The VLDB Journal (2002) 11: 292–314 / Digital Object Identifier (DOI) 10.1007/s00778-002-0080-y

Anatomy of a native XML base management system

Thorsten Fiebig[1], **Sven Helmer**[2], **Carl-Christian Kanne**,[3], **Guido Moerkotte**[2], **Julia Neumann**[2], **Robert Schiele**[2], **Till Westmann**[3]

[1] Software AG; e-mail: Thorsten.Fiebig@softwareag.com
[2] Universität Mannheim; e-mail: [mildenbe|rschiele]@uni-mannheim.de, [helmer|moerkotte]@informatik.uni-mannheim.de
[3] data ex machina GmbH; e-mail: [kanne|westmann]@data-ex-machina.de

Edited by Alon Y. Halevy. Received: December 15, 2001 / Accepted: July 1, 2002
Published online: December 13, 2002 –

Abstract. Several alternatives to manage large XML document collections exist, ranging from file systems over relational or other database systems to specifically tailored XML base management systems. In this paper we give a tour of Natix, a database management system designed from scratch for storing and processing XML data. Contrary to the common belief that management of XML data is just another application for traditional databases like relational systems, we illustrate how almost every component in a database system is affected in terms of adequacy and performance. We show how to design and optimize areas such as storage, transaction management – comprising recovery and multi-user synchronization – as well as query processing for XML.

Keywords: XML – Database

1 Introduction

As XML [7] becomes widely accepted, the need for systematic and efficient storage of XML documents arises. For this reason we have developed Natix, a native XML base management system (XBMS) that is custom tailored to the processing of XML documents. A general-purpose XBMS for large-scale XML processing has to fulfill several requirements: (1) To store documents effectively and to support efficient retrieval and update of these documents or parts of them; (2) To support standardized declarative query languages like XPath [9] and XQuery [5]; (3) To support standardized application programming interfaces (APIs) like SAX [30] and DOM [22]; (4) Last but not least, a safe multi-user environment via a transaction manager has to be provided including recovery and synchronization of concurrent access.

A concrete example for an application domain requiring (1)–(4) are life sciences. There, annotated biosequences (DNA, RNA, and amino acid) are a predominant form of data. The sequences and especially their annotations are commonly represented using XML, making (1) an obvious requirement. Typically, the annotated sequence data is processed by a mix of tools for generation, visualization, (further) annotation, mining, and integration of data from different sources. Existing XML tools relying on DOM or SAX interfaces need to be integrated (3). The emergence of new scientific methods regularly requires new tools. Their rapid development is facilitated by declarative XML query languages because they render trivial the frequently recurring task of selecting sequences based on their annotation (2). Each sequence is analyzed using several costly experiments performed concurrently. Their results are valuable annotations added to the sequence data. Obviously, full-fledged transaction management reduces the risk of wasting resources (4). Other application areas include online-shopping with product catalog management and logistics applications.

We are aware that several approaches based on traditional database management systems (DBMSs) exist, e.g., storing XML in relational DBMSs or object-oriented DBMSs [13,18, 25,29,39–42]. We believe, however, that a native XML base management system is the more promising solution, as approaches mapping XML onto other data models suffer from severe drawbacks. For example, let us look at the impact of mapping XML documents onto relational DBMSs on the storage of those documents. First, we have to decide on the actual schema. On the one hand, if we take a document-centric view we could retain all information of one document in a single data item, e.g., a CLOB (Character Large OBject). This is ideal for handling whole documents, but if we want to manipulate fragments of documents we would have to read and parse the whole document each time. On the other hand, if we take a data-centric view, each document is broken down into small parts, e.g., the nodes of a tree representation of an XML document. Obviously, handling parts of documents is now much more efficient, whereas importing or exporting a whole document has become a time-consuming task. This dilemma exemplifies a potential for improvement exploitable only by native XML base management systems. Opportunities for improvements are not limited to the storage layer, as we illustrate in this paper.

We cover all major components of Natix' runtime system. Those using existing techniques are discussed only briefly, those introducing new techniques are discussed in greater detail. Besides the system architecture with its selected list of applied techniques, we contribute the following to the state of the art of XBMSs. We introduce a storage format that clus-

ters subtrees of an XML document tree into physical records of limited size. Our storage format meets the requirement for the efficient retrieval of whole documents and document fragments. The size of a physical record containing the XML subtree is typically far larger than the size of a physical record representing a tuple in a relational database system. This affects recovery. To improve recovery in the XML context, we developed the novel techniques *subsidiary logging* to reduce the log size, *annihilator undo* to accelerate undo and *selective redo* to accelerate restart recovery. To allow for high concurrency a flexible multi-granularity locking protocol with an arbitrary level of granularities is presented. This protocol guarantees serializability even if transactions directly access some node in a document tree without traversing down from the root. Note that existing tree locking protocols fail here. Evaluating XML queries differs vastly from evaluating SQL queries. For example, SQL queries never produce an XML document; neither as a DOM tree nor as a string or a stream of SAX events. Obviously, a viable database management system for XML should support all these representations. Natix' query execution engine is not only flexible enough to do so but also highly efficient.

The rest of the paper is organized as follows. The next section presents the overall architecture of the system. The storage engine is the subject of Sect. 3. This is followed by a description of the transaction management in Sect. 4. Next, we take a look at the query execution engine in Sect. 5. Finally, we conclude our paper with an evaluation (Sect. 3.3) and a summary (Sect. 6).

2 Architecture

This section contains a brief overview of Natix' system architecture. We identify the different components of the system and their responsibilities. Forward pointers refer to sections describing these components in greater detail.

Natix' components form three layers (see Fig. 1). The bottom-most layer is the *storage layer*, which manages all persistent data structures. On top of it, the *service layer* provides all DBMS functionality required in addition to simple storage and retrieval. These two layers together form the *Natix engine*.

Closest to the applications is the *binding layer*. It consists of the modules that map application data and requests from other APIs to the Natix Engine Interface and vice versa.

2.1 Storage layer

The storage engine contains classes for efficient XML storage, indexes and metadata storage. It also manages the storage of the recovery log and controls the transfer of data between main and secondary storage. An abstraction for block devices allows to easily integrate new storage media and platforms apart from regular files. Details follow in Sect. 3.

2.2 Service layer

The database services communicate with each other and with applications using the *Natix Engine Interface*, which provides a unified facade to specify requests to the database system. These requests are then forwarded to the appropriate component(s). After the request has been processed and result fields have been filled in, the request object is returned to the caller. Typical requests include 'process query', 'abort transaction' or 'import document'.

There exist several service components that implement the functionality needed for the different requests. The Natix *query execution* engine (NQE), which efficiently evaluates queries, is described in Sect. 5. The *query compiler* translates queries expressed in XML query languages into execution plans for NQE. Additionally, a simple compiler for XPath is available. They both are beyond the scope of the paper. *Transaction management* contains classes that provide ACID-style transactions. Components for recovery and isolation are located here. Details can be found in Sect. 4. The *object manager* factorizes representation-independent parts for transferring objects between their main and secondary memory representations since this transformation is needed by several APIs.

All of these components bear challenges with respect to XML, which are related to the different usage profiles (coarse grain vs small grain processing). Typically, a simple mapping of operations on coarse granularities to operations on single nodes neutralizes a lot of performance potential. If both access patterns have to be supported in an efficient way, sophisticated techniques are needed.

2.3 Binding layer

XML database management is needed by a wide range of application domains. Their architectural and interface requirements differ. Apart from the classic client-server database system, there are scenarios with Internet access, possibly using protocols like HTTP or WebDAV [19]. For embedded systems it might be more appropriate to use an XML storage and processing library with a direct function call interface. For legacy applications that can only deal with plain files, the database has to be mounted as a file system. Other interfaces will arise when XML database systems are more widely used.

The responsibility of the binding layer is to map between the Natix Engine Interface and different application interfaces. Each such mapping is called a *binding*.

Applications may call the Natix Engine Interface directly. However, for rapid application development, it is often desirable to have interface abstractions that are closer to the applications domain. An example for such a higher-level API is the file system interface, a demonstration of which is available for download [12]. Using this binding, documents, document fragments, and query results can be accessed just like regular files. The documents' tree structure is mapped into a directory hierarchy, which can be traversed with any software that knows how to work with the file system. Internally, the application's file system operations are translated into Natix Engine Interface requests for exporting, importing, or listing documents.

Wherever feasible, the specification of a request to the Natix Engine is not only possible using C++ data types, but also by a simple, language independent string. A small parser is part of the Engine Interface. It translates strings into request objects. This simple control interface for the system can easily

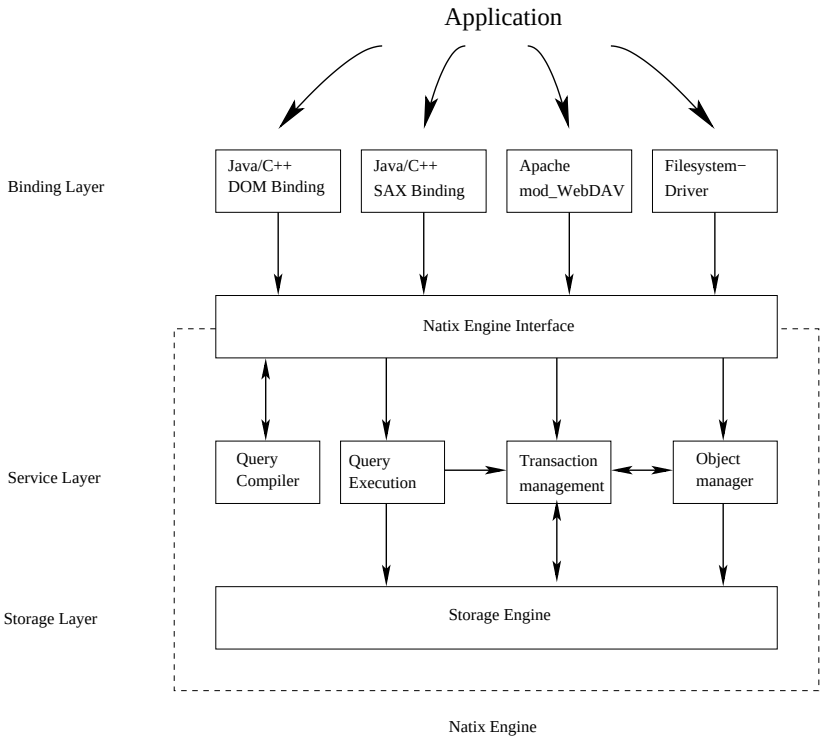


Fig. 1. Architectural overview

be incorporated into generic high-level APIs: by using request strings as URLs, for example, the HTTP protocol can be used to control the database system.

3 Storage engine

At the heart of every data base management system lies the storage engine that manages all persistent data structures and their transfer between main and secondary memory. The system's overall speed, robustness, and scalability are determined by the storage engine's design.

We briefly summarize the relevant techniques employed and elaborate on our novel XML storage method. Some query performance figures for the different storage formats follow. Descriptions of indexing techniques conclude the section.

3.1 Architecture

Storage in Natix is organized into *partitions*, which represent storage devices that can randomly read and write disk pages. Disk pages are logically grouped into *segments*. There are different types of segments, each implementing a different kind of object collection. Disk pages resident in main memory are managed by the *buffer manager*, and their contents are accessed using *page interpreters*.

The following paragraphs will illustrate the responsibilities of the individual components.

3.1.1 Segments

Segments export the main interfaces to the storage system. They implement large, persistent object collections, where an object may be larger than a page (depending on the segment type).

The data type specific operations on the multi-page data structures are mapped onto (sequences of) operations on single pages. The segment classes form a hierarchy, the base class of which factorizes common administrative functions like free space and metadata management. The page collection used to store the object collections is maintained using an extent-based system [45] that organizes segments into consecutive page groups (*extents*) of variable size. Intra-segment free space management is done using a Free Space Inventory (FSI) [34] describing the allocation state and free space on pages. A caching technique similar to [28] is used.

The most important segment types are standard *slotted page segments* supporting unordered collections of variable-size records, index segments (e.g., for B-Trees) and *XML segments*. The *XML segments* for XML document collections are novel and described below.

3.1.2 Buffer manager

The buffer manager is responsible for transferring pages between main and secondary memory. It also synchronizes multithreaded access to the data pages using latches. Special calls exist to avoid I/O for reading or writing newly allocated or deallocated pages.

3.1.3 Page interpreters

While a page resides in main memory, it is associated with a *page interpreter object* that abstracts from the actual data format on the page. The page interpreters form a class hierarchy with a single base class, from which one or more data-type

specific classes are derived for each segment type. For example, a B-Tree segment might use one page interpreter class for inner pages and leaf pages each.

While the architectural decision to strictly separate intra-page data structure management (page interpreters) from inter-page data structures (segments) seems to be minor and straightforward, it is often not present in existing storage systems. As it turns out, the existence of a common base class and abstract data type interfaces for the page interpreters tremendously simplifies the implementation of the recovery subsystem, as described in a later section.

3.1.4 Partitions

Partitions represent an abstraction of random-access block devices. Currently, there exist partition classes for Unix files, raw disk access under Linux and Solaris, and C++ iostreams. In addition to the user segments, each partition contains several metadata segments describing the segments on the partition and the free space available.

3.2 XML storage

One of the core segment types in Natix is the novel XML storage segment, which manages a collection of XML documents. Before detailing the XML storage segment, we briefly survey existing approaches to store XML documents.

Flat streams. In this approach, the document trees are serialized into byte streams, for example by means of a markup language. For large streams, some mechanism is used to distribute the byte streams on disk pages. The mechanism can be a file system, or a BLOB manager in a DBMS [4,8,26]. This method is very fast when storing or retrieving whole documents or big continuous parts of documents. Accessing the documents' structure is only possible through parsing [1].

Metamodeling. A different method is to model and store the documents or data trees using some conventional DBMS and its data model [13,18,25,29,39–42].

In this case, the interaction with structured databases in the same DBMS is easy. On the other hand, reconstructing a whole document or parts of it is slower than in the previous method. Other representations require complex mapping operations to reproduce a textual representation [40], even duplicate elimination may be required [13].

Mixed. In general, the meta-modeling approach introduces additional layers in the DBMS between the logical data and the physical data storage, slowing the system down. Consequently, there are several attempts to merge the two "pure" methods above. In *redundancy-based approaches*, to get the best of both worlds, data is held in two redundant repositories, one flat and one metamodeled [47]. Updates are propagated either way, or only allowed in one of the repositories. This allows for fast retrieval, but leads to slow updates and incurs significant overhead for consistency control. In *hybrid approaches*, a certain level of detail of the data is configured as "threshold". Structures coarser than this granularity live in a structured part of the database, finer structures are stored in a "flat object" part of the database [6].

Natix native storage. Natix uses a novel **native** storage format with the following distinguishing features: (1) Subtrees of the original XML document are stored together in a single (physical) record (and, hence, are clustered); thereby; (2) the inner structure of the subtrees is retained; (3) To satisfy special application requirements, their clustering requirements can be specified by a *split matrix*. Performance impacts of different clusterings are evaluated in Sect. 3.3.

We now turn to the details on design and implementation of Natix' XML storage. We start with the logical document data model used by the XML segment to work with documents, and the storage format used by the XML page interpreters to work with document fragments that fit on a page. Then, we show how the XML segment type maps logical documents that are larger than a page to a set of document fragments possibly spread out on different disk pages. Finally, we elaborate on the maintenance algorithm for this storage format, explaining how to dynamically split records when trees grow, and how to tune the maintenance algorithm for special access patterns.

3.2.1 Logical data model

The XML segment's interface allows to access an unordered set of trees. New nodes can be inserted as children or siblings of existing nodes, and any node (including its induced subtree) can be removed. The individual documents are represented as ordered trees with non-leaf nodes labeled with a symbol taken from an alphabet Σ_{Tags}. Leaf nodes can, in addition to a symbol from Σ_{Tags}, be labeled with arbitrarily long strings over a different alphabet. Figure 2 shows an XML fragment and its associated tree.

3.2.2 Mapping between XML and the logical model

A small wrapper class is used to map the XML model with its node types and attributes to the simple tree model and vice versa. The wrapper uses a separate segment to map tag names and attribute names to integers, which are used as Σ_{Tags}. All the documents in one XML segment share the same mapping. The interface of this so-called *declaration table* allows for small, hashed per-thread caches for those parts of the mapping that are in use. The caches can be accessed very fast without any further synchronization.

Elements are mapped one-to-one to tree nodes of the logical data model. Attributes are mapped to child nodes of an additional *attribute container* child node, which is always the first child of the element node the attributes belong to. Attributes, PCDATA, CDATA nodes and comments are stored as leaf nodes. External entity references are expanded during import, while retaining the name of the referenced entity as a special internal node. Some integer values are reserved in addition to the ones for tag and attribute names, to indicate attribute containers, text nodes, processing instructions, comments and entity references.

```
<SPEECH>
<SPEAKER>OTHELLO</SPEAKER>
<LINE>Let me see your eyes;</LINE>
<LINE>Look in my face.</LINE>
</SPEECH>
```

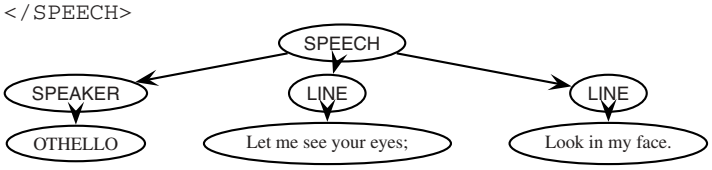


Fig. 2. A fragment of XML with its associated logical tree

3.2.3 XML page interpreter storage format

A (physical) record is a sequence of bytes stored on a single page. The logical data tree is partitioned into subtrees (see Sect. 3.2.4). Each subtree is stored in a single record and, hence, must fit on a page. Additionally to the subtree, a record contains a pointer to the record containing the parent node of the root node of its subtree (if it exists), and the identifier of the document the contained subtree belongs to.

Classified by their contents, there are three types of nodes in physical trees:

Aggregate nodes represent inner nodes of the logical tree.
Literal nodes represent leaf nodes of the logical tree and contain uninterpreted byte sequences like text strings, graphics, or audio/video sequences.
Proxy nodes are nodes which point to other records. They are used to link trees together that were partitioned into subtrees (see 3.2.4).

XML page interpreters are used to maintain the subtrees' records on data pages. They are based on a regular slotted page implementation, which maintains a collection of variable-length records on a data page. Each record is identified by a slot number which does not change even if the record is moved around on the page for space management reasons.

For the individual subtrees, distances between nodes have an upper limit, the page size. This raises opportunities to optimize the subtree representation inside the records. All structural pointers for intra-subtree relationships (parent and sibling pointers, node sizes etc.) fit into 2 bytes (if 64 kB pages are the maximum). We do not go into details about the exact representation used, as the maintenance algorithm described later does not depend on the details. The interested reader is referred to [24].

The currently used layout results in a node size for aggregate nodes of only 8 bytes, minimizing the overhead for storing the tree structure. Note that storing vanilla XML markup with only a 1-character tag name ($<\mathtt{X}>\ldots<\!/\mathtt{X}>$), for example, already needs 7 bytes! On average, XML documents inside Natix consume about as much space as XML documents stored as plain files in the file system.

3.2.4 XML segment mapping for large trees

Typical XML trees may not fit on a single disk page. Hence, document trees must be partitioned. Typical BLOB (*binary large object*) managers achieve this by splitting large objects at arbitrary byte positions [4,8,26]. We feel that this approach wastes the available structural information. Thus, we *semantically* split large documents based on their tree structure. We partition the tree into subtrees. *Proxy nodes* refer to connected subtrees not stored in the same record. They contain the RID (record identifier) of the record containing the subtree they represent. Substituting all proxies by their respective subtrees reconstructs the original data tree.

A sample is shown in Fig. 3. To store the given logical tree (which, say, does not fit on a page), the physical data tree is distributed over the three records r_1, r_2 and r_3. Two proxies (p_1 and p_2) are used in the top level record. Two helper aggregate nodes (h_1 and h_2) have been added to the physical tree. They group the children below p_1 and p_2 into a tree. Proxy and helper aggregate nodes are only needed to link together subtrees contained in different records.

Physical nodes drawn as dashed ovals like the proxies p_1, p_2 and the helper aggregates h_1, h_2, needed only for the representation of large data trees, are called *scaffolding nodes*. Nodes representing logical nodes (f_i) are called *facade nodes*. Only facade nodes are visible to the caller of the XML segment interface.

The sample physical tree is only one possibility to store the given logical tree. More possibilities exist since any edge of the logical tree can be represented by a proxy. The following section describes how to partition logical trees into subtrees fitting on a page.

3.2.5 Updating documents

We present Natix's algorithm for dynamic maintenance of physical trees. The principal problem addressed is that a record containing a subtree grows larger than a page. In this case, the subtree has to be partitioned into several subtrees, each fitting on a page. Scaffolding nodes (proxies and maybe aggregates) have to be introduced into the physical tree to link the new records together.

To describe our tree storage manager and split algorithm, it is useful to view the partitioned tree as an associative data structure for finding leaf nodes. We will first explain this metaphor and afterwards use it to detail our algorithm. Possible extensions to the basic algorithm and a flexible splitting mechanism to adapt it to special applications conclude this section.

Multiway tree representation of records. A data tree that has been distributed over several records can be viewed as a multiway tree with records as nodes. Each record contains a part of the logical data tree. In the example in Fig. 4, r_3 is blown up,

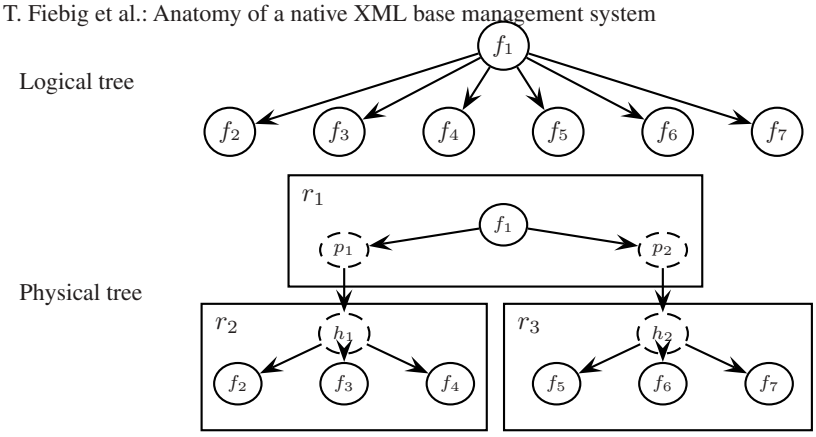


Fig. 3. One possibility for distribution of logical nodes onto records

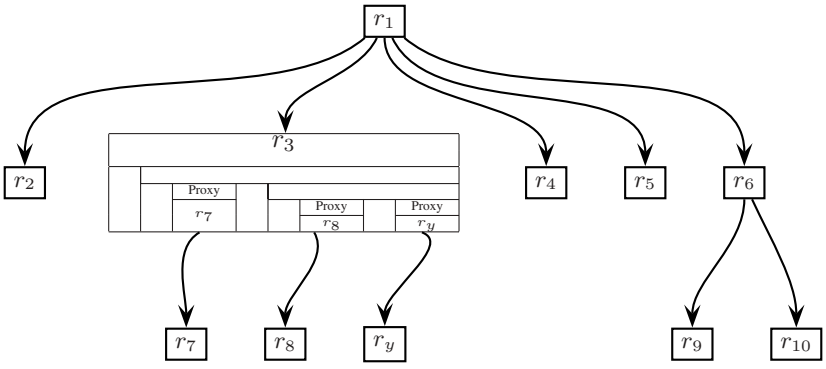


Fig. 4. Multiway tree representation of records

hinting at the flat representation of the subtree inside record r_3. The references to the child records are proxy nodes.

If viewed this way, our partitioned tree resembles a B-Tree-structure, as often used by traditional large object managers. The particularity is that the "keys" are not taken from a simple domain like integers or strings, but are based on structural features of the data tree. Nevertheless, this analogy gives us a familiar framework to describe the algorithms used to maintain the clustering of our records.

Algorithm for tree growth. Insertion into a Natix XML tree proceeds as follows. We determine the position where the new node has to be inserted, and if the designated page does not have sufficient space, the record is split. We explain the steps in detail:

1. Determining the insertion location. To insert a new node f_n into the logical data tree as a child node of f_1, it must be decided where in the physical tree the insert takes place. In the presence of scaffolding nodes, there may exist several alternatives, as shown by the dashed lines in Fig. 5: the new node f_n can be inserted into r_a, r_b, or r_c. In Natix, this choice is determined by the split matrix (see below).

2. Splitting a record. Having decided on the insertion location, it is possible that the designated record's disk page is full. First, the system tries to move the record to a page with more free space. If this is not possible because the record as such exceeds the net page capacity, the record is split by executing the following steps:

(a) Determining the separator. Suppose that in Fig. 5 we add f_n to r_b, which cannot grow. Hence, r_b must be split into at least two records r'_b and r''_b, and instead of p_b in the parent record r_a, we need a *separator* with proxies pointing to the new records to indicate where which part of the old record was moved.

In B-Trees, a median key partitioning the data elements into two subsets is chosen as separator. In our tree storage manager, the data in the records are not one-dimensional, but tree-structured. Our separators are paths from the subtree's root to a node d. The algorithm removes this path from the tree. The remaining forest of subtrees is distributed onto new records.

Figure 6 shows the subtree of one record just before a split. It is partitioned into a left partition L, a right partition R, and the separator S. This separator will be moved up to the parent record, where it indicates into which records the descendant nodes were moved as a result of the split operation.

The node d uniquely determines this partitioning (in the example, $d = f_7$): The separator $S = \{f_1, f_6\}$ consists of the nodes on the path from d to the subtree's root. Note that d is excluded. The subtree induced by d, the subtrees of d's right siblings, and all subtrees below nodes that are right siblings of nodes in S comprise the right partition (in the example, $R = \{f_7, f_8, \ldots, f_{14}\}$). The remaining nodes comprise the left partition (in the example, $L = f_2, \ldots, f_5$).

Hence, the split algorithm must find a node d, such that the resulting L and R are of roughly equal size. Actually, the desired ratio between the sizes of L and R is a configuration parameter (the *split target*), which can, for example, be set to achieve very small R partitions to prevent degeneration of the tree if insertion is mainly on the right side (as when creating a tree in pre-order from left to right). Another configuration parameter available for fine-tuning

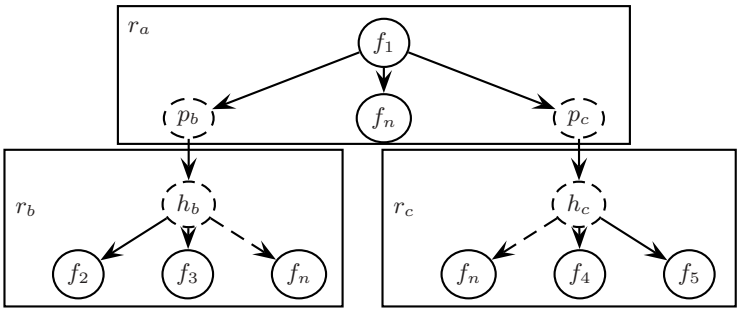


Fig. 5. Possibilities to insert a new node f_n into the physical tree

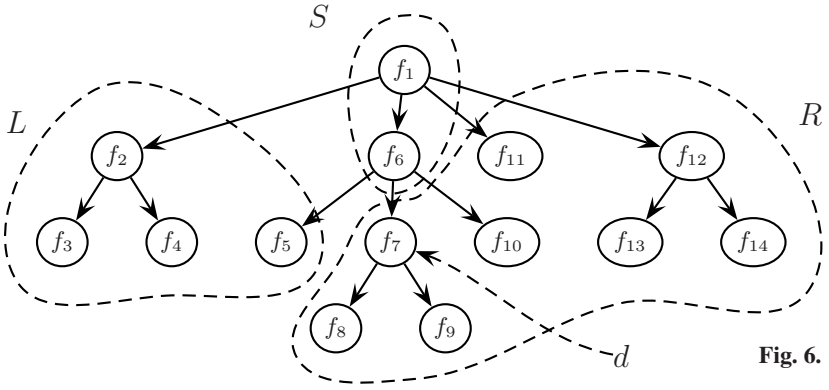


Fig. 6. A record's subtree before a split occurs

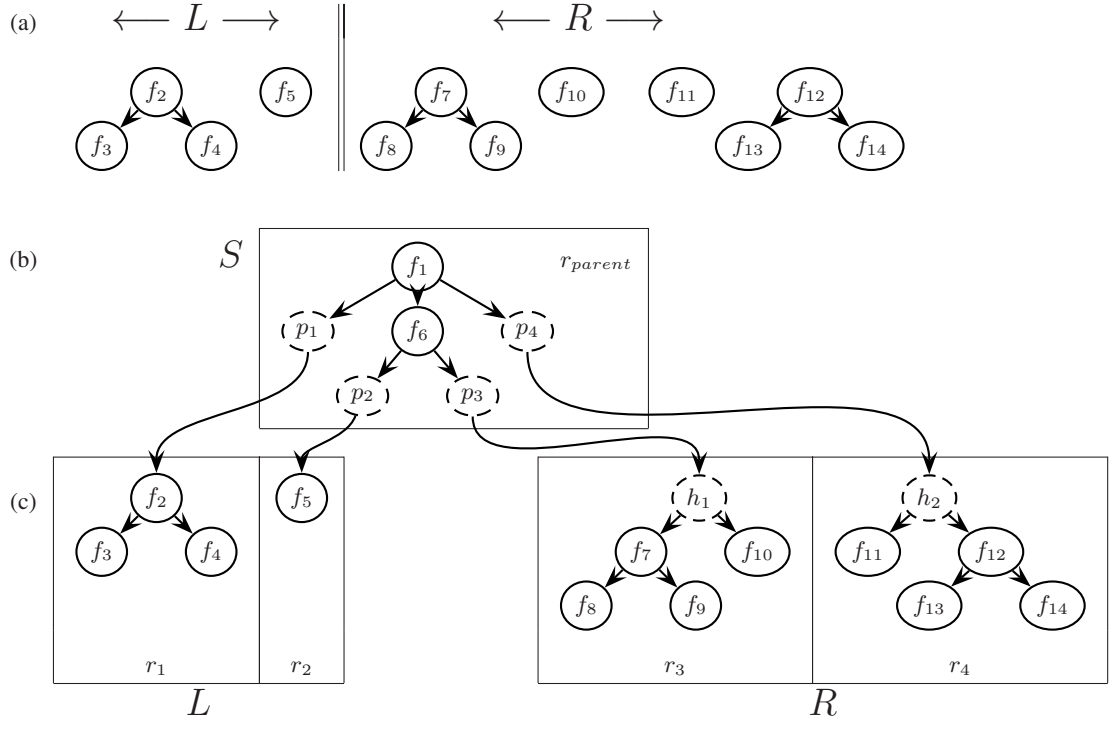


Fig. 7. Record assembly for the subtree from Fig. 6

is the *split tolerance*, which states how much the algorithm may deviate from this ratio. Essentially, the split tolerance specifies a minimum size for d's subtree. Subtrees smaller than this value are not split, but completely moved into one partition to prevent fragmentation.

To determine d, the algorithm starts at the subtree's root and recursively descends into the child whose subtree contains the physical "middle" (or the configured split target) of the record. It stops when it reaches a leaf or when the size of the subtree in which it is about to descend is smaller than allowed by the split tolerance parameter.

In the example in Fig. 6, the size of the subtree below f_7 was smaller than the split tolerance, otherwise the algorithm would have descended further and made $d = f_7$ part of the separator.

(b) Distributing the nodes onto records. Consider the partitioning implied by node $d = f_7$ (Fig. 6). The separator is removed from the old record's subtree, as in Fig. 7a. In the resulting forest of subtrees, root nodes in the same partition that were siblings in the original tree are grouped under one scaffolding aggregate. In Fig. 7c, this happened at nodes h_1 and h_2. Each resulting subtree is then stored in its own record. These new records $(r_1, \ldots, r_4)$ are called *partition records*.

(c) Inserting the separator. The separator is moved to the parent record – by recursively calling the insertion procedure – where it replaces the proxy which referred to the old, unsplit record. If there is no parent record, as in Fig. 7b, the separator becomes the new root record of the tree. The edges connected to the nodes in the partition records are replaced by proxies p_i. Since children with the same parent are grouped in one scaffolding aggregate, for each level of the separator a maximum of three nodes is needed, one proxy for the left partition record, one proxy for the right partition record, and one separator node.

To avoid unnecessary scaffolding records, the algorithm considers two special cases: First, if a partition record consists of just one proxy, the record is not created and the proxy is inserted directly into the separator. Second, if the root node of the separator is a scaffolding aggregate, it is disregarded, and the children of the separator root are inserted in the parent record instead.

3. Inserting the new node. Finally, the new node is inserted into its designated partition record.

The splitting process operates as if the new node had already been inserted into the old record's subtree, for two reasons. First, this ensures enough free space on the disk page of the new node's record. Second, this results in a preferable partitioning since it takes into account the space needed by the new node when determining the separator.

The split matrix. It is not always desirable to leave full control over data distribution to the algorithm. Special application requirements have to be considered. It should be possible to benefit from knowledge about the application's access patterns.

If parent-child navigation from one type of node to another type is frequent in an application, we want to prevent the split algorithm from storing them in separate records. In other contexts, we want certain kinds of subtrees to be always stored in a separate record, for example to collect some kinds of information in their own physical database area or to enhance concurrency.

To express preferences regarding the clustering of a node type with its parent node type, we introduce a *split matrix* as an additional parameter to our algorithm:

The split matrix S consists of elements $s_{ij}, i, j \in \Sigma_{\text{Tags}}$. The elements express the desired clustering behavior of a node x with label j as children of a node y with label i:

$$s_{ij} = \begin{cases} 0 & x \text{ is always kept as a standalone record and never clustered with } y \\ \infty & x \text{ is kept in the same record with } y \text{ as long as possible} \\ \mathbf{other} & \text{the algorithm may decide} \end{cases}$$

The algorithm as described above acts as if all elements of the split matrix were set to the value **other**.

It is easily modified to respect the split matrix. When moving the separator to the parent, all nodes x with label j under a parent y with label i are considered part of the separator if $s_{ij} = \infty$, and thus moved to the parent. If $s_{ij} = 0$, such nodes x are always created as a standalone object and a proxy is inserted into y. In this case, x is never moved into its parent as part of the separator, and treated like the root record for splitting purposes.

We also use the split matrix as the configuration parameter for determining the insertion location of a new node (see Sect. 3.2.5): when a new node x (label j) should be inserted as a child of node y (label i), then if $s_{ij} = \infty$, x is inserted into the same record y. If $s_{ij} = \mathbf{other}$, then the node is inserted on the same record as one of its designated siblings (wherever more free space exists). If $s_{ij} = 0$, x is stored as the root node of a new record and treated as described above.

The split matrix is an optional tuning parameter: it is not needed to store XML documents, it only provides a way to make certain access patterns of the application known to the storage manager. The "default" split matrix used when nothing else has been specified is the one with all entries set to the value **other**.

As a side effect, other approaches to store XML and semistructured data can be viewed as instances of our algorithm with a certain form of the split matrix [24].

3.3 Performance figures

We present some performance figures demonstrating the superiority of a clustered storage representation for query performance.

Crucial for query performance is the storage format's ability to support efficient navigation between document nodes. As a typical example, consider the evaluation of expressions in the XPath query language [9]. Navigation is at the core of evaluation plans for XPath queries. Since XPath is a declarative language, alternative evaluation plans exist. This allows us to investigate Natix' storage engine under different evaluation strategies.

Table 1 contains the elapsed evaluation times for different XPath expressions using Natix' query execution engine (detailed in Sect. 5), operating on two different storage formats. As a reference, the table also includes times for the Apache Project's XSLT Processor Xalan C++. Xalan operates on a main memory tree representation delivered by the Xerces XML Parser. The XPath evaluation times do not include the time needed for parsing in case of Xalan and system startup in case of Natix. These times are given separately at the bottom of Table 1. All times were measured on a PC with a 1-GHz AMD Athlon processor running Linux.

The first storage format uses the value **other** for all entries of the split matrix, the second 0 (see Sect. 3.2.5). The former clusters documents whereas the latter yields a storage format that stores one node per record. This is somewhat similar to mappings of XML documents to relational and object-oriented systems.

Three navigation-intensive XPath query expressions were evaluated. They are shown in Table 1 in a shorthand nota-

Table 1. Results for Xalan C++ and Natix storage (time in seconds)

			Fanout		
XPath	Method	Split Matrix	4	5	6
`desc`	Natix	$s_{ij} =$ **other**	0.0007	0.0010	0.0078
		$s_{ij} = 0$	0.0340	0.0903	0.2157
	Xalan C++		0.0029	0.0048	0.0097
`desc/desc`	Natix DupElim	$s_{ij} =$ **other**	0.0117	0.0339	0.0795
		$s_{ij} = 0$	0.1535	0.4499	1.0932
	Natix Pipe	$s_{ij} =$ **other**	0.0012	0.0086	0.0254
		$s_{ij} = 0$	0.0341	0.0921	0.2179
	Xalan C++		0.0099	0.0259	0.0630
`desc/fol`	Natix DupElim	$s_{ij} =$ **other**	1.4801	12.4536	77.9578
		$s_{ij} = 0$	20.8195	172.1488	993.7391
	Natix Pipe	$s_{ij} =$ **other**	0.0023	0.0135	0.0359
		$s_{ij} = 0$	0.0354	0.1020	0.2445
	Xalan C++		1.2779	11.4068	74.5812
`desc/fol/desc`	Natix DupElim	$s_{ij} =$ **other**	6.3942	53.4276	323.7528
		$s_{ij} = 0$	98.3362	826.3363	4862.5824
	Natix Push	$s_{ij} =$ **other**	1.4762	12.6416	78.7448
		$s_{ij} = 0$	21.0195	172.8896	995.3357
	Natix Pipe	$s_{ij} =$ **other**	0.0022	0.0139	0.0377
		$s_{ij} = 0$	0.0368	0.1035	0.2457
	Xalan C++		5.6159	49.9828	307.9632
Startup	Natix		0.0518	0.0524	0.0532
Startup/Parsing	Xalan C++		0.0980	0.1492	0.2578
Document Size			20 kB	58 kB	140 kB

tion, with `desc` meaning `descendant::test`, and `fol` meaning `following::test`.

Whenever applicable, we used three different query execution plans. All plans evaluate a path expression by a sequence of `UnnestMap` operators. They generate the result by performing nested loops for the XPath location steps (refer to Query Processing in Sect. 5 for details). Since such a straightforward evaluation sometimes produces duplicates and XPath requires results to be duplicate free, an additional duplicate elimination is added as the last operator of the plan whenever duplicates are produced. These plans are denoted by *DupElim* and are comparable to Xalan's evaluation strategy. Duplicates in intermediate results imply duplicate work by subsequent `UnnestMaps`. Plans denoted by *Push* eliminate duplicates as soon as they occur. Duplicate elimination is an expensive pipeline breaker. Hence, we developed a translation method of XPath expressions to sequences of `UnnestMaps` such that the programs by which they are parameterized guarantee that no duplicates are produced and, hence, no duplicate elimination is necessary. These plans are called *Pipe*. For XPath expressions with only a single axis, they coincide.

We evaluated the path expressions against three documents containing only element nodes with tag name `test`. All documents were trees of height 5, with a constant fanout at each inner node. We investigated three different fanouts.

Although there are many things that could be said about these numbers, we concentrate on the most important observations: (1) the clustered storage format ($s_{i,j} =$ **other**) is superior to the non-clustered storage format. The queries run faster on the clustered format by at least an order of magnitude, no matter how large the document is, no matter which query is answered, and no matter which execution plan is employed; (2) although Natix evaluates plans using a storage format for secondary memory whereas Xalan C++ uses a main memory representation for documents, the evaluation times of Xalan C++ and those of the comparable *DupElim* plans do not differ much for the clustered storage format. In most cases, Natix looses by less than 10%. For the single tree traversal needed to answer the `desc` query, Xalan is even slower than Natix' clustered format.

3.4 Index structures in Natix

In order to support query evaluation efficiently, we need powerful index structures. The main problem in building indexes for XBMSs is that ordinary full text indexes do not suffice, as we also want to consider the structure of the stored documents. Here we describe our approaches to integrate indexes for XML documents into Natix. We have followed two principle avenues of approach. On the one hand we enhanced a traditional full text index, namely inverted files, in such a way as to be able to cope with semistructured data. As will be shown, we opted for a versatile generic approach, InDocs (for Inverted Documents) [32], that can deal with a lot more than structural information. On the other hand we developed a novel index structure, called XASR (eXtended Access Support Relation) [17], for Natix.

3.4.1 Full text index framework

Inverted files are the index of choice in information retrieval [2,46]. Recently, the performance of inverted files improved considerably, mostly due to clever compression techniques. They usually store lists of document references to indicate in which documents search terms appear. Often offsets within a

document are also saved along with the references (this can be used to evaluate near-predicates, for example). However, in practice inverted files are handcrafted and tuned for special applications. Our goal is to generalize this concept by storing arbitrary *contexts* (not just offsets) without compromising performance.

The list implementation in Natix consists of four major components: the Index, the ListManager, the lists themselves and the ContextDescription. The main task of the Index is to map search terms to list identifiers and to store these mappings persistently. It also provides the main interface for the user to work with inverted files.

The ListManager maps the list identifiers to the actual lists, so it is responsible for managing the directory of the inverted file. We have implemented efficient methods for bulk load and bulk removal of lists, as well as for combining lists, namely union and intersection.

Each list is divided into fragments that fit on a page. All fragments that belong to one list are linked together and can be traversed sequentially. For performance reasons the content of each fragment is read and written sequentially.

The ContextDescription establishes the actual representation in which data is stored in a list. This includes not only which information is stored, but also determines the kind of compression used on the data. The strong point of our approach is the framework we provide. We can support new applications with little effort, as only the ContextDescription needs to be adapted. It is no problem at all to incorporate XML indexes based on inverted lists [14,31] into our framework. Among many other contexts, we have implemented a traditional context consisting of document IDs and offsets, as well as several contexts suited for XML data. They start with simple node contexts (including document IDs, node IDs, and offsets) and go up to complex node contexts that also consider structural information (e.g., d_{min} and d_{max} values as described in Sect. 3.4.2).

3.4.2 eXtended Access Support Relation

An extended access support relation (XASR) is an index that preserves the parent/child, ancestor/descendant, and preceding/following relationships among nodes. This is done by labeling the nodes of an XML document tree by depth-first traversal. We assign each node a d_{min} value (when we enter the node for the first time) and a d_{max} value (when we finally leave the node). For each node in the tree we store a row in an XASR table with information on d_{min}, d_{max}, the element tag, the document ID, and the d_{min} value of the parent node. Gaps can be used to reduce update costs [27].

A path in a query is translated into a sequence of self joins on the XASR table. For each location step in the path we have a join operation that connects the current context nodes to the relevant nodes of the next step. The XASR combined with a full text index provides a powerful method to search on (text) contents of nodes [17].

4 Transaction management

Enterprise-level data management is impossible without the transaction concept. The majority of advanced concepts for versioning, workflow and distributed processing depends on primitives based on the proven foundation of *atomic*, *durable* and *serializable* transactions.

Consequently, to be an effective tool for enterprise-level applications, Natix has to provide transaction management for XML documents with the above-mentioned properties. The transaction components supporting transaction-oriented programming in Natix are the subject of this section. The two areas covered are recovery and isolation, in this order.

For recovery, we adapt the ARIES protocol [35]. We further introduce the novel techniques of *subsidiary logging*, *annihilator undo*, and *selective redo* to exploit certain opportunities to improve logging and recovery performance which prove – although present in many environments – especially effective when large records with a variable structure are managed. This kind of record occurs when XML subtrees are clustered in records as in Natix' storage format.

For synchronization, an S2PL-based scheduler is introduced that provides lock modes and a protocol that are suitable for typical access patterns occurring for tree-structured documents. The main novelties are that: (1) granularity hierarchies of arbitrary depth are supported; and (2) contrary to existing tree locking protocols, jumps into the tree do not violate serializability.

4.1 Architecture

Fig. 8 depicts the components necessary to provide transaction management and their call relationships. Some of them are located in the storage engine and have already been described, while the rest is part of the transaction management module.

During system design, we paid special attention to a recovery architecture that treats separate issues (among them page-level recovery, logical undo, and metadata recovery) in separate classes and modules. Although this is not possible in many cases, we made an effort to separate the concepts as much as possible, to keep the system maintainable and extendible.

Although most components need to be extended to support recovery, in most cases this can be done by inheritance and by extension of base classes, allowing for the recovery-independent code to be separate from the recovery-related code of the storage manager.

4.2 Recovery components

We will not explain the ARIES protocol here, but concentrate on extensions and design issues related to Natix and XML. A description of ARIES can be found in the original ARIES paper [35] and in books on transaction processing (e.g., [21, 43]).

4.2.1 Log records

Natix writes a recovery log describing the actions of all update transactions using *log records*. Each log record is assigned a log-sequence number (LSN) that is monotonically increasing and can directly (without additional disk accesses) be mapped to the log record's location on disk.

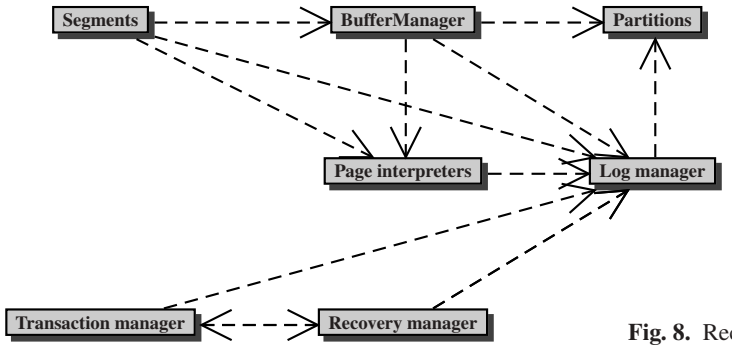


Fig. 8. Recovery components

Natix log records consist of the usual fields, including a transactionID, log record type and operation code information, flags that specify whether the record contains redo-only, media-recovery only, redo-undo, or undo-only information. In addition, log records include the ID of the updated object (segmentID, pageID, slot number, offset in a XML subtree) and, possibly, data to redo and/or undo the logged operation.

We come to the first important difference between ARIES and Natix' recovery protocol. The log records of a transaction are linked together into a pointer chain. In ARIES, a prevLSN pointer contains the LSN of the log record previously written by the same transaction. Natix does not use such a pointer. Instead, Natix' log records contain a nextUndoLSN pointer, which in standard ARIES is only contained in compensation log records (CLRs).

The nextUndoLSN of a log record points to the log record of the same transaction *that has to be undone* after this log record in case of a rollback. Usually, this will be the previously written log record with undo information of the same transaction. Redo-only log records do not participate in the nextUndoLSN chain, as the backward chaining is only necessary for undo processing.

Only in case of *compensation log records* which describe undo actions, the undoNextLSN points to the operation logged before the undone operation. Sect. 4.4 shows another situation where the nextUndoLSN chain of log records beneficially deviates from the reverse sequence of log records of one transaction.

4.2.2 Segments

From the view of the recovery subsystem, the segment classes comprise the main interaction layer between the storage subsystem and the application program. As part of their regular operations, application programs issue requests to modify or access the persistent data structures managed by the segments.

The segments map operations on their data structures – which can be larger than a page – to sequences of operations on single pages. The page interpreters deal with logging and recovery for operations on pages. This means that the code for multi-page data structures is the same for recoverable and nonrecoverable variants of the data structure, it only has to instantiate different page interpreter versions in the recoverable case. This is a significant improvement in terms of maintainability of the system, because less code is necessary.

The segments only handle logging and recovery for those update operations on multi-page data structures whose inverse operations are not described by the inverses of the respective page-level operations. We call them *L1 operations* (following [43]), while regular page-level operations are *L0 operations*. L1 operations occur for segment types where a high degree of concurrency is required (e.g., B-Trees [36]), and where other transactions may have modified the same structures while the original updater is still running (examples include index splits, where keys have to be moved *without* the split page being locked for the whole transaction duration).

Metadata is permanently accessed by the segments, and access to metadata needs to be highly concurrent. Therefore, L1 operations play a major role in the implementation of metadata and free space management. Issues in metadata recovery, as raised for example in [33, 34], are far from being simple but involve delicate dependencies. To keep the system simple and maintainable, they require an architecture prepared for them. Our framework for segment metadata recovery provides such an architecture, which can integrate solutions as described in the above-mentioned papers, as well as our own approaches. Details can be found in [23].

4.2.3 Page interpreters

The page interpreter classes are responsible for page-level logging and recovery. They create and process all page-level (i.e., the majority of) log records. The page-level log records use physical addressing of the affected page, logical addressing within the page, and logical specification of the performed update operation. This is called *physiological logging* [21].

For every page type in the page interpreter hierarchy that has to be recoverable, there exists a derived page interpreter class with an identical interface that, in addition to the regular update operations, logs all performed operations on the page and is able to interpret the records during redo and undo.

The page interpreter maintains the pageLSN attribute on the page and has a member attribute redoLSN containing the LSN of the first update operation after the last flush. The redoLSN is included in the buffer manager's checkpoint records (see below) and makes it possible to determine the start of the redo log scan after a crash.

4.2.4 Buffer manager

The buffer manager is controlling the transfer of pages between main and secondary memory. Although ARIES is independent of the replacement strategy used when caching pages [35], the buffer manager enables adherence to the ARIES protocol by notifying other components about page transfers between main and secondary memory and by logging information about the buffer contents during checkpoints.

A recovery issue related to the buffer manager is caused by the fact that it sometimes avoids I/O. It doesn't load pages which have been newly allocated, and it drops deallocated pages without writing them to disk. This introduces some complications, as redo has to deal with old or uninitialized page states on disk. This is not a problem of the buffer manager itself, but is taken care of by the segments' metadata management (see above).

4.2.5 Recovery manager

The recovery manager orchestrates system activity during undo processing, redo processing and checkpointing. It is stateless and serves as a collection of the recovery-related top-level algorithms for restart and transaction undo. During redo and undo, it performs log scans using the log manager (see below) and forwards the log records to the responsible objects (e.g., segments and page interpreters) for further processing.

Before forwarding each log record during undo, the recovery manager prepares the transaction control block. Flags and undoLSN pointers in the control block make sure that the log manager can properly chain log records together using the nextUndoLSN chain. This includes the special nextUndoLSN chaining necessary for L0 and L1 compensation log records, and L0 log records written as part of an inverse L1 operation.

4.2.6 Log manager

The log manager provides the routines to write and read log records, synchronizing access of several threads that create and access log records in parallel.

It keeps a part of the log buffered in main memory (using the Log Buffer as explained below) and employs special partitions, log partitions, to store log records.

The log manager (1) maintains the mapping of log records to LSN (and its inverse); (2) persistently stores the LSN of the most recent checkpoint; and (3) maintains the transaction's nextUndoLSN chain of log records using information from the transaction's control block. During undo, the recovery manager makes sure that the transaction control block contains proper values even for L0 and L1 compensation log records.

The automatic undoLSN chaining by the log manager allows for the segments and logging page interpreters to use regular forward processing methods to undo operations and write compensation log records, as the only difference between forward processing and undo is the different chaining of log records. As a result, the code for the logging page interpreters becomes much simpler, as no special functions for undo have to be coded.

The *log buffer* is part of the log manager and performs the transfer of log records from memory to disk and vice versa. Although recovery literature does not describe a detailed protocol how to access the log buffer and considers the problem trivial, the log buffer can easily become a bottleneck for update intensive transactions and access characteristics that are quite different from the regular buffer manager. Natix allows massively parallel log reading and log writing, several CPUs may simultaneously write even to the same log page.

4.2.7 Transaction manager

The transaction manager maintains the data structures for active transactions and is used by the application programs to group their operations into transactions. Each transaction is associated with a control block that includes recovery-related information like the LSN of the first log record, the LSN of the last written log record, an undoLSN field, and a pending actions list.

The LSN of the first log record is also considered a unique and persistent identifier for update transactions and is also called transactionLSN. The undoLSN field is used to hold the next record that requires undo, and is used by the log manager to chain log records together using the log records' nextUndoLSN fields. During forward processing, this field is set to the last log record written by the transaction that contained undo information. During undo processing, it is set to the nextUndoLSN field of the log record currently being undone to provide automatic nextUndoLSN chaining for CLRs.

The pending actions list contains a set of operations that have to be performed before the transaction commits. The pending actions list is a main memory structure and may not contain actions that are needed to undo the transaction (as it may be lost in a crash). Examples for its use include metadata recovery and subsidiary logging.

4.3 Subsidiary logging

Existing logging-based recovery systems follow the principle that every modification operation is immediately preceded or followed by the creation of a log record for that operation. An *operation* is a single update primitive (like insert, delete, modify a record or parts of a record). *Immediately* usually means before the operation returns to the caller. In the following, we explain how Natix reduces log size and increases concurrency, boosting overall performance, by relaxing both constraints.

Suppose a given record is updated multiple times by the same transaction. This occurs frequently when using a storage layout that clusters subtrees into records, for example, when a subtree is added node by node. It is desirable that a composite update operation is logged as one big operation, for example by logging the complete subtree insertion as one operation: merging the log records avoids the overhead of log record headers for each node (which can be as much as 100% for small nodes), and reduces the number of serialized calls to the log manager, increasing concurrency.

In the following, we sketch how Natix' recovery architecture supports such optimizations and elaborate on the concrete implementation in case of XML data.

4.3.1 Page-level subsidiary logging

In Natix, physiological logging is completely delegated to the page interpreters: how the page interpreters create and interpret log records is up to them.

Each page interpreter has its own state, which it can use to collect logging information for the associated page without actually transferring them to the log manager, thus keeping a private, *subsidiary log*. The interpreter may reorder, modify, or use some optimized representation for these private log entries before they are published to the log manager.

To retain recoverability, some rules have to be followed. To abide by the write-ahead-logging rule, the subsidiary log's content has to be published to the regular log manager before writing a page to disk. Likewise, all subsidiary log entries must be published to the regular log manager before the transaction commits, allowing them to be forced to disk to make the transaction durable.

When following these rules, the subsidiary logs become part of the log buffer as far as correctness of the recovery process is concerned. Although part of the log buffer is now stored in a different representation, its effects for undo and redo processing are the same. Basically, the rules cause a sequence of operations by one transaction on one page to be treated by logging and recovery as a single atomic update operation.

Natix' flexible architecture can incorporate these rules without change.

Since the buffer manager notifies page interpreters before their associated page is written to disk, the page interpreter is able to guarantee write-ahead-logging by transferring its subsidiary log to the log manager.

To force the subsidiary logs to disk before a commit occurs, all page interpreters that maintain a subsidiary log can add a pending action to the transaction control block (see Sect. 4.2.7) that is executed before the transaction commits, and that causes its subsidiary log to be published to the log manager.

Additional precautions have to be taken to guarantee proper recovery also in the presence of savepoints, which allow partial rollbacks. Integration of savepoints and L1 operations into subsidiary logs is described in [23].

4.3.2 XML-Page subsidiary logging

A typical update operation of Natix applications is the insertion of a subtree of nodes into a document, be it during initial document import or later while a document evolves. Often applications insert a subtree by inserting single nodes.

If every node insertion is logged using individual log records, every node will cause a log header to be written. Recall that an element node with no children and no literal data is stored using only 8 bytes of storage. A log header needs 32 bytes. For such a node the amount of log generated is five times as large as the actual data. With regard to update performance, this nullifies the effect of the compact storage format.

Since the updates are local and can easily be expressed in terms of a single insert operation into the record, logging this single operation allows for amortizing the costs for all the node insertions. To achieve this, conventional recovery systems would require the application to construct the subtree separately from the storage system and then add it with one insertion. Apart from requiring additional copying of data, the application would need to do some kind of dynamic memory management to maintain the intermediate representation. In addition, with page-level physiological logging, only merging of update operations that affect the same page is desirable. Applications would need to know about the mapping of the logical data structures to pages, violating encapsulation.

Using the page contents as the subsidiary log. The log entries for the subsidiary log are not explicitly stored. Instead, the XML page interpreters reuse the data page as a representation for log records before publishing them to the global log.

Using a flag called *fresh* in the node headers on the data page, new nodes/subtrees are marked. All information necessary to log the subtree insertion is available inside the data page itself, except for the transactionID. Instead of logging node insertions directly, the page interpreter only marks them as to-be-logged using the fresh flag.

Publishing the subsidiary log to the log manager then consists of a scan of the page's records. Every time a node is encountered that has the *fresh* flag set, a creation log record for the subtree implied by that node is written (and this subtree is skipped before further scanning the nodes of that record). The after image for this log record is the subtree as it is stored on the data page. The *fresh* flags are all cleared after publishing the subsidiary log.

Even if the fresh subtrees are modified before their creation is logged, no further maintenance of the subsidiary log is required: if a node is deleted, the *fresh* flag in its header is deleted as well, so no log record is written. If a node is modified, only the final version is included in the log record. Note that these situations occur frequently, e.g., when an XML editor is operating directly on the Natix API.

If non-fresh subtrees are modified, we have to be careful before directly creating non-subsidiary log records with the log manager. Since intra-record physical addressing is used in log records, they can only be redone and undone correctly if the data record is in the same state as it was when the operations were originally executed. Thus, we need to make sure that all modifications in the subsidiary logs are published *before* any nonsubsidiary log record for the same data record is created. Thus, the log is not only published as outlined in Sect. 4.3.1, but also when a node is modified that has its *fresh* flag not set.

To create complete log records from the subsidiary log, the page interpreter must know the transactionIDs that created the subtrees. The transactionIDs are not stored in the data page's contents.

This means that XML page interpreters must reserve some extra storage for the transactionIDs. Since we use record-level locking, only one transaction can have subsidiary log entries for any given record, so we only need one transactionID per record.

The transactionID is only necessary to maintain the subsidiary log and does not need to be stored on disk. Since it is unfavorable to add small objects dynamically just to store some transactionIDs, we limit the subsidiary log to a fixed number of transactionIDs. If more transactions want to add subsidiary log entries, we publish the subsidiary log to the log manager. On average, only a small number of records is stored on each page, as XML subtree records usually are quite large.

Therefore, allowing only one transaction per page interpreter to have subsidiary log entries is usually sufficient.

Effects of subsidiary logging. If a large document tree is created through repeated insertions of single nodes, splits occur frequently (see Sect. 3.2). A conventional logging approach would not only create bulky log records for every single node insertion, but would also log all of the split operations. The split log records are quite large, as they contain the contents of all partition log records. As a result, every node may be logged more than once.

With subsidiary logging, log records are only created when necessary for recoverability. If the newly created document(s) fit into the buffer, the log volume created is nearly equal to the size of the data, as only the "final" state of the document is logged upon commit. In addition, only a few log record headers are created (one for each subtree record), amortizing the logging overhead for the large number of small objects. Even if the whole document cannot reside in the buffer, subsidiary logging pays off by only creating log records as needed.

4.4 Annihilator undo

Transaction undo often wastes CPU resources, because more operations than necessary are executed to recreate the desired result of a rollback. For example, any update operations to a record that has been created by the same transaction need not be undone when the transaction is aborted, as the record is going to be deleted as a result of transaction rollback anyway. Refer to Fig. 9 which shows a transaction control block and log records and their nextUndoLSN chain. During undo, the records would be processed in the sequence $5, 4, 3, 2, 1$, starting from the undoLSN in the transaction control block and traversing the nextUndoLSN chain. Looking at the operations' semantics, undo of records 4 and 1 would be sufficient, as undo of 1 would delete record R1, implicitly undoing all changes to R1.

For our XML storage organization, creating a record and performing a series of updates to the contained subtree afterwards is a typical update pattern for which we want to avoid unnecessary overhead in case of undo. And since the abort probability of transactions can be much higher than in traditional database applications, undos occur more frequently. For example, online shoppers often fill their shopping carts and then decide not to go to the cashier.

Annihilators. We call undo operations that imply undo of other operations following them in the log *annihilators*. For example, the undo of a record creation like log record 1 in the example above is an annihilator, as it implies undo of all update operations applied to the record. For better performance, it is desirable to skip undo of operations implied by the annihilators. Natix realizes this to some extent.

Chaining annihilators. Let us recall from Sect. 4.2.1 that the nextUndoLSN pointer of every log record points to the previous operation of that transaction that requires undo, which is taken from the transaction control block's undoLSN field. Redo-only records are skipped by the nextUndoLSN chain.

If we know that undo for an operation is never required because an annihilator exists, as is the case when updating a subtree that has been created by the same transaction, the operation can be logged as a redo-only operation. This will prevent it from entering the nextUndoLSN chain of that transaction and it will not be undone explicitly, but implicitly by its annihilator. An additional advantage is that no undo information has to be included in the log record, which further reduces the amount of log generated.

The situation is slightly complicated by partial rollbacks. Partial rollbacks might want to reestablish an intermediate state of the transaction. In this case, undo information is required even if annihilators exist, because a partial rollback might not include the annihilator, and the updates must be rolled back explicitly.

XML annihilators. Let us now look at the way Natix exploits the optimization potential described above for XML data.

The XML page interpreters augment the stored information for the subtree as follows: In every XML subtree record header, an annihilatorLSN is stored containing the LSN of the last operation that logged a complete before image of the subtree. Usually, this is the creation LSN of the record (with the implicit "empty" before image), the annihilatorLSN is also set if for some other reason a log record with a full before image of the subtree is logged. The transactionID of the transaction which performed the annihilator is also written to the subtree record header.

The update operations for XML subtrees now check whether the stored annihilatorLSN for the subtree to be modified is greater than or equal to the last savepointLSN, and if the annihilator was performed by the same transaction. If yes, no rollback will be initiated that does not include the annihilator operation. Hence, the update operation can be logged redo-only and will be skipped during undo.

Figure 10 shows the resulting undo chain after log records 1–5 from the example in Fig. 9 have been written, assuming that no savepoint is taken. The annihilatorLSN for record R1 is the LSN of the creation record 1. Because of the annihilatorLSN checks during forward processing, the undo chain for the depicted transaction is now $4, 1$ – no unnecessary undos are performed.

This technique can be beneficial not only for freshly created records, but also if, for example, an application knows that rolling back to a certain state is likely, as may be the case for shopping cart applications in eCommerce shops that will rollback to an empty shopping cart when there are connection problems. Before every session, the application can explicitly announce major impending modifications to a subtree (the shopping cart), causing a before image to be written and the annihilatorLSN to be set. The state of the shopping cart before the session can easily be recreated by just one log record undo containing a complete before image, no matter how many single operations were executed in the meantime.

There are alternatives for the storage location of the annihilatorLSN, as reserving space for a whole LSN might be considered too high a cost for the benefits of annihilator undo. For example, it is possible to store the annihilatorLSN in main memory only, in the state of the page interpreter object. This

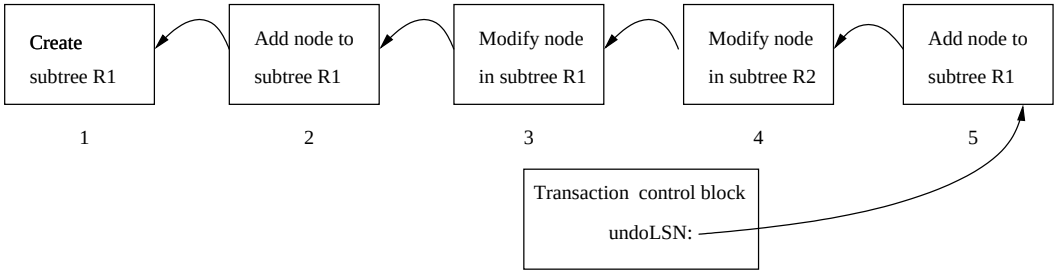


Fig. 9. Log records for an XML update transaction

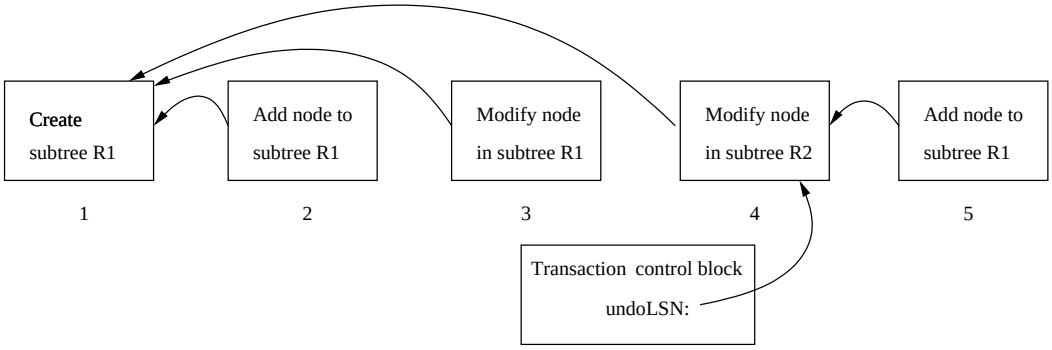


Fig. 10. Undo chaining with check for annihilators

would disallow annihilator undo if the page is kicked out of the buffer, which should be unlikely.

Please note that again, as with subsidiary logging explained in the previous section, the annihilatorLSN concept is local in its consequences for the system. It can be decided for every page interpreter class (i.e., data type) individually whether or not to support the annihilator undo concept, without affecting or modifying other parts of the system.

Other types of annihilators also exist. For example, if pages are deallocated, then it is not necessary to undo all record insertions to those pages first. However, it is dangerous to exploit this opportunity in the presence of record-level locking. Implementing page-level annihilators must be done with great care, as explained in [23].

4.5 Selective redo and selective undo

The ARIES protocol is designed around the redo-history paradigm, meaning that the complete state of the cached database is restored after a crash, including updates of loser transactions. After the redo pass has accomplished this, the following undo pass may unconditionally undo all changes of loser transactions in the log.

In the presence of fine-granularity locking, when multiple transactions may access the same page concurrently, the redo-history method is necessary for proper recovery, together with writing log records that describe actions taken during undo (compensation log records, or CLRs). Unfortunately, this may cause pages that only contain updates by loser transactions to be loaded and modified during restart, although their on-disk version (without updates) already reflects their desired state as far as restart recovery is concerned.

If a large buffer is employed and concurrent access to the same page by different transactions is rare, ARIES' restart performance is less than optimal, as it is likely that all uncommitted updates were only in the buffer at the time of the crash, and thus no redo and undo of loser transactions would be necessary.

In Natix, records used to store XML documents are typically large, so that each page only contains a few records, reducing the amount of concurrent access to pages. Since large buffers are also the rule, we would like to improve on the restart performance of our recovery system by avoiding redo (and undo) when possible.

There exists an extension to ARIES, called ARIES/RRH [37], that addresses this problem. Here, for pages that are updated with coarse-granularity locking, special flags are set in the log records. If during the redo pass log records of a loser transaction are encountered which have the flag set, the log record is ignored by the redo phase. During the latter undo phase, log records with the flag are only undone if the pageLSN indicates that the log record's update is really present on the page.

This procedure is complicated by the presence of CLRs. To facilitate media-recovery, undo operations are logged using a CLR, even if they have *not* actually been performed because the original operation was not redone in the first place. To allow to determine whether a CLR needs to be redone during restart or whether it is only necessary for media recovery, CLRs receive an additional field undoneLSN that contains the LSN of the log record whose undo caused the CLR to be written. Only if a page's pageLSN lies between the undoneLSN and the CLR's LSN, the CLR needs to be redone.

In Natix, we wanted to avoid increasing the log record header by another LSN-sized field, but we also wanted to benefit from avoidance of redo and undo when possible, without having to employ page-level locking at all times. Although there exists a relaxed version of ARIES/RRH that in some

situations allows selective redo and undo for fine-granularity locking, this requires an additional analysis scan of the log.

In the remainder of the section, we explain our extension of the method used by ARIES/RRH. For selective redo in an ARIES-based recovery environment to work, it is not really necessary that page-level locking is in effect for the affected pages during forward processing. Instead, it is sufficient that uncommitted updates of at most one transaction are present on affected dirty pages. This can be the case even without being enforced by page-level locking.

By adding a transactionID field to the main-memory page interpreter, it can easily be determined whether one or more transactions have updates on a dirty page. This information is included in the dirty page checkpoint log records, and as a result is available after restart analysis. Hence, it can be used during restart redo to avoid redo of loser updates on pages with only updates of one transaction.

We also avoid adding an undoneLSN field to log records. Suppose that during forward processing, we know the LSN of the current on-disk version of a dirty page (this can be derived from the redoLSN of the dirty page's frame control block). In this case, we can determine during forward processing if a CLR is required only for media recovery or if it may also have to be redone during restart:

If the page was *not* written between execution of the original operation and the undo operation, then the CLR is only necessary for media recovery, because either both the original operation and its inverse, or none of the two operations are contained in the disk version of the page. In both cases, the CLR will never be necessary for restart redo. We can set a flag in the log record header accordingly, which consumes much less space than a full-blown undoneLSN.

A drawback of this method is that certain knowledge about the on-disk state of a page is required during forward processing. This means that it is not allowed to asynchronously write a page. A common buffer write optimization is, for example, to protect a page with a latch only for the time necessary to make a private memory copy of the page, then to release the latch on the buffer frame, which allows operation on the page to continue. The write is performed in the background from the memory copy. In this case, the redoLSN cannot be used to determine which version of the page is on disk, as the asynchronous write may or may not have completed at the time. In [23] we show how to relax this condition.

4.6 Synchronization components

Since XML documents are semi-structured, we cannot apply synchronization mechanisms used in traditional, structured relational databases. XML's tree structure suggests using tree locking protocols as described e.g. in [3,43]. However, these protocols fail in the case of typical XML applications, as they expect a transaction to always lock nodes in a tree in a top-down fashion. Navigation in XML documents often involves jumps right into the tree by following an IDREF or an index entry. This jeopardizes serializability of traditional tree locking protocols. Another objection to tree locking protocols is the lack of lock escalation. Lock escalation is a proven remedy for reducing the number of locks held at a certain point in time. Tamino, a commercial product by Software, solves this problem by locking whole XML documents, limiting concurrency in an unsatisfactory manner. In order to achieve a high level of concurrency, one might consider locking at the level of XML nodes, but this results in a vast amount of locks. We strive for a balanced solution with a moderate number of locks while still preserving concurrent updates on a single document.

Although a traditional lock manager [21] supporting multi granularity locking (MGL) and strict two-phase locking (S2PL) can be used as a basis for the locking primitives in Natix, we need several modifications to guarantee the correct and efficient synchronization of XML data. In the remainder of this section we present our approach to synchronizing XML data. This involves an MGL hierarchy with an arbitrary number of levels and the handling of IDREF and index jumps into the tree. More information about the protocol and its implementation can be found in [38].

4.6.1 Lock protocol

Granularities. Locks can be requested at the segment, document, subtree and record level. The segment, document or record granularities are uniquely determined by a given XML node. This is different for the subtree granularity. There can be multiple subtrees containing a given node since the node is contained in a hierarchy of subtrees starting with the node itself as the smallest subtree up to the whole document as the largest subtree containing the node. This leads to an unconventional granularity hierarchy with an undefined number of subtree levels as shown in Fig. 11. Note that with the split matrix the user can enforce splitting a document into records such that those parts of the document that are likely to be modified concurrently reside in different records. As we will see, concurrent updates on different records are possible. Hence, a high level of customized concurrency is possible while avoiding an excessive amount of locks.

Lock modes and compatibility matrix. In addition to the lock modes described by Gray and Reuter [21], the lock manager provides a special shared parent pointer lock mode (SPP). We use this mode, which is described later on, to meet the requirements of supporting indexes and the ID/IDREF constructs. The full lock mode hierarchy and the corresponding compatibility matrix are shown in Fig. 12.

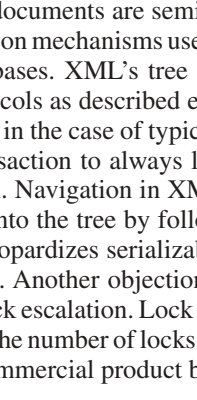
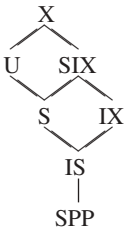


Fig. 11. Hierarchy of granularities



		lock held						
		S	X	U	IS	IX	SIX	SPP
re-quest-ed lock	S	+	−	−	+	−	−	+
	X	−	−	−	−	−	−	−
	U	+	−	−	+	−	−	+
	IS	+	−	−	+	+	+	+
	IX	−	−	−	+	+	−	+
	SIX	−	−	−	+	−	−	+
	SPP	+	−	−	+	+	+	+

Fig. 12. Hierarchy and compatibility matrix of lock modes

4.6.2 Special issues

Protocol. We distinguish two cases. For operations on non-structural data, we acquire a lock on the node containing the data. For structural changes we request a lock on each node in the vicinity of the affected node, i.e., on each node that has a direct pointer to the affected node. We (only) conceptually assume the existence of pointers between siblings and from a parent node to its first and last child in an XML tree. Thus, the siblings of the affected node are locked, and in the special case of terminal nodes the parent node is locked. Strictly speaking, these pointers do not really exist in the physical representation of XML in Natix, but they should be seen as an auxiliary, logical construct. In addition, note that in view of having records at the finest granularity level, we cannot lock a node directly. Instead we have to lock the record containing the nodes we wish to lock.

Top-down navigation. When an application traverses a document from the root to the leaves, we are back to the traditional tree locking case. The transaction holds (some kind of) locks on all nodes along the path from the root to the current node. Therefore, no other transaction can change the structure of the tree along this path.

Jumps. Sometimes transactions access an arbitrary node within the tree, e.g., by dereferencing an IDREF link. As we do not necessarily have locks on all ancestor nodes of the accessed node, this may lead to complications with other transactions working higher up in the subtree.

The goal is to lock all nodes from the node N the transaction jumped to up to the root node of the document with the according intention lock[1]. It is important to note that the nodes on this path are not known to the transaction. They have to be discovered by traversing upwards towards the root. This is very dangerous, since in an extreme case a whole subtree containing N could have been deleted by another transaction, nodes on the path to the root may have been moved to another disk page and so on. To guarantee serializability, we must carefully traverse up the tree. While traversing up the path to the root, the transaction acquires SPP locks. Other transactions that move records around or split records, have to adjust parent pointers and acquire X locks on records whose parent pointer they change. Since SPP locks are incompatible with X locks, we can only traverse up if no other transaction performed changes to the document that would endanger serializability. Once the transaction reaches the root, it traverses down the path from the root to node N, thereby converting the SPP locks to the required lock mode. Note that the SPP locks prevent other transactions from interfering while walking down.

Lock escalation. We invoke lock escalation whenever a transaction holds an excessive number of locks causing a lot of overhead. This is checked when the transaction requests a new lock. A heuristic is used to decide which locks to escalate. If after the escalation of node locks to document locks there are still too many locks, whole segments are locked.

Deadlock detection. If a transaction has waited for a lock request longer than a specified timeout value, we start a deadlock detection by searching for cycles in the waiting graph, starting at the node of the current transaction.

[1] If the transaction already owns an intention lock for a node M on this path, it is sufficient to acquire the locks on the path from N to M.

5 Natix query execution engine

A query is typically processed in two steps: the query compiler translates a query into an optimized query evaluation plan, and then the query execution engine interprets the plan. We describe Natix' query execution engine. The query compiler is beyond the scope of this paper.

5.1 Overview

While designing the Natix Query Execution Engine (NQE) we had three design goals in mind: efficiency, expressiveness, and flexibility. Of course, we wanted our query execution engine to be efficient. For example, special measures are taken to avoid unnecessary copying. Expressiveness means that the query execution engine is able to execute all queries expressible in a typical XML query language like XQuery [5]. Flexibility means that the algebraic operators implemented in the Natix Physical Algebra (NPA) – the first major component of NQE – are powerful and versatile. This is necessary in order to keep the number of operators as small as possible. Let us illustrate this point by an example. The result of a query can be an XML document or fragment. However, there exist several alternatives to represent an XML document. First, a textual representation is possible. Then the result of the query is a simple – though possibly long – string. If further processing of the query result is necessary, e.g., by a stylesheet processor, then a DOM [22] representation makes sense. A third alternative is to represent the query result as a stream of SAX [30] events. Since we did not want to implement different algebraic operators to perform the implied different result constructions, we needed a way to parameterize our algebraic operators in a very flexible way. The component to provide this flexibility is the Natix Virtual Machine (NVM) – the second major component of NQE.

Let us now give a rough picture of NPA and NVM. The Natix Physical Algebra (NPA) works on sequences of tuples. A tuple consists of a sequence of attribute values. Each value can be a number, a string, or a node handle. A node handle can be a handle to any XML node type, e.g., a text node, an element node or an attribute node.

NPA operators are implemented as *iterators* [20]. All NPA operators inherit from an iterator superclass which provides the `open`, `next`, and `close` interface. However, this classical interface of an iterator has been extended by splitting the `open` call into three distinct calls:

create Performs context independent resource allocations and initializations.
initialize Performs context dependent resource allocations and initializations.
start Prepares to fetch the first tuple.

Accordingly, the `close` call has been split into `finish`, `deinitialize`, and `destroy`. The reason for this split is the efficient support of nested algebraic expressions, which are used for example to represent nested queries that for efficiency or other reasons are not unnested by the query compiler.

NPA operators usually take several parameters which are passed to the constructor. The most important parameters are programs for the Natix Virtual Machine (NVM). Take for example the classical `Select` operator. Its predicate is expressed as an NVM program. The `Map` operator takes as parameter a program that computes a function and stores the result in some attribute. Other operators may take more than one program. For example, a typical algebraic operator used for result construction takes three NVM programs. We considered several alternatives to NVM programs to represent expressions. Operator trees turned out to be too expensive. Further experiments showed that compilation into machine code boosts performance sometimes by a factor of almost two. However, we decided against it due to its implementation complexity and machine dependence. Instead we decided to use a virtual machine as the middle course. We did not use an existing virtual machine like the Java Virtual Machine (JVM) for several reasons. First, we would have to extend an existing JVM to incorporate special instructions used for query and XML processing. Second, the JVM opcode is of variable length decreasing performance since opcode interpretation becomes more expensive. Third, memory management as performed by the JVM does not fit our memory management model.

The rest of the section is organized as follows. We first introduce the Natix Virtual Machine. Then we describe the Natix Physical Algebra. Last, we give some example plans.

5.2 Natix virtual machine

The Natix Virtual Machine interprets commands on register sets. Each register set is capable of holding a tuple, e.g., one register holds one attribute value. At any time, an NVM program is able to access several register sets. The situation is illustrated in Fig. 13 for unary (a) and binary (b) NPA operators. There always exists a global register set (named X) which contains information that is global to but specific for the current plan execution. It contains information about partitions, segments, lookup tables, and the like. It is also used for intermediate results or to pass information down for nested query execution. Between operators, the tuples are stored in the register sets Z and Y where Y is only available for binary operators. In case of a join operator, Z contains an outer and Y an inner tuple.

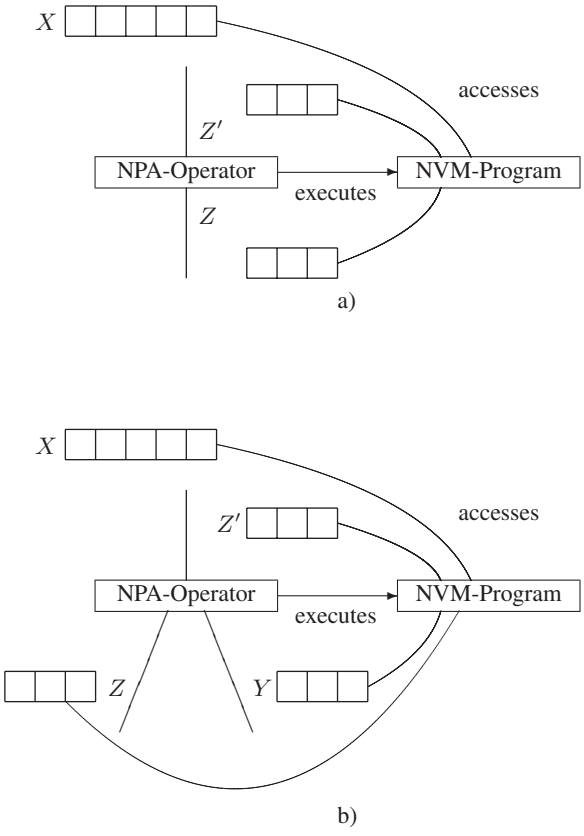


Fig. 13. Register sets, NPA-operators and NVM-programs

It is database lore that during query execution most of the time is spent on copying data around. We have been very careful to avoid unnecessary copying in NQE. Let us briefly describe our approach here. In order to avoid unnecessary copying, pointers to registers sets are passed among different NPA operators. If there is no pipeline breaker in a plan, only one Z register set is allocated and its address is passed down the tree. The general rule for Z registers used by pipelined operators is the following: memory is passed from top to bottom and content from bottom to top.

The situation differs for pipeline breakers. A pipeline breaker usually allocates register sets. For example, a simple grouping operator may allocate a Z register set for every group and fill it with corresponding values while processing its input. When `next` is called on the group operator, memory is passed to the group operator by providing the `next` call with a pointer to a Z register set. It would be straightforward to copy the contents of the next local Z register of the group operator into the Z register passed down by the `next` call. However, this is unnecessary. Instead, a pointer to the local Z register of the group operator is passed upwards to the caller of `next`. Hence, the `next` method contains a parameter which holds a reference to a pointer to a register set. This way, the operator has a choice to either modify the contents of the register set or to return a different register set.

NVM commands. NVM commands can be divided into groups. For example, there exists a group for arithmetics. A

typical command is ARITH_ADD_A_UI4_ZZZ which adds two unsigned four byte integers found in Z registers and puts the result into another Z register. In general, a command name starts with a group name followed by the command. Then an optional result mode (borrowed from AVM, see [44]) and a type follow. Last in the command name is a specification of the register sets for the arguments and the result. Altogether there are more than 1500 commands that can be interpreted by the NVM. Let us consider a small example of a program that adds two numbers given in X registers 1 and 2. The following program adds these numbers, puts the result in X register 3 and prints the output:

```
ARITH_ADD_A_SI4_XXX  1  2  3
PRINT_SI4_X          3
STOP
```

The STOP command ends the execution of the NVM. Besides STOP, NVM provides more control commands like EXIT_FALSE_X which exits if a specified X register contains false. The following small program implements the selection predicate $a \leq 55$ for some variable a where we assume that a is contained in Z register number 1.

```
CMP_LEQ_SI4_ZCX  1  55  2
EXIT_FALSE_X     2
```

The C indicates that the corresponding argument is a constant.

The XML specific part of NVM contains about 150 commands. Among these are simple operations that copy a node handle from one register to another, compare two handles, print the XML fragment rooted at a handle with or without markup and the like. The main portion of the XML specific commands consists of navigation operations roughly corresponding to the axes in XPath. These commands retrieve the attributes of an element node, its children, or its descendants.

Let us consider an example. Evaluating XPath expression can sometimes be performed by a sequence of UnnestMap operations. An UnnestMap operation takes in its logical form a set-valued expression and produces a single output tuple for every element in the result of this expression. At the physical level, an UnnestMap operation takes three programs. The first program initializes the first tuple to be returned. The second program computes the next tuples. After all tuples have been produced, the third program is called for cleanup operations. The following table contains the three programs for an UnnestMap operator that accesses all child nodes of a node contained in Z register 1. The children are written to Z register 2. X Register 3 is used to indicate whether there is a new tuple or the iteration ends. X Register 4 is used to save the current child node since it will not necessarily survive in the Z register between two next calls.

init	XML_CHILD_ZZ	1	2
	XML_VALID_ZX	2	3
	EXIT_FALSE_X	3	
	MV_XML_ZX	2	4
step	XML_SIBLING_NEXT_XX	4	4
	XML_VALID_XX	4	3
	EXIT_FALSE_X	3	
	MV_XML_XZ	4	2
fin			

In the init program, we look for the first child. Subsequently, we check it for validity. If there are no children, Z register 2 will contain an invalid node. Analogously, after retrieving the next child with the step program's first command, we check whether there has been a next child. No fin program is necessary here. Note that we omitted the STOP commands.

These kinds of programs are generated by the query compiler. We use the query compiler BD 2 for this purpose. BD 2 is a multilingual query compiler speaking several query languages. Its description is beyond the scope of the paper. Since NVM programs are a little difficult to read for human readers, we will use expressions with function calls instead in the plans of the next section.

Implementation of the NVM. All commands are represented by consecutive non-negative integers. The NVM interpreter is implemented as an infinite loop with a switch statement inside. Only control commands may halt the execution by jumping outside the loop. The advantage of this implementation of NVM is that no function calls are necessary to execute a command. This makes NVM program execution very fast.

5.3 Natix physical algebra

Query languages for XML (for example XQuery) often provide a three-step approach to query specification. The first part (let and for in XQuery) specifies the generation of variable bindings. The second part (where in XQuery) specifies how these bindings are to be combined and which combinations are to be selected for the result construction. The final part (return in XQuery) specifies how a sequence of XML fragments is to be generated from the combined and selected variable bindings.

Reflecting this three-step approach, NPA operators exist to support each of these steps. The middle step – binding combination and selection – can be performed by standard algebraic operators borrowed from the relational context. Those provided in NPA are a select, map, several join and grouping operations, and a sort operator. Some operators like the d-join and the unary and binary grouping operators are borrowed from the object-oriented context [10,11]. Since these operators and their implementations are well-known (see e.g., [20]), we concentrate on the XML specific operations for variable binding generation and XML result construction.

At the bottom of every plan are scan operations. NPA provides several of them. The simplest is an expression scan (ExpressionScan) which generates tuples by evaluating a given expression. It can be thought of as a Map operator working without any input. It is used to generate a single tuple containing the root of a document identified by its name. The second scan operator scans a collection of documents and provides for every document a tuple containing its root. Index scans complement the collection of scan operations.

Besides the scan operations UnnestMap is used to generate variable bindings for XPath expressions. An XPath expression can be translated into a sequence of UnnestMap operations. Consider for example the XPath expression /a//b/c. It can be translated into

UnnestMap$_{\$4=child(\$3,c)}$(

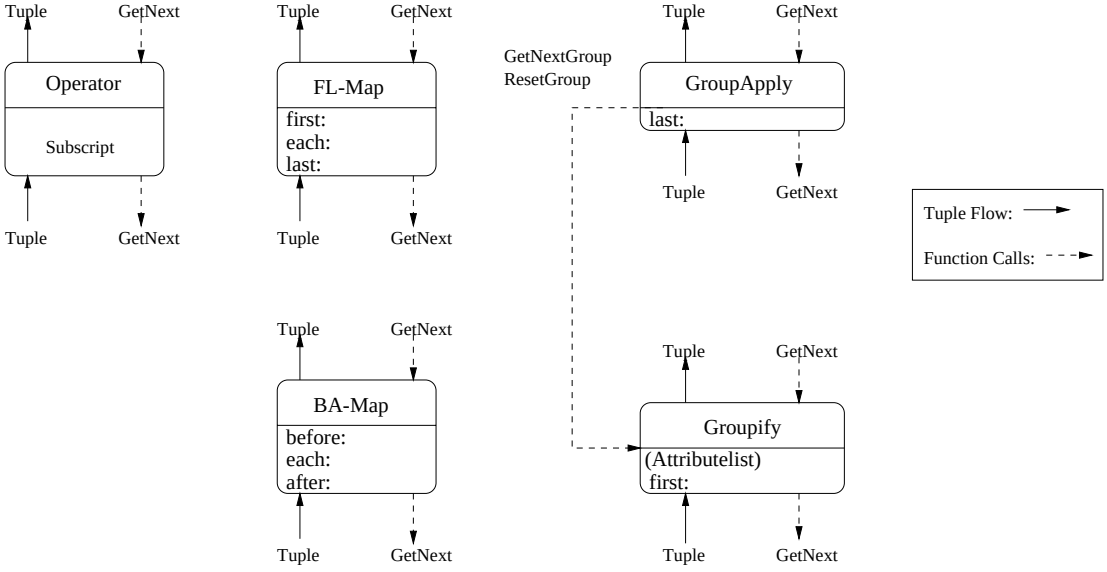


Fig. 14. Interfaces of construction operators

```
<!ELEMENT bib (conference|journal)*>
<!ELEMENT conference (title, year, article+)>
<!ELEMENT journal (title, volume, no?, article+)>
<!ELEMENT article (title, author+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author EMPTY>
<!ATTLIST author last CDATA #REQUIRED
                 first CDATA #REQUIRED>
```

Fig. 15. Sample DTD

$$\text{UnnestMap}_{\$3=desc(\$2,b)}(\text{UnnestMap}_{\$2=child(\$1,a)}([\$1]))$$

However, one has to be careful: not all XPath expressions can be translated straightforwardly into a sequence of `UnnestMap` operations. Often relatively complex transformations are needed to guarantee a correct and efficient pipelined evaluation of XPath expressions. The details thereof are beyond the scope of the paper.

For XML result construction NPA provides the `BA-Map`, `FL-Map`, `Groupify-GroupApply`, and `NGroupify-NGroupApply` operators. The interfaces of these operators are shown in Fig. 14. The `BA-Map` and `FL-Map` operators are simple enhancements of the traditional Map operator. They take three NVM programs as parameters. The program called `each` is called on every input tuple. The programs `before` and `after` of the `BA-Map` operator are called before the first and after the last tuple, respectively. The programs `first` and `last` of the `FL-Map` operator are called on the first and last tuple, respectively. In general `BA-Map` is more efficient (`FL-Map` needs to buffer the current tuple) and should be used whenever applicable.

The `Groupify` and `GroupApply` pair of operators detects group boundaries and executes a subplan contained between them for every group. The `Groupify` operator has a set of attributes as parameters. These attributes are used to detect groups of tuples. On every first tuple of a group the program `first` is executed. Whenever one attribute's value changes, it signals an end of stream by returning `false` on the `next` call. The `GroupApply` operator then applies the `last` program on the last tuple of the group. It then asks the `Groupify` operator to return the tuples of the next group by calling `GetNextGroup`. `ResetGroup` allows to reread a group. The use of these operators will become more clear when looking at the examples of the next section. The `NGroupify` and `NGroupApply` pair of operators allows multiple subplans to occur between them. They are rather complex and beyond the scope of the current paper. More details about the operators and the generation and optimization of construction plans can be found in [15,16].

5.4 Example plans

Let us consider two plans which rely on a bibliography document whose DTD is shown in Fig. 15. The first plan implements the evaluation strategy for the following XQuery (Query 1):

```
<result>
{
 FOR $c IN document("bib.xml")/bib/conference
 WHERE $c/year > 1996
 RETURN
    <conference>
      <title> { $c/title } </title>
      <year> { $c/year } </year>
    </conference>
}
</result>
```

This query retrieves the title and year for all recent conferences. The corresponding plan is shown in Fig. 16. Note that

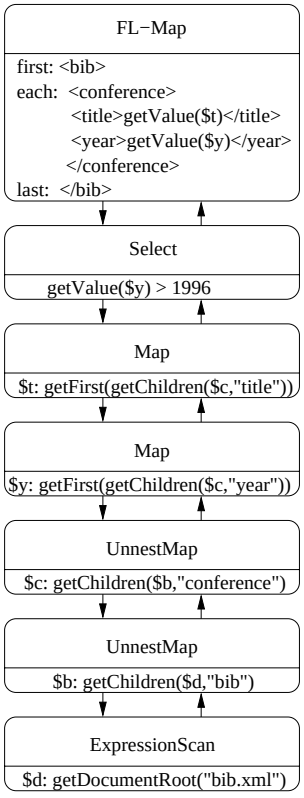


Fig. 16. Construction plan of Query 1

this plan is unoptimized and results from a rather straightforward translation process of the query into the algebra. The bottom-most operator is an `ExpressionScan`. It evaluates its expression to build a single tuple whose attribute $d is set to the root of the document `bib.xml`. Then a sequence of `UnnestMap` operations follows to access the `bib`, `conference`, `year`, and `title` elements. In case an axis returns only a single element, `Map` and `UnnestMap` operations are interchangeable. After producing all variable bindings, the selection predicate is applied. Last, the result is constructed by a single `FL-Map` operator. This is the usual situation for a query that selects and projects information from an XML document without restructuring it.

The second query restructures the original bibliography document such that papers are (re-) grouped by authors (Query 2):

```
<bib>
{
  FOR $a IN document("bib.xml")
             //conference/article/author
  RETURN
   <author>
    <name first={$a/@first} last={$a/@last}/>
    <articles>
    {
          FOR $b IN document("bib.xml")
                     //conference/article,
              $c IN $b/author
          WHERE $c/@first = $a/@first
                 AND $c/@last=$a/@last
          RETURN
```

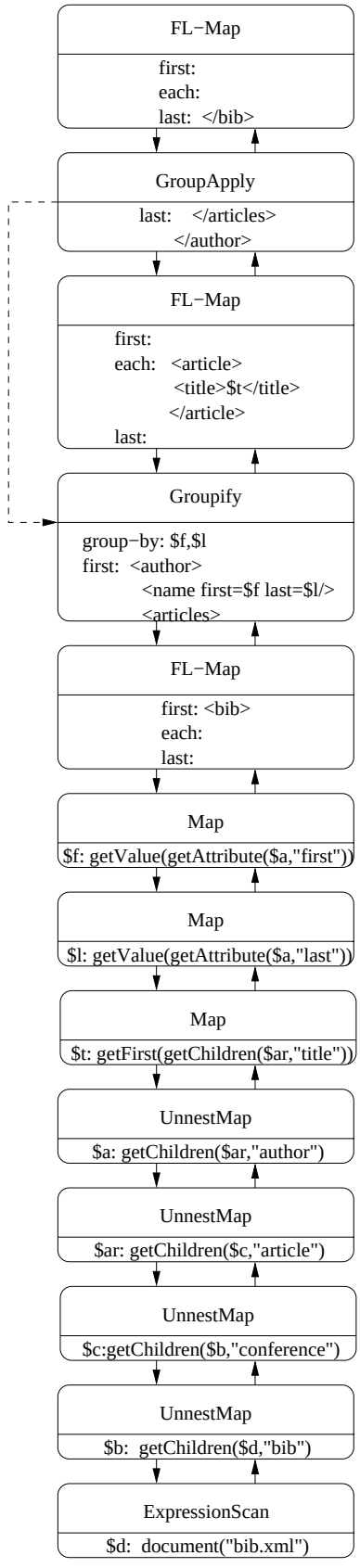


Fig. 17. Construction plan of Query 2

```
            <article> {$b/title} </article>
        }
        </articles>
    </author>
}
</bib>
```

The corresponding plan is shown in Fig. 17. The lower half of the plan produces the variable bindings necessary to answer the query. The outer two `FL-Map` operations produce the outermost <bib> and </bib> tags. Since they print constants, they can be replaced by `BA-Map` operations, but again we show an unoptimized initial plan. The `Groupify` operation groups the input relation by the first and last name of the authors. For every such group, the inner `FL-Map` operator prints the title of the current group's author. The `author` and `article` open tags are printed by the `first` program of `Groupify`. The corresponding close tags are produced by `GroupApply`.

6 Conclusion

Exemplified by storage management, recovery, multi-user synchronization, and query processing, we illustrated that the challenges of adapting database management systems to handling XML are not limited to schema design for relational database management systems.

We believe that sooner or later a paradigm shift in the way XML documents are processed will take place. As the usage of XML and its storage in DBMSs spreads further, applications working on huge XML document collections will be the rule. These applications will reach the limits of XML-enhanced traditional DBMSs with regard to performance and application development effectiveness. Our contribution is to prepare for the shift in processing XML documents by describing how efficient, native XML base management systems can actually be built.

Acknowledgements. The authors thank Simone Seeger for her help in preparing the manuscript. Soeren Goeckel, Andreas Gruenhagen, Alexander Hollmann, Oliver Moers, Thomas Neumann, Frank Ueltzhoeffer, and Norman May provided invaluable assistance in implementing the system. We also thank the anonymous referees for their comments and suggestions.

References

1. S. Abiteboul, S. Cluet, T. Milo (1993) Querying and updating the file. In: Very large data bases, VLDB '93: Proc. 19th International Conference on Very Large Data Bases, pp 73–84, Dublin, Ireland, August
2. R. Baeza-Yates, B. Ribeiro-Neto (1999) Modern information retrieval. Addison-Wesley, Reading, Mass., USA
3. P. Bernstein, V. Hadzilacos, N. Goodman (1987) Concurrency control and recovery in database systems. Addison-Wesley, Reading, Mass., USA
4. A. Biliris (1992) An efficient database storage structure for large dynamic objects. In: Proc. International Conference on Data Engineering, pp 301–308
5. S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, J. Siméon, M. Stefanescu (2002) XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, 2002. W3C Working Draft 30 April
6. K. Böhm, K. Aberer, E.J. Neuhold, X. Yang (1997) Structured document storage and refined declarative and navigational access mechanisms in HyperStorM. VLDB J. 6(4):296–311
7. T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler (2000) Extensible markup language (xml) 1.0 (2nd edn). Technical report, World Wide Web Consortium (W3C)
8. M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita (1986) Object and file management in the EXODUS extensible database system. In: Proc. 12th International Conference on Very Large Data Bases, pp 91–100, Los Altos, Calif., USA
9. J. Clark, S. DeRose (1999) XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C)
10. S. Cluet, G. Moerkotte (1993) Nested queries in object bases. In: Proc. Int. Workshop on Database Programming Languages
11. S. Cluet, G. Moerkotte (1995) Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen
12. data ex machina (2001) NatixFS technology demonstration available at: http://www.data-ex-machina.de/download.html
13. A. Deutsch, M. Fernandez, D. Suciu (1999) Storing semistructured data with STORED. In: Proc. 1999 ACM SIGMOD International Conference on Management of Data, pp 431–442, Philadelphia, Penn., USA, June
14. J. Naughton, et al (2001) The Niagara internet query system. IEEE Data Eng Bull 24(2):27–33
15. T. Fiebig, G. Moerkotte (2001) Algebraic XML construction and its optimization in Natix. WWW J 4(3):167–187
16. T. Fiebig, G. Moerkotte (2001) Algebraic XML construction in Natix. In: Proc. 2nd International Conference on Web Information Systems Engineering (WISE'01), pp 212–221, Kyoto, Japan, December. IEEE Computer, New York
17. T. Fiebig, G. Moerkotte (2001) Evaluating queries on structure with extended access support relations. In: The World Wide Web and Databases, 3rd International Workshop WebDB 2000, Dallas, Tex., USA, May 18–19, 2000, Selected Papers, Lecture Notes in Computer Science, vol. 1997. Springer, Berlin Heidelberg New York
18. D. Florescu, D. Kossmann (1999) Storing and querying xml data using an rdmbs. IEEE Data Eng Bull 22(3):27–34
19. Y. Goland, E. Whitehead, A. Faizi, S. Carter, D. Jensen (1999) Http extensions for distributed authoring – webdav. Technical Report RFC2518, Internet Engineering Task Force, February
20. G. Graefe (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):73–170
21. J. Gray, A. Reuter (2000) Transaction processing: concepts and techniques. Morgan Kaufmann, San Francisco
22. A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne (2000) Document object model (DOM) level 2 core specification. Technical report, World Wide Web Consortium (W3C)
23. C.-C. Kanne (2002) Natix: a native XML base management system. PhD thesis, University of Mannheim (to appear)
24. C.-C. Kanne, G. Moerkotte (1999) Efficient storage of XML data. Technical Report Nr. 8, Lehrstuhl für praktische Informatik III, Universität Mannheim, June
25. M. Klettke, H. Meyer (2000) XML and object-relational database systems – enhancing structural mappings based on statistics. In: ACM SIGMOD Workshop on the Web and Databases (WebDB)

26. T.J. Lehman, B.G. Lindsay (1989) The Starburst long field manager. In: Proc. 15th International Conference on Very Large Data Bases, pp 375–383, Amsterdam, The Netherlands, August
27. Q. Li, B. Moon (2001) Indexing and querying XML data for regular path expressions. In: Proc. 27th VLDB, pp 361–370, Rome, Italy
28. M.L. McAuliffe, M.J. Carey, M.H. Solomon (1996) Towards effective and efficient free space management. In: Proc. 1996 ACM SIGMOD International Conference on Management of Data, pp 389–400, Montreal, Canada, June
29. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom (1997) Lore: a database management system for semistructured data. SIGMOD Rec 26(3)
30. David Megginson (2001) SAX: A simple API for XML. Technical report, Megginson Technologies
31. H. Meuss, C. Strohmaier (1999) Improving on index structures for structured document retrieval. In: 21st Annual Colloquium on IR Research (IRSG'99)
32. J. Mildenberger (2001) A generic approach for document indexing: design, implementation, and evaluation. Master's thesis, University of Mannheim, Mannheim, Germany, November (in German)
33. C. Mohan (1995) Disk read-write optimizations and data integrity in transaction systems using write-ahead logging. In: P.S. Yu, A. L. P. Chen (eds) Proc. 11th International Conference on Data Engineering, March 6–10, 1995, Taipei, Taiwan, pp 324–331. IEEE Computer, New York
34. C. Mohan, D. Haderle (1994) Algorithms for flexible space management in transaction systems supporting fine-granularity locking. Lecture Notes in Computer Science, vol. 779. Springer, Berlin Heidelberg New York, pp. 131–144
35. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz (1992) ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans Database Syst 17(1):94–162
36. C. Mohan, Frank Levine (1992) Aries/im: an efficient and high concurrency index management method using write-ahead logging. In: M. Stonebraker (ed) Proc. 1992 ACM SIGMOD International Conference on Management of Data, San Diego, Calif., June 2–5, 1992, pp 371–380. ACM, New York
37. C. Mohan, H. Pirahesh (1991) Aries-rrh: restricted repeating of history in the aries transaction recovery method. In: Proc. 7th International Conference on Data Engineering, April 8–12, 1991, Kobe, Japan, pp 718–727. IEEE Computer, New York
38. R. Schiele (2001) NatiXync: Synchronisation for XML database systems. Master's thesis, University of Mannheim, Mannheim, Germany, September
(in German).
39. A. Schmidt, M. Kersten, M. Windhouwer, F. Waas (2000) Efficient relational storage and retrieval of XML documents. In: ACM SIGMOD Workshop on the Web and Databases (WebDB)
40. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton (1999) Relational databases for querying XML documents: limitations and opportunities. In: Proc. 25th International Conference on Very Large Data Bases, pp 302–314, Edinburgh, Scotland, UK
41. B. Surjanto, N. Ritter, H. Loeser (2000) XML content management based on object-relational database technology. In: Proc. 1st Int. Conf. on Web Information Systems Engineering (WISE), pp 64–73
42. R. van Zwol, P.M.G. Apers, A.N. Wilschut (1999) Modeling and querying semistructured data with MOA. In: ICDT'99 Workshop on Query Processing for semistructured data
43. G. Weikum, G. Vossen (2002) Transactional information systems: theory, algorithms and the practice of concurrency control and recovery. Morgan Kaufmann, San Francisco
44. T. Westmann, D. Kossmann, S. Helmer, G. Moerkotte (2000) The implementation and performance of compressed databases. SIGMOD Rec 29(3):55–67
45. G. Wiederhold (1987) File organization for database design. McGraw-Hill Computer Science Series. McGraw-Hill, New York
46. I.H. Witten, A. Moffat, T.C. Bell (1999) Managing gigabytes. Morgan Kaufmann, San Francisco
47. T.W. Yan, J. Annevelink (1994) Integrating a structured-text retrieval system with an object-oriented database system. In: Very large data bases, VLDB '94: Proc. 20th International Conference on Very Large Data Bases, pp 740–749, Santiago, Chile