Sept. 2004)

[14] T. Härder, A. Reuter. Concepts for Implementing a Centralized Database Management System. Proc. Int. Computing Symposium on Application Systems Development, Nürnberg, Germany, 28-60 (March 1983)

[15] M. Haustein, T. Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. Proc. 7th East-European Conference on Advances in Databases and Information Systems, Dresden, Germany, 88-102 (Sept. 2003)

[16] M. Haustein, T. Härder. Adjustable Transaction Isolation in XML Database Management Systems. Proc. 2nd Int. XML Database Symposium (XSym 2004), Toronto, Canada (August 2004)

[17] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne. Document Object Model (DOM) Level 2 Core Specification. Version 1.0, W3C Recommendation (Nov. 2000)

[18] H. V. Jagadish, S. Al-Khalifa, A. Chapman. TIMBER: A native XML database. The VLDB Journal 11:4, 274-291 (Oct. 2002)

[19] Y. K. Lee, S.-J. Yoo, K. Yoon. Index Structures for Structured Documents. Proc. ACM 1st Int. Conference on Digital Libraries, 91-99, Bethesda, Maryland, USA (March 1996)

[20] J. McHugh, S. Abiteboul, R. Goldman. D. Quass, J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record 26:3, 54-66 (1997)

[21] W. Meier. eXist: An Open Source Native XML Database. A. B. Chaudri et al. (Eds.): Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Erfurt, Germany, LNCS 2593, Springer, 2003, 169-183 (Oct. 2002)

[22] Y. Papakonstantinou, H. Garcia-Molina, J. Widom. Object Exchange Across Heterogeneous Information Sources. Proc. 11th Int. Conference on Data Engineering, 251-260, Taipei, Taiwan (March 1995)

[23] A. Schmidt, F. Waas, M. Kersten. XMark: A Benchmark for XML Data Management. Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong, China, 974-985 (Aug. 2002)

[24] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. ACM SIGMOD, Madison, Wisconsin, USA, 204-215 (June 2002)

[25] M. Yoshikawa, T. Amagasa. XRel: A Path-Based Approach to Storage and Retrieval XML Documents using Relational Databases. ACM Transactions on Internet Technology 1:1, 110-141 (Aug. 2001)