Anatomy of a Native XML Base Management System

Thorsten Fiebig Sven Helmer Carl-Christian Kanne
Julia Mildenberger Guido Moerkotte Robert Schiele
Till Westmann

Fakultät für Mathematik und Informatik
D7, 27
Universität Mannheim
68131 Mannheim
Germany
Thorsten.Fiebig@softwareag.com
[helmer|moerkotte]@informatik.uni-mannheim.de
kanne@data-ex-machina.de
julia@pi3.informatik.uni-mannheim.de
rschiele@uni-mannheim.de
Till.Westmann@xyleme.com

Abstract

Several alternatives to manage large XML document collections exist, ranging from file systems over relational or other database systems to specifically tailored XML repositories. 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.

1 Introduction

As XML [6] becomes widely accepted, the need for systematic and efficient storage of XML documents arises. For this reason we have developed Natix, a native XML repository that is custom tailored to the requirements of processing XML documents. In our opinion such a system has to fulfill several prerequisites. First of all it has to allow storing documents effectively and accessing these documents (or parts of the documents) in an efficient manner. Associated with this is the support of standardized, declarative query languages like XPath [9] and XQuery [8]. An XML repository should also feature interfaces like SAX [29] and DOM [22] to assist in the rapid development of applications. Last but not least a safe multi-user environment via a transaction manager should be provided, which must be able to recover the system after a crash and synchronize the concurrent access of many users. Here we describe how we put Natix in a position to cope with all these tasks.

We are aware of the fact that several approaches based on traditional database management systems (DBMSs) exist, e.g. storing XML in relational DBMSs or object-oriented DBMSs [13, 17, 25, 28, 37, 38, 39, 40]. We believe, however, that a native XML base management system is the more promising solution, as approaches that map 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. As relational systems represent data as tuples in flat tables, 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 several 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 throughout the rest of the paper.

The contributions of the paper are the following. We introduce a storage format that clusters subtrees of an XML document tree into physical records of limited size. This storage format solves the above mentioned dilemma. To improve recovery in the XML context, we develop *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 access directly 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 nor as a stream of SAX events.

Obviously, a viable database management system for XML should support all these representations. Natix's query execution engine is not only flexible enough to do so but also highly efficient.

The rest of the paper is organized as follows. In the next section we present the overall architecture of the system. The storage engine is the subject of Section 3. This is followed by a description of the transaction management in Section 4. Next, we take a look at the query execution engine in Section 5. Finally, we conclude our paper with a summary in Section 6.

2 Architecture

This section gives a brief introduction into the architecture of the Natix system. We identify the different components of the system and their responsibilities. The rest of the paper then deals with some of the components in greater detail.

While Natix was mainly realized in C++ on Unix platforms, experimental Windows versions and a Java binding exist.

Natix' components form three layers, as shown in figure 1. The bottommost 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*, which consists of all the modules that map application data and requests from other application programming interfaces to the Natix Engine Interface and vice versa.

2.1 Storage Layer

The storage engine contains classes for efficient XML storage as well as several 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 section 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 maybe some result fields (for query results or other return values) have been filled in, the request object is returned to the caller. Typical requests include ‘process query’, ‘abort transaction’ or ‘import document’.

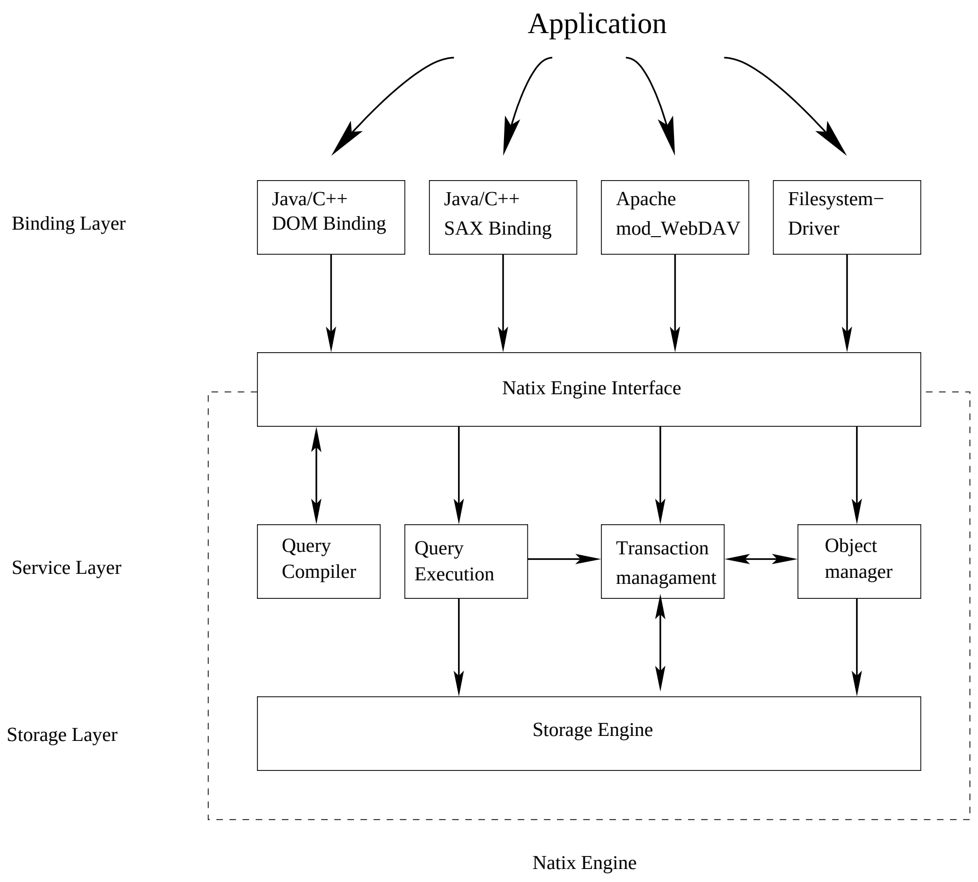


Figure 1: Architectural overview

Among the service components that implement the functionality needed for the different requests are

The Natix Query Execution Engine (NQE) contains the Natix Physical Algebra (NPA) and the Natix Virtual Machine (NVM). They are used to efficiently evaluate queries. Apart from simple operations on basic types, NVM also understands more powerful commands to access XML structures. These directly access the physical document structure used on secondary storage, without CPU-intensive conversion to a main memory representation (for example a DOM tree).

More on query processing can be found in section 5.

Query Compiler The query compiler translates queries expressed in XML query languages into execution plans for NQE. A simple compiler for XPath is available, and a more complex cost-based query optimizer (BD2) is in the works.

Transaction Management contains classes that provide ACID-style transactions. Components for recovery and isolation are located here. Both of these aspects yield challenges with respect to XML, which are related to the different views one can take on XML documents, the data-centric view and the document-centric view.

On one hand, applications that are based on the data-centric view usually access and update single nodes. On the other hand, the document-centric view causes manipulation of coarser granularities, like whole documents or subtrees of documents.

For both recovery and isolation, simply mapping operations on coarse granularities to operations on single nodes neutralizes a lot of performance potential, as even for simple documents, thousands of locks and log records have to be created.

If both access patterns have to be supported in an efficient way, more sophisticated techniques have to be used. Details can be found in section 4.

Object Manager The different application programming interfaces (APIs) often require a main memory representation for documents and their component nodes. To avoid each application binding from implementing its own management for transferring objects between their main and secondary memory representations, the object manager factorizes the representation-independent parts of this task.

2.3 Binding Layer

In the binding layer, the Natix Engine Interface is translated into different application interfaces.

XML database management is required by a great range of different environments. Apart from the classic client-server database system, possibly using protocols like HTTP or WebDAV [18]. For embedded systems it might be more appropriate to use an XML storage and processing library with a high performance, direct function call interface. Legacy applications that are not specifically aware of XML documents but can only deal with plain files would need the database to be mounted as a file system. Other, more XML specific interfaces may arise when XML databases are more widely used.

The most straightforward interface is to use the Natix Engine Interface directly as a library from C++ applications.

A higher-level application binding is the file system view, a demonstration of which is available for download [12]: Using this binding, documents 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.

Whereever 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 and can translate strings into request objects. This simple control interface for the system can easily be incorporated into generic APIs: By using request strings as URLs, for example, the HTTP protocol can be used to control the database system.

3 Storage Engine

3.1 Architecture

In figure 2, the different modules of the storage subsystem and their call relationships are shown.

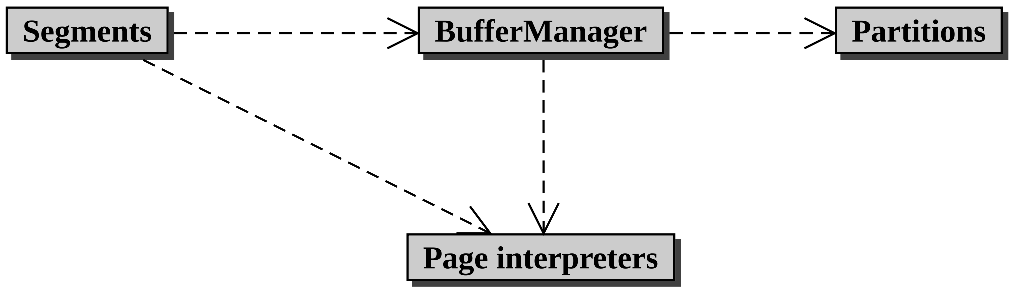


Figure 2: Storage Engine Architecture

Storage in Natix is organized into *partitions*, which represent storage devices that can randomly read and write a fixed number of disk pages. Disk pages are grouped in *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 sections will elaborate on the storage system's main modules, while later sections will give further information about implementation of concrete segment types used to store documents and indexes.

3.1.1 Segments

Segments export the main interfaces to the storage system. They implement large, persistent object collections.

The most important segment types are

SlottedPageSegment unordered collections of variable-size records, where each record is smaller than a page and is identified using a RID (record identifier) that is stable even if the record is moved.

BTreeSegments for secondary and primary key B-Tree structures

XMLSegments for XML document collections.

The segment classes form a class hierarchy, the base class of which factorizes common administrative functions like free space and metadata management. Each segment consists of a collection of pages on which the data structures are stored. These page collections are maintained using an extent-based system [43] that organizes segments into consecutive page groups (*extents*) of variable size. Intra-segment free space management is done using a Free Space Inventory FSI [32]. This data structure allows quick access to approximate information about free space on a given page, which is necessary when performing a search for a given amount of space for insertion or growth of objects. Small main-memory caches of the FSI information is maintained for increased performance (a similar approach is described in [27]).

Every segment type has a different interface providing the operations necessary to manipulate the respective persistent data structure. The data structures managed by the segments can be larger than a page. Operations on such structures are mapped onto (sequences of) operations on single pages.

3.1.2 Buffer Manager

The buffer manager is reponsible for transferring pages between main and secondary memory. Segments request main memory addresses for disk pages by issuing fetch calls for page IDs. After they have finished working with a page, segments make an unfix call to allow replacement of the page in the buffer. Special calls exist for assigning buffer space to a page without loading its current disk contents (to avoid read access for new pages), and to discard dirty pages from the buffer without writing their contents to disk (to avoid writing deallocated pages).

The buffer manager also synchronizes page access by multiple threads. The techniques are similar to the buffer manager organization described by [33]: Each buffer page is synchronized by a latch. Latches are short-duration locks that guarantee the physical consistency of page contents by allowing only readers to share page access, while writers must hold the page latch in exclusive mode. The associative access structure mapping page IDs to memory buffer frames is synchronized in a way that allows for symmetric multi-processing (SMP) with several CPUs, where several processors want to look up page IDs and modify the buffer's contents in parallel.

3.1.3 Page Interpreters

The segments use the buffer manager to load disk pages into main memory. 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. Hence, a page's contents in the buffer are never accessed directly. Which kind of page interpreter is used for a particular page is specified by the segment when it requests a page. A B-Tree segment might use page interpreter classes for inner pages and leaf pages, respectively.

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 page interpreters tremendously simplifies implementation of the recovery subsystem, as described in a later section.

The page interpreter classes form a class hierarchy, the base class of which provides support for the common protocol the interpreters use with the buffer manager. From this base class, one or more classes are derived for each segment type.

3.1.4 Partitions

Partitions represent an abstraction of random-access block devices.

Besides the user segments, each partition contains several metadata segments describing the segments on the partition and the free space available.

The classes in the partition hierarchy themselves implement the basic partition interface, which consists of a small set of block-level read/write functions. Currently, there exist partition classes for Unix files, raw disk access under Linux and Solaris, and C++ iostreams.

3.2 XML Storage

One of the core segment types in Natix is the XML storage segment, which manages a flat collection of XML documents and is based on a slotted page segment as described above.

Before detailing the XML storage segment, we will first give a short overview of existing approaches to store XML documents. They can be divided into three categories:

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, 7, 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].

A Web server's HTML file tree, stored in the file system, is a simple example.

Metamodeling A different method is to model and store the documents or data trees using some conventional DBMS and its data model [13, 17, 25, 28, 37, 38, 39, 40].

In this case, the interaction with structured databases in the same DBMS is easy. On the other hand, scanning a whole document or parts of a document, as needed when reconstructing a textual representation, for example, is slower as in the previous method. The creation of just one typical web page from its abstract syntax tree requires retrieval of maybe thousands of database objects. Other representations require complex mapping operations to reproduce a textual representation [38], even duplicate elimination may be required [13].

In general, this approach introduces additional layers in the DBMS between the logical data and the physical data storage, slowing down query processing.

Mixed There are several attempts to merge the two ”pure” methods above.

Redundant To get the best of both worlds, data is held in two redundant repositories, one flat and one metamodeled [45]. 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.

Hybrid 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 [5].

Natix is similar to the hybrid systems, with two major extensions:

First, our ”flat” parts of the database are not completely flat, but clustered groups of tree nodes treated as atomic records by a slotted page segment. Second, the ”threshold” need not be statically configured, but can be a dynamic value, adapting to the size and structure of documents at runtime. As subtrees of the document are changed, clustered nodes can become records of their own or again be merged into clusters. To satisfy special application requirements, clustering of certain node types can be enforced or forbidden by a configuration matrix.

For more detailed comparisons with the related approaches please refer to [24].

In the following we will describe the logical document data model used by the XML segment to work with documents, and the storage format used by the XML page

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

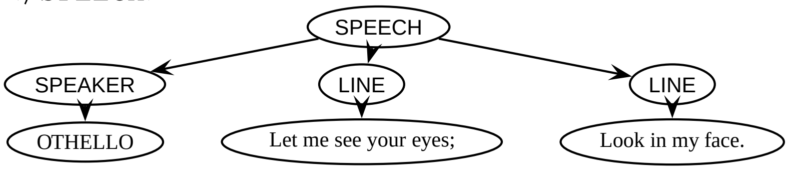


Figure 3: A fragment of XML with its associated logical tree

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 on pages. Finally, we elaborate on the maintenance algorithm for this storage format.

3.2.1 Logical data model

The interface for our XML segment type allows to access an unordered set of trees. New nodes can be inserted as children or siblings of existing nodes, and any node can be removed, which also removes its induced subtree.

The individual documents are represented as ordered trees in which each non-leaf node is labelled with a symbol taken from an alphabet Σ_{DTD}. Leaf nodes can also be labelled with arbitrarily long strings over a different alphabet (Σ^*). Figure 3 shows a an XML fragment and its associated tree. Note that the shown XML document is missing the schema, called *document type definition* (DTD). Details of XML schema descriptions do not concern us here. For our purposes, the DTD is just a way of specifying the node alphabet Σ_{DTD}. Additionally, the DTD can place constraints on how node labels can be combined.

The simple labelled tree model described here is very similar to XML abstract syntax trees. It also captures all information present in the textual representation of a document, most notably the order of child elements. 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. This wrapper is not discussed here for ease of exposition.

3.2.2 XML page interpreter storage format

The logical data tree is partitioned into subtrees (as described in section 3.2.3), each of which is stored on a single data page. XML page interpreters are used to maintain the subtrees on the data pages, which besides the logical nodes contain additional nodes needed to manage the physical structure of large trees. Large trees are trees that cannot be stored on a single disk page.

Note that in the following we use the terms *node* and *object* synonymously for the nodes of a tree. On the other hand, a *record* is something different: It may contain a set of nodes/objects, as explained below.

Classified by their contents, there are three types of nodes in the page-level subtrees:

Aggregate nodes are inner nodes of the tree. They contain their respective child nodes.

Literal nodes are leaf nodes containing an uninterpreted stream of bytes, like text strings, graphics, or audio/video sequences.

Proxy nodes are nodes which point to different records. They are used in the representation of large trees, as detailed in section 3.2.3.

The page interpreters 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.

Instead of placing each tree node in a separate record on the page, we store the nodes of a subtree (maybe a whole document) together in one record. Each record contains exactly one subtree. Each record contains a parent record pointer (used to connect it to a larger tree, see below), and the document ID of the document it belongs to.

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 64K pages are the maximum). We will not go into detail about the exact representation that is used, and 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, for example, already needs 7 bytes ($<\mathtt{X}>\ldots</\mathtt{X}>$)! On average, XML documents inside Natix consume about as much space as XML documents stored as plain files in the file system.

3.2.3 XML Segment Mapping for Large Trees

Typical XML trees may not fit on a single disk page. Hence, the XML segment type must map the logical document to a set of page-level subtrees, and our page-level model must provide data structures to represent connections between the trees.

One method often used in document management systems is to store a ”flat” representation as a BLOB (*binary large object*) and use a mechanism for managing large byte collections inside the storage manager [4, 7, 26]. We feel that this approach wastes

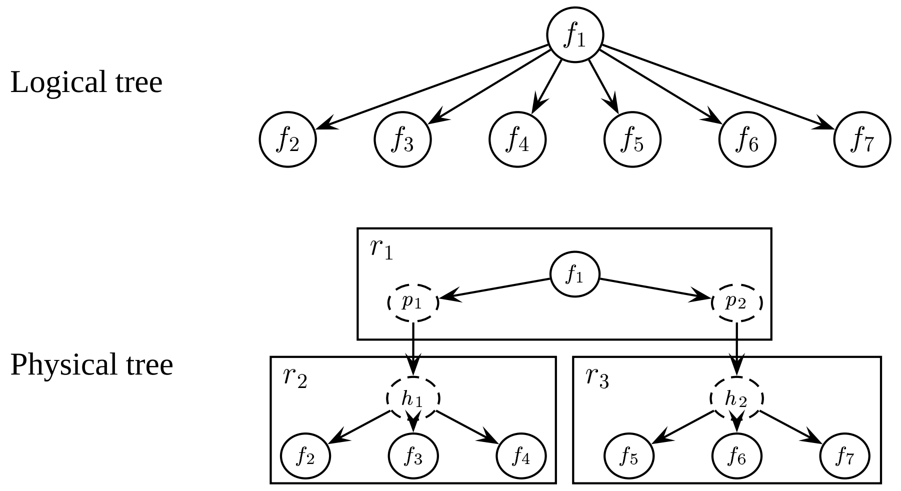


Figure 4: One possibility for distribution of logical nodes on records

the available structural information about the data, because treating the representation as a BLOB regards all bytes as equal:

A certain amount of insertions, removals and updates of objects stored in this way would lead to an unfavorable distribution of the data. Some part of even a small object would reside on one page, and the remainder on a different page.

To avoid this, we *semantically* split large objects based on the underlying tree structure. We partition the data tree into subtrees that are managed by XML page interpreters as described above. Connected subtrees residing in other records are referred to by *Proxy objects*. Proxy objects consist of the RID (record identifier, see section 3.1.1) of the record which contains the subtree they represent. Substituting all proxies by their respective subtrees reconstructs the original data tree. To allow upward navigation, each subtree contains a parent record pointer that refers to the record containing its proxy. The parent record pointer for the root record is null.

A sample is shown in figure 4. To store the given logical tree (which, say, does not fit on a page), the physical data tree is distributed over three records r_1, r_2 and r_3. To achieve this, two proxies (p_1 and p_2) are used in the top level record. Two helper aggregate objects (h_1 and h_2) have also been added to the physical object tree. They are needed to group the children below p_1 and p_2 into records.

Physical objects 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 objects*, while objects representing logical nodes (f_i) are called *Facade objects*. Only the facade objects are visible to the caller of the XML segment interface, the scaffolding objects are encapsulated.

Note that the sample physical tree is only one possibility to store the shown logical tree. There are more, since more of the logical tree's edges could be represented by proxies.

The following section will now describe the method used to map the logical document trees onto subtrees smaller than a page.

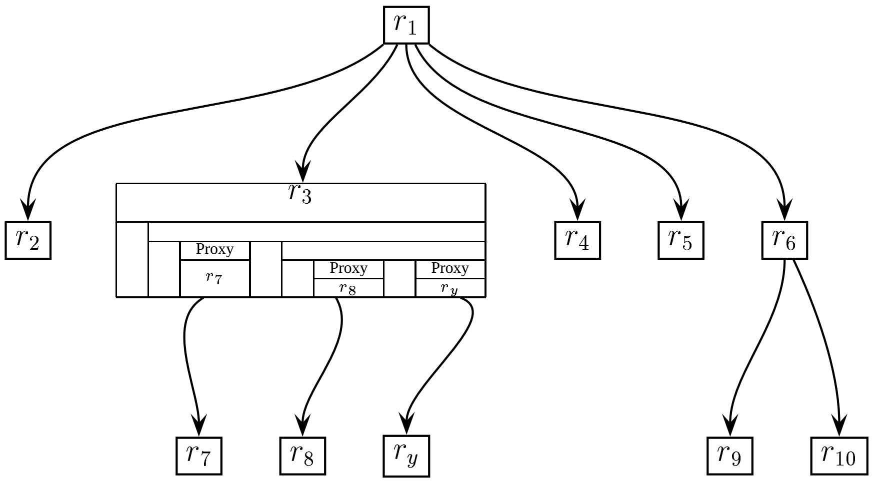


Figure 5: Multiway tree representation of records

3.2.4 Updating Documents

We will now present the algorithm used by Natix for dynamic maintenance of physical trees. The principal problem adressed is that a record containing a subtree can grow larger than a page if a node is added or grows.

In this case, the subtree contained in the record has to be partitioned into several subtrees, which can subsequently be distributed on one or more additional records and pages. 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 our 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 configuration 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 the records as nodes, each record containing a small part of the logical data tree. In the example in figure 5, r_3 is blown up, hinting at the flat representation of the subtree inside record r_3. The references to the child records are proxy objects.

If viewed this way, our partitioned tree resembles a B-Tree-structure, as often used by traditional large object managers, with the particularity that the "‘keys"’ are not taken from a simple domain like integers or strings. Instead, they are based on structural features of the data tree.

This analogy gives us a familiar framework with which we can describe the algorithms used to maintain the clustering of our records.

1. Determine the record r into which the node has to be inserted.
2. If there is not enough space on the page, try to move r. If the node still does not fit into the record, split the record:
   (a) Determine the separator by recursively descending into the r's subtree
   (b) Distribute the resulting partitions onto records
   (c) Insert the separator into the parent record, recursively calling this procedure
3. Insert the new node

Figure 6: The Tree Growth Procedure

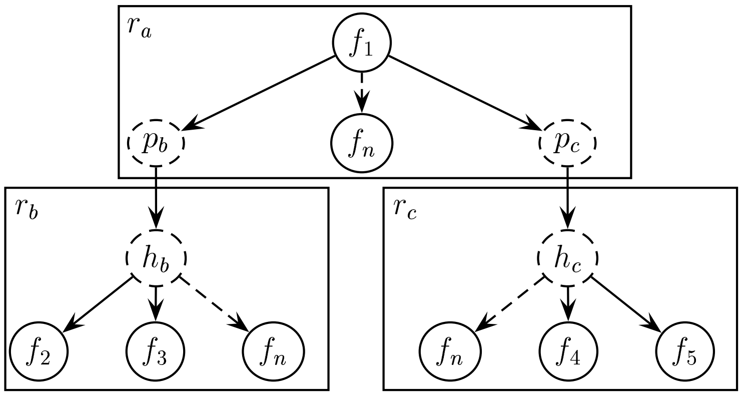


Figure 7: Possibilities to insert a new node f_n into the physical tree

Algorithm for Tree Growth Figure 6 shows the basic structure of our algorithm for adding nodes to a tree. We now explain the steps in detail.

1. Determining the Insertion Location In order 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 should take place. In the presence of scaffolding nodes, there may exist several possibilities, as shown by the dashed lines in figure 7 (the nodes drawn as dashed ovals are scaffolding nodes); the new node f_n can be inserted into r_a, r_b, or r_c. In our system, this choice may be determined by a configuration parameter, as explained in section 3.2.4.

2. Splitting a record Having decided on the insertion location, it is possible that the designated record's disk page is full. In this case, 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 has to be split.

(a) Determining the separator Suppose that in figure 7 we want to add f_n to record r_b, which cannot grow. Hence, r_b must be split into at least two

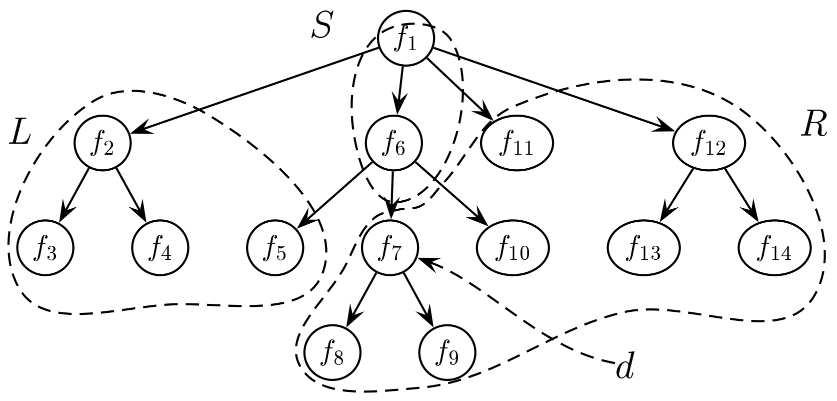


Figure 8: A record's subtree before a split occurs

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 that partitions the data elements into two subsets is chosen as separator. In our tree storage manager, the data in the records is not one-dimensional, but tree-structured. It follows that our separator has to be tree-structured as well.

In fact our algorithm slices a small subtree off the old record's root. This small subtree then serves as separator. The remaining forest of subtrees is the data that has to be distributed onto the new records.

Figure 8 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.

Already a single node d uniquely determines this partitioning (in the example, $d = f_7$): The separator S consists of all the nodes on the path from d to the subtree's root (in the example, $S = \{f_1, f_6\}$), excluding d. The subtree below 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, our split algorithm must find a node d, such that the resulting L and R are of 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 preorder from left to right). Another configuration parameter available for fine-tuning is the *split tolerance*, which states how much the algorithm may deviate from this ratio. Essentially, the split tolerance specifies a minimum

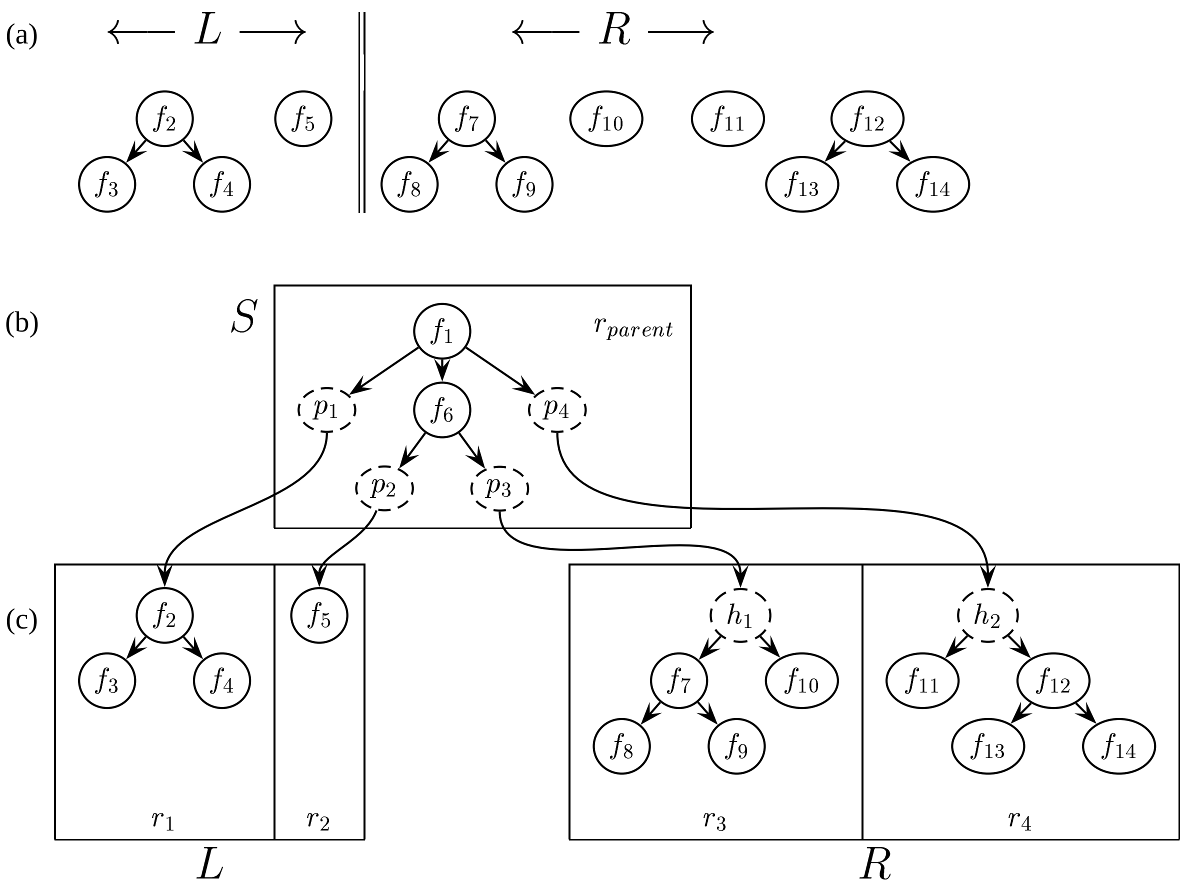


Figure 9: Record assembly for the subtree from figure 8

size for the subtree of d. 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 subtree size in which it is about to descend is smaller than allowed by the split tolerance parameter.

In the example in figure 8, the size of the subtree below f_7 was smaller than the split tolerance, otherwise the algorithm had descended further and made f_7 part of the separator.

(b) Distributing the nodes on records After determining the partitioning, the contents of the record have to be distributed onto new records.

Consider a partitioning as implied by node $d = f_7$ in figure 8. The separator is removed from the old record's subtree, as in figure 9(a). 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 figure 9(c), 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 and inserted instead of the proxy which referred to the old, unsplit record, figure 9(b). 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 would consist 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.

To ensure that the parent record contains enough space to hold the separator, the insertion procedure is recursively called for the parent record using the separator as the node to be inserted. If the old record had no parent record, a new root record for the tree is created which contains just the separator.

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, it also results in a preferable partitioning since it takes the space needed by the new node into account 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 had to be considered. In general, 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.

To express preferences regarding the clustering of a node type and its parent node type, we introduce a *Split Matrix* as parameter to our algorithm:

The Split Matrix S consists of elements $s_{ij}, i, j \in \Sigma_{\mathrm{DTD}}$. The elements express the desired clustering behaviour of a node x with label j as children of a node y with label i:

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

The algorithm as described in section 3.2.4 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 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 for splitting purposes like the root record.

We also use the Split Matrix as the configuration parameter for determining the insertion location of a new node (see section 3.2.4): When a new node x (label j) shall be inserted as a child of node y (label i), then if $s_{ij} = \infty$, x is inserted into the same record y. If $s_{ij} = \textbf{other}$, then the node is inserted on the same record as one of its designated siblings (whereever exists more free space). 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, as described in [24].

3.3 Index Structures in Natix

In order to support query evaluation efficiently, we need powerful index structures. The main problem in building indexes for XML repositories is that ordinary full text indexes do not suffice, as we also want to consider the structure of the stored documents. Here we describe the approaches taken by us to integrate indexes for XML documents in 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) [30], that can deal with a lot more than structural information. On the other hand we developed a novel index structure, called XASR (eXtendend Access Support Relation) [14], for Natix.

3.3.1 Full Text Index Framework

Inverted files are the index of choice in the information retrieval context [2, 44]. In the last years the performance of inverted files improved considerably, mostly due to clever compression techniques. Usually inverted files store lists of document references to indicate in which documents certain words 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) with references without compromising the performance.

Let us briefly sketch the architecture of the list implementation in Natix before going into details on the five different classes Index, ListManager, FragmentedList, ListFragment, and ContextDescription (see also Figure 10).

Index The main task of the class Index is to map search terms to list identifiers and to store those mappings persistently. It also provides the main interface for the user to work with inverted files.

ListManager This class maps the list identifiers to the actual lists, so it is responsible for managing the directory of the inverted file. If the user works directly with identifiers and not with search terms, it is possible to use ListManager directly. We have implemented efficient methods for bulkload and bulkremoval of lists, as well as for concatenation of lists, namely union and intersection.

FragmentedList, ListFragment We describe these modules together, because they are tightly coupled with each other. ListFragment is an implementation of lists that need at most one page of memory to store. The content of a fragment can be read and written sequentially. All fragments that belong to one inverted list

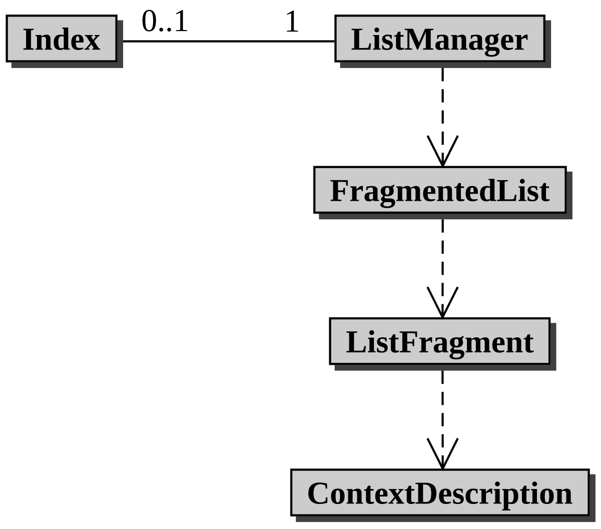


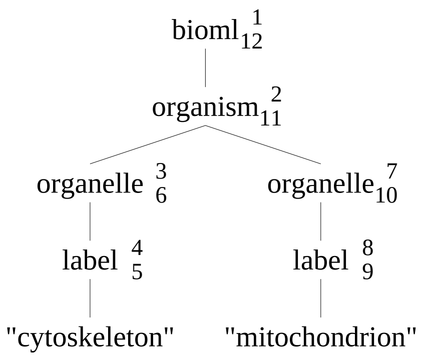
Figure 10: The classes for list management and relations

are linked together and can be traversed sequentially. The job of the class FragmentedList is to manage all the fragments of one list and control insertions and deletions on this list.

ContextDescription This class determines the actual representation in which data is stored in a list. With representation we do not only mean what kind of data is stored, but also the compression technique that is used. We have implemented the traditional context consisting of a document ID and the positions within the document where a certain term appears. More importantly, we devised contexts for XML data. A simple node context consists of a document ID, a node ID, and the position of the search term within the node, whereas a more complex node context also considers structural information (e.g. d_{min} and d_{max} values as described in Section 3.3.2).

3.3.2 eXtended Access Support Relations

An extended access support relation (XASR) is an index that preserves the parent/child and ancestor/descendant relationships among the nodes. This is done by labeling the nodes of an XML document tree by depth-first traversal (see Figure 11). 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 name of the element tag, the document ID, and the d_{min} value of the parent node (see Figure 11).



(a) Example document tree

d_{min}	d_{max}	eType	docID	parent_d_{min}
1	12	’bioml’	0	NULL
2	11	’organism’	0	1
3	6	’organelle’	0	2
4	5	’label’	0	3
7	10	’organelle’	0	2
8	0	’label’	0	7

(b) corresponding table

Figure 11: XASR

XASR is combined with regular full text indexes that supply the node numbers of nodes containing words we are searching for. That means, if we are looking for nodes containing specific words or nodes of a certain type in a path, we also join these nodes to the nodes fetched by XASR.

A path in a query is translated into a sequence of joins on an XASR table. Let x_i and x_{i+1} be two nodes in the path. Depending on the path connector (’/’ or ’//’) the join predicate is

- for ’/’:
  x_i.docID $= x_{i+1}$.docID$\wedge$
  $x_i.d_{min} = x_{i+1}$.parentID
- for ’//’:
  x_i.docID $= x_{i+1}$.docID$\wedge$
  $x_i.d_{min} < x_{i+1}.d_{min}\wedge$
  $x_i.d_{max} > x_{i+1}.d_{max}$

For more details on query processing with XASRs see [14].

4 Transaction Management

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

To be an effective tool for enterprise-level applications, Natix therefore must provide transaction management for XML documents with the above-mentioned properties. The transaction components that support transaction-oriented programming in

Natix are the subject of this section. The two main areas that are covered are recovery and isolation, in this order.

For recovery, we use an extended version of the well-known ARIES protocol [33]. The extensions that we introduce (called subsidiary logging, annihilator undo, and selective redo) exploit certain opportunities to improve logging and recovery performance that are, although present in many environments, especially effective with our XML storage format.

For synchronization, a S2PL-based scheduler is employed that provides a protocol and lock modes that are suitable for typical access patterns that occur for tree-structured documents.

The two subsections describing recovery and isolation are preceded by an introduction that outlines how transaction management fits into the system architecture.

4.1 Architecture

In figure 12 we show 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 in section 3, while the rest is part of the transaction management module.

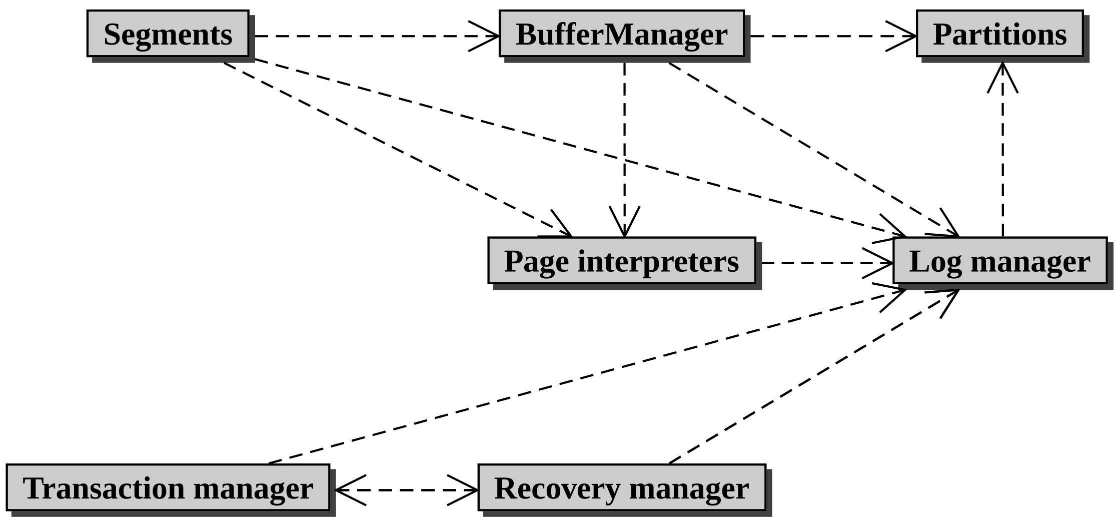


Figure 12: Recovery Components

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 the ARIES concepts can be found in the original ARIES paper [33] and most books on transaction processing (e.g. [20, 41]).

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 physical location on disk.

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, the objectID (segmentID, pageID, slot number, offset in a XML subtree) and, possibly, data to redo and/or undo the logged operation.

The log records of a transaction are linked together into a pointer chain. In ARIES, there is a prevLSN pointer that 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 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 log records that describe undo actions, so called compensation log records (CLRs), the undoNextLSN points to the operation logged before the operation that this log record is an undo log record for. In section 4.4, we show another situation where the nextUndoLSN chain of log records is just not the reverse sequence of log records of one transaction.

4.2.2 Segments

The segment classes comprise, from the view of the recovery subsystem, 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 data structures provided by the segments can be larger than a page. The segments map operations on these data structures to operations on single pages, and employ the buffer manager to transfer the pages between main memory and disk storage.

Logging and recovery for operations on the pages is dealt with by the page interpreters (see section 4.2.3). 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 these operations L1 operations (following [41]). They occur for segment types where a high degree of concurrency is required (e.g. B-Trees [34]), 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 splitted 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 [32, 31], are far from being simple implementation issues but involve delicate dependencies, and to keep the system simple and maintainable, they require an architecture that is 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* [20].

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 also has a member attribute redoLSN that contains the LSN of the first update operation after the last flush.

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 [33], the buffer manager enables adherence to the ARIES protocol by notifying other componentes about page transfers between main and secondary memory, and by logging information about the buffer contents during checkpoints.

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.

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 also maintains the mapping of log records to LSN (and its inverse). It also persistently stores the LSN of the most frequent checkpoint.

During undo processing, the log manager knows that a particular transaction is rolling back. Instead of chaining a log record to the previous log record of the same transaction, the log manager properly links the nextUndoLSN field to the log record of the operation currently being undone, as required for CLRs by the ARIES protocol.

The automatic undoLSN chaining by the log manager allows for the 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 a 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. 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

Apart from the segment classes, the transaction manager is the only class that is directly called by application programs.

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 last written log record, a 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 that is currently being undone to provide automatic nextUndoLSN chaining for CLRs.

The pending actions list contains a set of operations that has 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 (section 4.3).

4.3 Subsidiary Logging

Conventional recovery systems that use logging follow the principle that every modification operation is immediately preceded or followed by the creation of a log record for that operation.

Operation usually means 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 will explain how Natix reduces log size and increases concurrency, boosting overall performance, by relaxing both of these constraints.

Suppose a given record is updated multiple times by the same transaction, which is frequent when using the storage layout for XML documents as described in section 3.2, for example when a subtree is added node-by-node. In many cases, it would be desirable if this composite update operation was logged as one big operation, for example by logging the complete subtree insertion as one operation. Merging the log records would avoid the overhead of log record headers for each node (which can be as much as 100% for small nodes), and would reduce the number of serialized calls to the log manager, increasing concurrency.

In the following, we will sketch how Natix' recovery architecture supports such logging optimizations. We will also elaborate on the concrete implementation for the 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 log records, and how those log records are interpreted during undo and redo is up to the page interpreter.

The page interpreter has its own state for each memory-resident page, which it can use to collect logging information 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, to follow the force-at-commit rule of ARIES.

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 is the same. Basically, the rules below cause a sequence of operations by one transaction on one page to be treated by logging and recovery as a single atomic update operation.

Implementation of the rules is rather straightforward in Natix' architecture. 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 than maintain a subsidiary log can add a pending action to the transaction control block (see section 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 into subsidiary logs is described in [23].

4.3.2 XML-Page subsidiary logging

We now describe how to employ the technique outlined in the previous section for a concrete page interpreter class (namely XML pages) to improve performance for the logging version of that class. Logging for XML data was the primary reason for introducing 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. The Natix storage format (section 3.2) will usually cause such a subtree insertion to be mapped into a sequence of updates on a single record.

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 5 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 localized and can easily be expressed in terms of one single insert operation to the record, logging this single operation would allow 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, so applications would need to know about the mapping of the logical data structures to pages, breaking down encapsulation.

We will now show how a subsidiary log inside the XML page interpreters allows to amortize logging costs for subtree insertions.

Using the page contents as Subsidiary Log The log entries for the subsidiary log are not explicitly stored. Instead, the XML page interpreters reuse the data page as 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 in the page are marked. All information necessary to log the subtree insertion is available inside the data page itself, except for the transaction id. 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 records of the page. 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.

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. So, the log is not only published as outlined in section 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 transaction IDs that created the subtrees. The transaction IDs are not stored in the data page's contents.

This means that XML page interpreters must reserve some extra storage to store the transaction IDs. 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. As we do not like to add small object dynamic storage management to our page interpreters just to store some transactionIDs, we limit the subsidiary log to a fixed amount of transactionIDs. If more than this amount of transactions wants to add subsidiary log entries, we publish the subsidiary log to the log manager. On average, only a very small amount 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, frequent XML page splits occur (refer to section 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 main memory page buffer, then 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 that have been created. Even if not the whole document resides in the buffer, subsidiary logging profits by only creating log records as needed.

4.4 Annihilator Undo

Transaction undo often wastes CPU resources, because more operations are executed than necessary to recreate the state that is 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

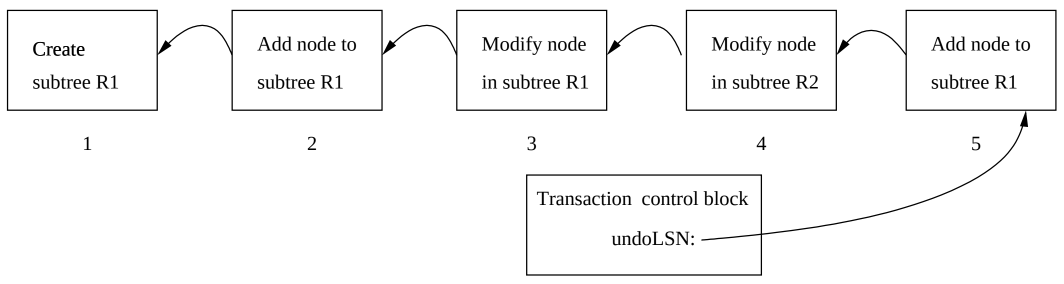


Figure 13: Log records for an XML update transaction

to be deleted as a result of transaction rollback anyway. Refer to figure 13 which shows a transaction's 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.

We call undo operations that imply undo of other operations that follow 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 that have been done to the record.

For better undo performance, it is desirable to skip undo of operations implied by the annihilators.

Natix realizes this to some extent. Let us recall from section 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, then the operation can be logged as a redo-only operation. This will prevent the operation from entering then 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 that is generated.

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

Let us now look at the way Natix exploits the optimization potential described

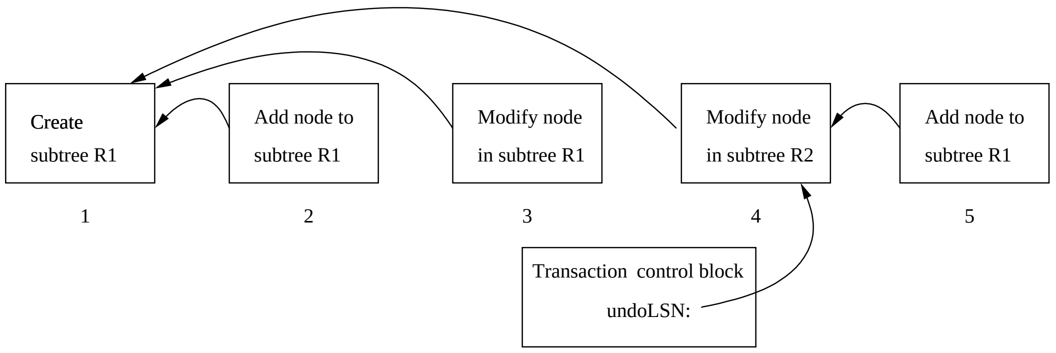


Figure 14: Undo chaining with check for annihilators

above for the special case of XML data.

The XML page interpreters augment the stored information for the subtree as follows: In every XML subtree record header, an annihilator LSN is stored that contains 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 update operations for XML subtrees now check whether the stored annihilatorLSN for the subtree that is going to be modified is greater or equal to the last savepointLSN. If yes, then 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 14 shows the resulting undo chain after log records 1–5 from the example in figure 13 have been written, under the assumption 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. For example, if 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 explictly 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.

It is possible to store the annihilatorLSN in main-memory only, in the state of the page interpreter object. This would disallow annihilator undo if the page is kicked out of the buffer, which should be an unlikely event.

Another possibily is to store the annihilatorLSN for subtrees in the lock control blocks (see section 4.6 below) of the corresponding object. We do not consider this a good solution, as it introduces an additional dependency between the implementation of the recovery and synchronization components, which makes the already complicated maintenance of those modules even harder.

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.

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. The redo pass that accomplishes this is followed by an unconditional undo pass that undoes changes of loser transactions.

In the presence of fine-granularity locking, when multiple transactions may access the same page concurrently, the redo-history method together with writing log records that describe actions taken during undo (compensation log records, or CLRs) is necessary for proper recovery. 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 transaction would be necessary.

In Natix, records used to store XML documents are frequently very large, so that each page only contains very few records, reducing the amount of concurrent access to pages. Since large buffers are also the rule and not the exception, 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 [35], 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.

The 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 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-granulartiy 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 than one transactions have updates on a dirty page. This information is included in the dirty page checkpoint log records, and as 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 always know what the LSN of the current on-disk version of a dirty page is (this is recorded in 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. This flag 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 disallows write access to a page while it is written to disk, even if this is done from a consistent memory copy. In [23] we show how to relax this condition.

4.6 Synchronization Components

Since XML documents are only semi-structured, we cannot apply synchronization mechanisms used in traditional fully structured relational databases. As already mentioned, XML documents can be represented as a tree where the root node represents the whole XML document and the other nodes either contain data or nodes. This tree structure suggests using tree locking protocols as described, e.g. in [3, 41]. 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 that are held at a certain point in time. Tamino, a commercial product by Software AG, solves this problem by locking whole XML documents, which limits 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 [20] supporting multi granularity locking (MGL) and strict two-phase locking (2PL) 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 [36].

4.6.1 Lock Protocol

Granularities. Locks can be requested at the segment, document, subtree and physical 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 figure 15. Note that with the split matrix the user can enforce splitting a document into physical records such that those parts of the document that are likely to be modified concurrently reside in different physical records. As we will see, concurrent updates on different physical 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 [20] 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 Figure 16. In [21] independent lock modes for structural and content related operations were used. Essentially, this is

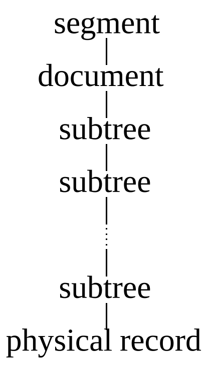
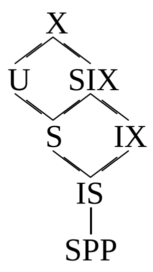


Figure 15: hierarchy of granularities

not necessary as the type of node (representing structural information or actual content) implicitly determines the type of operation.



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

Figure 16: Hierarchy and compatibility matrix of lock modes

Protocol. As in the NO2PL protocol described in [21], 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 assume the existence of pointers between siblings and from a parent node to its first and last child in an XML tree. So 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. They should be seen as an auxiliary, logical construct. Also note that in view of having physical 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. An application often traverses a document from the root to the leaves. Consequently, 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.

Take, for example, a transaction having held shared locks so far decides to request an exclusive lock on a node. Then it has to go back up the hierarchy of granularities until it finds an appropriate lock ("IX" in this case) held by the transaction or it has reached the top of the hierarchy. On our way back down it acquires the appropriate intention locks ("IX" in this case).

Jumps. However, transactions do not behave well all times. For example, they access an arbitrary node within the tree 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 must be to lock all nodes from the node X the transaction jumped to up to the root node of the document with the coresponding 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. Note that this is very dangerous since in an extreme case a whole subtree containing X 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 aquires SPP locks. Since SPP locks are incompatible with X locks, we can only traverse up to the root if no other transaction performed prior changes to the document that would endanger serializability. Once the transaction reaches the root, it traverses down the path from the root to node X and thereby converting the SPP locks to the required lock mode. Note that the SPP locks prevent other transactions to interfere while walking down.

4.6.2 Special Issues

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. The first stage involves escalating locks to the corresponding lock on the document level. We use a heuristic to determine documents for which a high number of locks is held. All locks for these documents are then escalated to the document level. On demand we repeat this process on other documents on which the transaction holds locks. If we are still not satisfied and all locks on documents are already escalated, we do the same on the segment level. In the very unlikely situation that we still cannot meet the lock request, the request is denied.

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

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. When a deadlock is detected, the requested lock for the current transaction is denied. In order to prevent the system from stalling because of deadlock detection, we only start a search if none is running at the moment.

4.6.3 Implementation

This protocol is implemented in two major classes. The most important classes and their relationships are shown in figure 17.

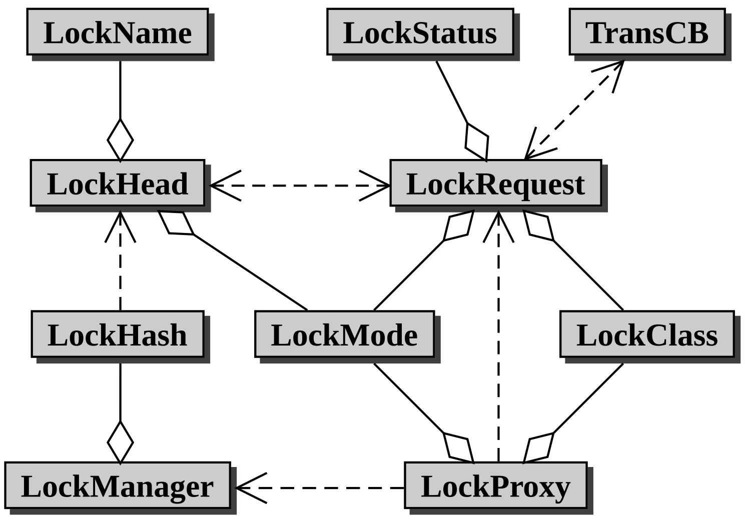


Figure 17: The classes for synchronization and their relationships

LockManager. The class LockManager, which is a singleton (i.e., only one instance exists at a time), provides basic locking primitives similar to the ones presented by Gray and Reuter [20]. Its responsibility is deadlock detection, without considering MGL rules, however. It also contains LockHash, which in turn directly or indirectly references all LockHead objects. These LockHead objects comprise the LockName and the accumulated LockMode of all granted lock requests. Additionally a LockHead object also references a list of all LockRequests for this lock. LockRequest objects store the current LockStatus, the current LockMode and in case of a pending lock conversion the desired LockMode. A LockRequest also contains the LockClass of this request, although this feature is currently not used and only implemented for future enhancements. In addition to that the LockRequest objects reference the transaction control block they belong to. A transaction control block references a list of all locks belonging to the corresponding transaction.

LockProxy. Another important class is the LockProxy, which assures strict 2PL locking, performs lock escalation and implements the enhanced MGL protocol. Every transaction has one instance of this class and every lock request is first sent to this class. The proxy then issues the appropriate requests to the lock manager. For caching reasons it references some strategic LockRequests to prevent sending multiple requests for the same lock. It also contains a LockMode object for internal reasons and the LockClass, which is assigned to every new lock requested.

Requirements and restrictions of the current implementation. The current implementation assumes that the database provides information on the parent node ID of any node in an XML document that is traversed during lock acquisition. We need this for lock requests on behalf of index jumps to acquire intention locks on parent nodes.

At the moment we only support single-threaded transactions assuming that a transaction does not request more than one lock concurrently. Also splits and shifts of records are only allowed if the modifying transaction holds exclusive locks on those records.

5 Natix Query Execution Engine

5.1 Overview

When 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. By expressiveness we mean that our query execution engine is able to execute all queries expressible in a typical XML query language like XQuery [8]. By flexibility we mean that the algebraic operators implemented in the Natix Physical Algebra (NPA)—the first major component of NQE—are powerful and versatile operators. 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 does make sense. A third alternative is to represent the query result as a sequence of SAX [29] 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* [19]. 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.

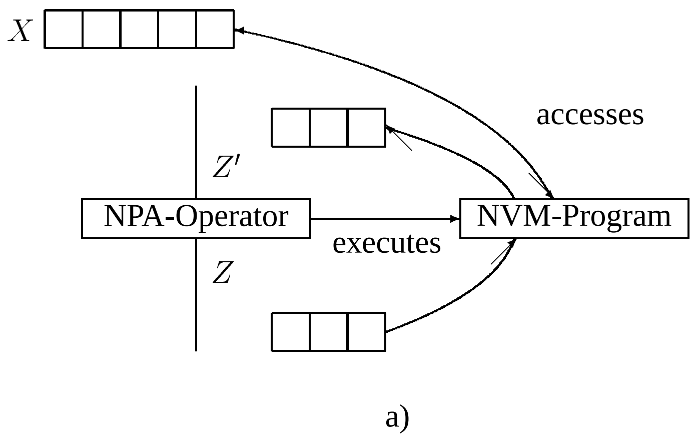
Accordingly, the `close` call has been split into `finish`, `deinitialize`, and `destroy`. The main reason for this split is the efficient support of nested algebraic expressions which are used for example to represent nested queries that for efficiency or other reasons are not unnested by the query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

NPA operators ususally 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.

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. At any time, an NVM program is able to access several register sets. The situation is illustrated in Fig. 18 for unary (a) and a binary NPA operators (b). 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 for example. It may also be used for intermediate results or to pass information down for nested query execution. Between operators, the tuples are stored in Z and Y registers where Y registers are only available for binary operators. For example, in case of a join operator the Z register contains an outer tuple and the Y register contains an inner tuple.



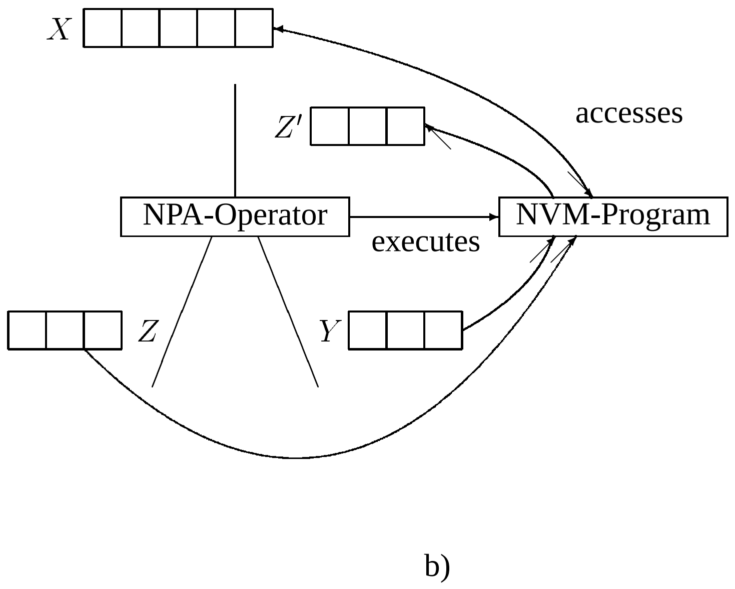


Figure 18: Register Sets, NPA-Operators and NVM-Programs

It is database lore that most of the time is spend on copying data around during query execution. 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 non pipeline breakers 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 according values while scanning and 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 straight forward to copy the next 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 copying. 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 or the register set or to return a different register set.

The global register set X only contains information that is global for a single plan execution. Other information needed by the NVM is contained in control blocks also accessible during program interpretation. These control blocks hold information about the current transaction and the current session.

Summarizing, while executing a program the NVM has access to the X, Z, possibly Y register sets and control blocks. However, for most commands only the Z or X register sets need to be accessed.

NVM commands NVM commands can be divided into groups. For example, there exists a group for arithmetic commands. 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 whith a group name followed by the command. Then an optional result mode (borrowed from AVM, see [42]) 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_ZZZ  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_F` 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_F           2
```

The `C` indicates that the according argument is a constant.

The XML specific part of NVM contains about 150 commands. Among these are simple operations that copy a document node handle from one register to another, compare two document handles, print the XML fragment rooted at a document 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 allow to access the attributes of an element node, retrieve its children or descendants. Let us consider an example. As we will see in the next section, evaluating XPath expressions can be performed by a sequence of `UnnestMap` operations. A single `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. Here are the three programs for an `UnnestMap` operator that accesses all child nodes of a document 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_F	3	
	MV_XML_ZX	2	4
step	XML_SIBLING_NEXT_XX	4	4
	XML_VALID_XX	4	3
	EXIT_F	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 these 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. [19]), 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 scan operations. The simplest scan 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` and `PathScan` are used to generate variable bindings for XPath expressions. An XPath expression can be translated into a sequence of `UnnestMap` operations or into a single `PathScan` operation. Consider for example the XPath expression `/a//b/c`. It can be translated into

$$\begin{array}{l}\text{UnnestMap}_{\$4=child(\$3,c)}(\\ \quad\quad \text{UnnestMap}_{\$3=desc(\$2,b)}(\\ \quad\quad\quad \text{UnnestMap}_{\$2=child(\$1,a)}([\$1])))\end{array}$$

However, one has to be careful. Not all XPath expressions can be translated into a sequence of `UnnestMap` operations due to the duplicate eliminating semantics of XPath. For example, the straight forward translation of the path expressions `//*//*` into a sequence of two `UnnestMap`s generates duplicates. For this reason, the `PathScan` exists in Natix. Further, experiments have shown that for path expressions with at least two descendant-or-self axes the `PathScan` is faster than a sequence of `UnnestMap` operations.

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. 19. 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

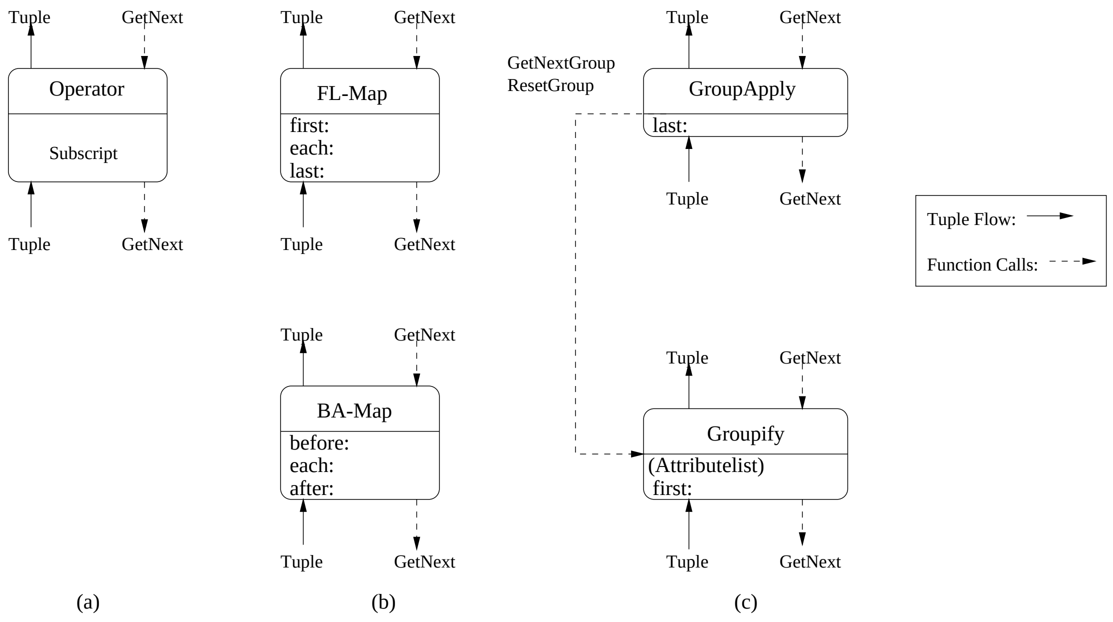


Figure 19: Interfaces of construction operators

`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 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 value of the attributes 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 illustrate the algebraic operators by two plans. Both plans build on a bibliography document whose DTD is shown in Fig. 20. The first plan implements the evaluation strategy for the following XQuery (Query 1):

```
<result>
```

```
<!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>
```

Figure 20: Sample DTD

```
{
  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 according plan is shown in Fig. 21. Note that 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 next query requires restructuring. There, the `Groupify` and `GroupApply` operators are necessary.

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}>
```

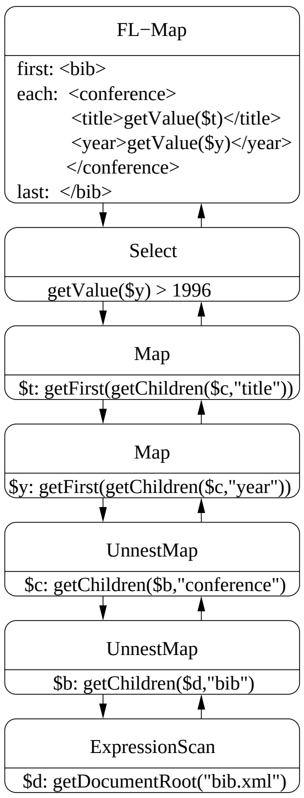

FL−Map
first: <bib>
each: <conference>
<title>getValue($t)</title>
<year>getValue($y)</year>
</conference>
last: </bib>
Select
getValue($y) > 1996
Map
$t: getFirst(getChildren($c,"title"))
Map
$y: getFirst(getChildren($c,"year"))
UnnestMap
$c: getChildren($b,"conference")
UnnestMap
$b: getChildren($d,"bib")
ExpressionScan
$d: getDocumentRoot("bib.xml")

Figure 21: Construction plan of Query 1.

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

The corresponding plan is shown in Fig. 22. 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 then 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. Our contribution is to prepare for the shift in processing XML documents by describing how efficient, native XML base management system can actually be built.

References

[1] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In Rakesh Agrawal, Sean Baker, and David Bell, editors, *Very large data bases, VLDB '93: proceedings of the 19th International Conference on Very Large Data Bases, August 24–27, 1993, Dublin, Ireland,* pages 73–84, Los Altos, CA 94022, USA, 1993. Morgan Kaufmann Publishers. Co-sponsored by VLDB Endowment and Irish Computer Society; in co-operation with the IEEE Technical Committee on Data Engineering.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, Massachusetts, USA, 1987.

[4] Alexandros Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the International Conference on Data Engineering*, pages 301–308, 1992.

[5] Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. Structured document storage and refined declarative and navigational access mechanisms in HyperStorM. *VLDB Journal: Very Large Data Bases*, 6(4):296–311, November 1997. Electronic edition.

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (xml) 1.0 (second edition). Technical report, World Wide Web Consortium (W3C), 2000.

[7] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the EXODUS extensible database system. In Yahiko Kambayashi, Wesley Chu, Georges Gardarin, and Setsuo Ohsuga, editors, *Twelfth international conference on very large data bases, proceedings (VLDB '86)*, pages 91–100, Los Altos, CA 94022, USA, 1986. Morgan Kaufmann Publishers.

[8] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. Xquery 1.0: An xml query language. Technical report, World Wide Web Consortium, 2001. W3C Working Draft 07 June 2001.

[9] James Clark and Steve DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C), 1999.

[10] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.

[11] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[12] data ex machina. NatixFS technology demonstration, 2001.
available at `http://www.data-ex-machina.de/download.html`.

[13] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28(2):431–??, 1999.

[14] T. Fiebig and G. Moerkotte. Evaluating queries on structure with extended access support relations. In *WebDB (Informal Proceedings)*, pages 41–46, Dallas, Texas, 2000.

[15] T. Fiebig and G. Moerkotte. Algebraic XML construction in Natix. In *Web Information Systems Engineering (WISE)*, page to appear., 2001.

[16] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *WWW Journal*, page to appear., 2002.

[17] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[18] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. Http extensions for distributed authoring – webdav. Technical Report RFC2518, Internet Engineering Task Force, February 1999.

[19] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[20] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA 94104-3205, USA, 2000.

[21] S. Helmer, C.-C. Kanne, and G. Moerkotte. Isolation in xml bases. Technical Report Nr. 15, Lehrstuhl für Praktische Informatik III, Universität Mannheim, 2001.

[22] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (dom) level 2 core specification. Technical report, World Wide Web Consortium (W3C), 2000.

[23] Carl-Christian Kanne. *Natix: A Native XML Base Management System*. PhD thesis, University of Mannheim, 2002. To appear.

[24] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. Technical Report Nr. 8, Lehrstuhl für praktische Informatik III, Universität Mannheim, June 1999.

[25] M. Klettke and H. Meyer. XML and object-relational database systems — enhancing structural mappings based on statistics. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[26] Tobin J. Lehman and Bruce G. Lindsay. The Starburst long field manager. In P. M. G. (Petrus Maria Gerardus) Apers and Gio Wiederhold, editors, *Very large data bases: proceedings: proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22–25, 1989, Amsterdam, The Netherlands*, pages 375–383, Los Altos, CA 94022, USA, 1989. Morgan Kaufmann Publishers.

[27] Mark L. McAuliffe, Michael J. Carey, and Marvin H. Solomon. Towards effective and efficient free space management. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):389–??, ???? 1996.

[28] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3):54–??, ???? 1997.

[29] David Megginson. SAX: A simple API for XML. Technical report, Megginson Technologies, 2001.

[30] Julia Mildenberger. A generic approach for document indexing: Design, implementation, and evaluation. Master's thesis, University of Mannheim, Mannheim, Germany, November 2001. (in German).

[31] C. Mohan. Disk read-write optimizations and data integrity in transaction systems using write-ahead logging. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 324–331. IEEE Computer Society, 1995.

[32] C. Mohan and D. Haderle. Algorithms for flexible space management in transaction systems supporting fine-granularity locking. *Lecture Notes in Computer Science*, 779:131–144, 1994.

[33] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992. Also published in/as: IBM Almaden Res. Ctr, Res. R. No. RJ-6649, Jan. 1989, 45 pp.

[34] C. Mohan and Frank Levine. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 371–380. ACM Press, 1992.

[35] C. Mohan and Hamid Pirahesh. Aries-rrh: Restricted repeating of history in the aries transaction recovery method. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 718–727. IEEE Computer Society, 1991.

[36] Robert Schiele. NatiXync: Synchronisation for XML database systems. Master's thesis, University of Mannheim, Mannheim, Germany, September 2001. (in German).

[37] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[38] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very Large Databases, Edinburgh, Scotland, UK, 7–10 September, 1999*, pages 302–314, Los Altos, CA 94022, USA, 1999. Morgan Kaufmann Publishers. Also known as VLDB'99.

[39] B. Surjanto, N. Ritter, and H. Loeser. XML content management based on object-relational database technology. In *Proc. 1st Int. Conf. on Web Information Systems Engineering (WISE),* pages 64–73, 2000.

[40] Roelof van Zwol, Peter M. G. Apers, and Annita N. Wilschut. Modeling and querying semistructured data with moa. In *ICDT'99 Workshop on Query Processing for semistructured data*, 1999.

[41] Gerhard Weikum and Gottfried Vossen. *Transactional information systems : theory, algorithms and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers, San Francisco, CA 94104-3205, USA, 2002.

[42] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[43] Gio Wiederhold. *File organization for database design*. McGraw-Hill computer science series; McGraw-Hill series in computer organization and architecture; McGraw-Hill series in supercomputing and artificial intelligence; McGraw-Hill series in artificial intelligence. McGraw-Hill, New York, NY, USA, 1987.

[44] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, San Francisco, CA 94104-3205, USA, 1999.

[45] Tak W. Yan and Jurgen Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 740–749, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

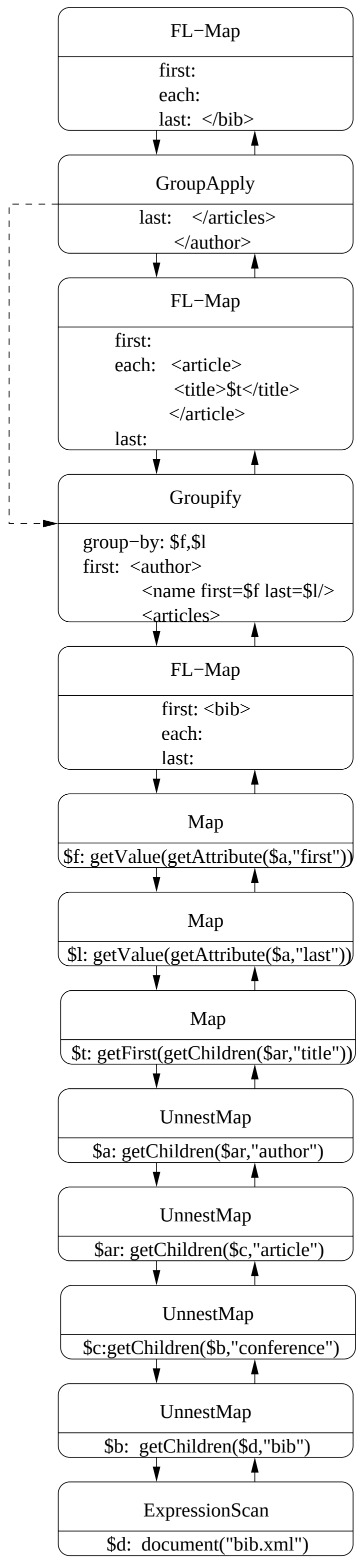


Figure 22: Construction plan of Query 2.