Core Technologies for
Native XML Database Management Systems

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Informatiker Carl-Christian Kanne

aus Herford

Mannheim, 2003

Dekan:	Professor Dr. Herbert Popp, Universität Mannheim
Referent:	Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent:	Professor Alfons Kemper, Ph.D., Universität Passau

Tag der mündlichen Prüfung: 14. Februar 2003

Für Ingrid & Walter;

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I am just now beginning to discover the difficulty of expressing one's ideas on paper. As long as it consists solely of descriptions it is pretty easy; but where reasoning comes into play, to make proper connection, a clearness & a moderate fluency, is to me, as I have said, a difficulty of which I had no idea.

–Charles Darwin

The eXtendible Markup Language (XML) [10] has become a popular data format to integrate applications. It provides application designers and developers with a convenient and standardized way to exchange data beyond the boundary of a single program. By sharing syntax of, tools for, and experience with a common data exchange format across applications, and even across application domains, software integration becomes simpler and less costly.

With increased popularity, the role of XML is changing. XML was designed as an interchange format for arbitrary data, but it is used more and more as a general representation language for data outside of an application's main memory address space. Application data is no longer converted to XML and back for communication purposes only, but is also stored as XML to make it persistent across executions of the same program.

When reliable, scalable, flexible, high-speed processing and management of persistent data is required, we enter the domain of database management systems (DBMS). We call DBMSs designed to manage large collections of XML documents *XML Base Management Systems* (XBMS).

The goal of this work is to investigate the core technologies required to build XBMSs. We identify requirements, and analyze how they can be met using a conventional DBMS. Our conclusion is that an XML support layer on top of an existing conventional DBMS does not address the requirements for XBMSs.

Hence, we built a *Native* XML Base Management System, called Natix. Natix has been developed completely from scratch, incorporating optimizations for high-performance XML processing in those places where they are most effective. Almost all of the classical components of a DBMS are affected in terms of adequacy and performance, including the Storage Engine, Schema Management, Transaction Processing, Query Processing, and Application Programming Interfaces.

A thorough discussion of the complete Natix system and all its techniques and modules is beyond the scope of this work. Instead, we concentrate on the foundation of Natix, its overall architecture, the Storage Engine, Schema Management, and Recovery Subsystem.

Natix introduces a *storage format* that clusters subtrees of an XML document tree into compact physical records of limited size. Our storage format allows for very efficient access to documents and document fragments.

Natix offers means to logically and physically partition the set of all XML documents it manages. Applications can specify logical schemas to group documents into logical collections, which can be used to address data in queries. Physical schemas allow to control how the storage primitives provided by the Storage Engine are used to materialize the application's documents on physical media.

The size of a physical record containing an XML subtree is typically far larger than the size of a physical record representing a tuple in a relational database system. This affects *recovery*. To improve recovery in the XML context, we developed the novel techniques *subsidiary logging* to reduce the log size, *annihilator undo* to accelerate undo and *selective restart* to accelerate restart recovery.

Thesis Outline

XML, its query languages and standard processing interfaces are briefly reviewed in Chapter 2. Chapter 3 discusses how existing database technology can be applied to persistently store XML documents and identifies the flaws of existing DBMSs in this area. Chapter 4 introduces Natix and explains its architecture and interfaces. The remaining chapters deal with details of some of Natix's modules, such as the Storage Engine (Chapter 5), Schema Management (Chapter 6) and Recovery (Chapter 7). Chapter 8 concludes the thesis.

Acknowledgements

I would like to thank the people that worked with me on Natix during the last years.

Without the help of Sören Göckel, Andreas Grünhagen, Alexander Hollmann, Norman May, Julia Neumann, Robert Schiele, and Frank Ueltzhoeffer, Natix would not exist.

Thanks to Wolfgang Leideck, Beate Rossi, Simone Seeger and Uwe Steinel for maintaining an environment that made this possible.

Thanks to Thorsten Fiebig, Till Westmann, and Sven Helmer, for many discussions, brainstorming sessions, papers, discussions, talks, books, hints, reviews and discussions.

Thanks to Guido Moerkotte, for many things.

Thanks to Susanne for keeping me sane, and being there.

Chapter 2

eXtensible Markup Language – XML

The reason clichés become clichés is that they are the hammers and screwdrivers of communication.

–Terry Pratchett

This chapter contains a brief introduction to XML and related standards. The core language is presented, including treatment of namespaces. To prepare our requirements analysis and the system architecture, we review query languages and typical processing interfaces for XML.

2.1 The Core Language

The core XML language is defined in a Recommendation by the World Wide Web Committee [10]. We give only a brief summary, referring to [10] for details.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure.

Essentially, XML is a standardized syntax for *semi-structured data* [12]. Semi-structured data can be defined as a tree with labelled edges and leaves. The markup in XML documents implies a mapping of the document's character sequence to such a tree and vice versa.

Markup always begins with a ’$<$’ character and ends with a ’$>$’ character. The predominant form of markup are *elements*, which denote inner nodes of the tree, or regions of the document. Each element has a *tag name*, which represents the edge label of the edge leading to the node. An element with tag name X begins by an *open tag* $< X >$ and ends with a *close tag* $< /X >$. The data between the open and close tag represents the child nodes of the element.

Character data is mapped to labelled tree leaf nodes.

There are other markup types, including attributes which are attached to elements, processing instructions to encapsulate application-specific metadata, and more.

Document Type Defi nitions, or *DTDs*, can be used to put constraints on the set of allowed tag names and the way elements are nested. More on DTDs can be found in Section 6.1.

2.2 Namespaces

The *markup vocabulary* of a document, its set of possible markup tag names, can be a combination of tag names from different sources and for different purposes. For example, markup for visual layout can be combined with markup for specification of scientific experiments to create experimental protocols.

Naming conflicts can be the result of combining vocabularies. In the above example, the tag name `color` would be meaningful in both vocabularies. To avoid naming conflicts in such situations, tag names can be qualified with namespaces [9]. A namespace is identified using a unique Uniform Resource Identifier [3].

To avoid including the complete namespace URI for every tag name in a document, namespaces are assigned small prefix strings in a document. These prefix strings are then used to qualify tag names. Additionally, default namespaces for subtrees of a document can be specified, implying that all unqualified element names in that subtree belong to its default namespace.

2.3 XPath

A simple query language for XML is *XPath* [18].

The main purpose of XPath is to select document nodes. Given a *context node*, an XPath expression allows to select a *result set* of nodes in a document tree.

The most important construct in XPath expression is the *Location Path*. A Location Path consists of a sequence of *Location Steps*, separated by `/`. Each location step selects a set of document nodes based on a context node. The result of a location path is computed by first calculating the result of the location path without its last step. Each node in this result set is used as context node for the last step. The produced result sets are then unioned to produce the final result.

A location step is of the form $\texttt{axis::nodetest[pred}_1\texttt{][pred}_2\texttt{]...}$. `axis` is a direction of navigation from the context node. `nodetest` is a special predicate (see below), and $\texttt{pred}_i$ are arbitrary predicate expressions which have to evaluate to true for a selected node.

Axes include $child$, $descendant$, $parent$, $ancestor$, $following-sibling$, $preceding-sibling$. A node test is a check for a node type and/or certain tag name.

Hence, the three-location-step expression

```
child::department/descendant::employee/child::name
```

selects the name of all employees in all department tags below the context node, even if there are additional levels of markup between `department` and `employee`. Predicates can base node selection on more complicated expressions:

```
child::department[child::name='Sales']/descendant::employee
```

only selects employees from the Sales department.

The XPath language is used as a building block in more powerful languages, such as XSLT [17] and XQuery [6].

2.4 XML Processing Interfaces

There are many different application programming interfaces (APIs) to process XML documents. Their goals and feature sets vary widely. In the following brief survey, we divide them into three classes.

First, we describe interfaces based on an object-oriented tree model of the documents. Second, we look at interfaces which model documents as a series of events. Third are interfaces intended to deal with document collections and documents as a whole, in contrast to accessing the internal structure of individual documents.

The section concludes with a summary, which lists the interfaces and their features in form of a matrix.

2.4.1 Tree-Based Interfaces

As explained in Section 2.1, semi-structured data rendered as XML can be modeled as a labelled tree. Hence, a straightforward way to access XML documents is to implement an object-oriented tree model of documents using an object-oriented language. Instances of a tree model can easily be created from the document's abstract syntax tree.

Document Object Model

The most prominent member of the tree-based interfaces is the Document Object Model, or DOM [45]. It is a language and platform-independent specification of a set of classes with which document trees can be accessed.

DOM provides a class hierarchy for document node types. A base class `Node` implements basic tree functionality, such as navigation to parent, child, attribute and sibling nodes, and access to node labels. For every type of tree node, special classes exist, including classes for elements, attributes, and text nodes. Tree nodes are referenced through the language's native pointer type.

Factory classes support creation of new nodes. Simple queries can be issued which retrieve nodes based on tag name equality. Document iterators allow to enumerate the elements of a document, possibly filtered by an application-specified filter object.

The standard document specifies several levels of compliance, which require different amounts of support of advanced features including querying, namespaces, document iterators, and conversion into textual format.

The Apache Software Foundations versions for Java [74] and C++ [73] are widely used reference implementations. Implementations for C [15] and other languages exist.

Others

Other tree-based interfaces are the native interfaces of some XML parsers (such as libxml2 [92]), which allow to access the abstract syntax tree in main memory.

JDOM [46] is a DOM variant which incorporates Java-specific idioms instead of relying on language-independent interfaces. In contrast to regular DOM, its node base class is not sufficient to access all aspects of a document. XOM [39] is a similar Java-only interface.

2.4.2 Event-Based Interfaces

Event-based interfaces represent documents as a stream of events occuring while the document is parsed.

SAX

The SAX interface [57] (Simple API for XML) is a push-interface, where application programs register callback-handlers. Processing is initiated by calling a `parse()` method. The XML processor then traverses the document tree in depth-first sequence.

Every time a node is visited, a callback function of the registered handler is called, notifying the application about the begin or end of an element, a text node, or any other part of the document tree.

Several implementaions of SAX exist in different languages including C [92], C++ [73] and Java [74].

Others

The XNI interface [74] is similar to SAX, but allows to annotate events and pass them on to other handlers. It allows to customize XML processors by configuring the sequence of scanner, schema validator, tree builder and other application specific add-on handlers that is traversed by each event.

The XMLPull [40] interface uses an iterator interface for events, which allows the application to pull individual events from the XML processor by application request.

2.4.3 Document-Based Interfaces

In contrast to the XML interfaces above, which deal with the internal structure of documents, document-based interfaces operate on whole documents and document collections.

WebDAV

WebDAV [32] (Web-based Distributed Authoring and Versioning) is a communications protocol for the internet which allows transfer, modification, version management and concurrency control for documents.

WebDAV is built on top of HTTP [28], which is also used for read access to documents. Hence, every WebDAV server is also an HTTP web server.

Documents can be organized in arbitraily deep hierarchies, and special primitives exist to lock documents, to avoid conflicts if multiple users try to work with the same document concurrently. Iterator primitives allow to enumerate all documents and subcollections in a collection.

WebDAV is not aware of XML documents, apart from the fact that a per-document attribute allows applications to specify the document format, which may be XML. Document contents are transferred as byte stream.

However, we list WebDAV as XML processing interface, as it is commonly used to maintain web sites and intranet document stores. Hence, it is a popular means of remotely accessing XML documents, and as such needs to be supported in an XBMS.

XML:DB

The XML:DB initiative [84] specifies an interface intended to provide uniform access to XBMSs of different vendors.

The specification is targeted to the Java language, and provides classes to manipulate and query document collections and documents. Individual documents can be accessed using either DOM or SAX interfaces.

Query support currently is limited to the XPath language. An XPath-based update query language is under development.

An extension mechanism allows to introduce additional interface features while maintaining backwards compatibility.

2.4.4 Summary

Table 2.1 lists some of the surveyed APIs and supported features.

Features Below, we explain the meaning of the rows in Table 2.1.

Structure is checked when the interface allows to access the internal document tree structure.

Collections refers to the availability of a *collection of documents* as first-class object in the interface.

Iteration indicates whether the interface provides a means to iterate over sets of nodes or documents.

XML-aware Has the interface been designed specifically for XML?

	DOM	SAX	WebDAV	XML:DB
structure	$\surd$	$\surd$	–	–[1]
Collections	–	–	$\surd$	$\surd$
iteration	$\surd$	–	–	$\surd$
XML-aware	$\surd$	$\surd$	–	$\surd$
Non-XML data	–	–	$\surd$	$\surd$
Explicit Concurreny	–	–	$\surd$	–
Multidocument	–	$\surd$	$\surd$	$\surd$
Queries	$\surd$[2]	–	–[3]	$\surd$[4]
Updates	$\surd$	–	$\surd$	–[5]

Table 2.1: API properties

Non-XML data Is it possible to process non-XML documents with the interface?

Explicit Concurrency is checked when the interface provides primitives to lock documents or parts of documents to prevent access conflicts.

Multidocument refers to the possibility of using the interface to process more than one document at a time.

Queries indicates whether the interface provides primitives to query documents.

Updates is checked when applications can modify documents using the interface.

Remarks 1) XML:DB includes DOM and SAX Java packages to access document structure. DOM-based updates need to be propagated as a complete document tree, hence, updates of individual nodes is impossible. 2) The query capability of DOM is limited to an tag-name equality query for a single document. 3) WebDAV Search allows for a limited form of keyword search. 4) The query support is not mandatory and limited to XPath 5) The DOM interface is a subinterface of XML:DB and may be used to update documents. An XUpdate language is under development.

Chapter 3

XML Base Management

Fundamental progress has to do with the reinterpretation of basic ideas.

–Alfred North Whitehead

A primary design goal for XML was to have a flexible exchange format for data between applications. To exchange data, it must be converted into a format that is meaningful outside the current application's main memory address space.

XML's increased popularity has made knowledge about XML and related technologies common among developers. A large number of tools is available. This makes XML a natural ingredient for many other situations where data must leave the current application's address space, apart from mere data exchange: For all kinds of *persistent* application data that is not easily mapped to the tabular structure of relational DBMSs, XML is rapidly becoming the storage format of choice.

Reliable, scalable, flexible, high-speed processing of persistent data is the domain of database management systems (DBMS). How can DBMS technology be used to manage XML document collections? This chapter examines answers to this question.

In Section 3.1, we look at some application domains in which XML is employed, establishing the core requirements for an *XML Base Management System* (*XBMS*). Then, we examine typical tasks that occur when working with an XBMS in more detail (Section 3.2).

In Section 3.3, by considering different approaches that use traditional DBMSs to realize the mentioned tasks, we establish a central fact:

Traditional DBMSs fall short of addressing all the requirements for XBMSs.

This is illustrated concisely in Section 3.3.3, which contains a matrix mapping the different approaches to the resulting problems. It motivates the construction of our XBMS Natix, which is introduced in Chapter 4.

3.1 Requirements

A general-purpose XBMS for large-scale XML processing has to fulfill several requirements:

1. To store documents effectively and to support efficient retrieval and update of these documents or parts of them.

2. To support the standardized declarative query languages XPath [18] and XQuery [6].

3. To support the standardized application programming interfaces (APIs) and protocols, such as SAX [57], DOM [45], and WebDAV [32].

4. Last but not least a safe multi-user environment via a transaction manager has to be provided including recovery and synchronization of concurrent access.

We motivate these core requirements using two concrete application domains.

3.1.1 Online Auctions

An online auctioning site manages auctions for a great variety of products. The products which are classified into categories such as ”cars”, ”software”, or ”furniture”. Product descriptions assign values to a set of attributes, where the set of attributes varies depending on the category. For cars, atttributes might include engine power, age, and color, while software offers list version and vendor. Some descriptive text and images are included in all categories. For an internet-based solution, a straightforward representation of such product descriptions are XML documents.

Other documents that are processed include descriptions and ratings of auction participants, templates for dynamically generated web pages, themed product catalogs, and other editorial documents. Progress of auctions is also captured in documents. Bids, messages and final price are appended to the auction document.

Efficient access to all those documents is crucial to the maintenance of a responsive site in the presence of many actors in the system. Apart from whole documents, parts of the documents are also frequently accessed, for example when constructing auction summary views, then only important parts of an auction document are required, such as a short description and the current price. Requirement 1 follows.

Users and editors of the site need powerful and efficient search capabilities, for example when looking for auctions, or when creating special themed product selections. Search functionality can be implemented with much less effort and cost when declarative query languages are available. The selection of documents or parts of documents based on predicates is also required by many other modules in a dynamic web site, establishing requirement 2.

To create the dynamic web pages based on the users’ navigation through a site, stylesheet processors are employed [72]. These are typically based on standard interfaces such as DOM or SAX, pointing to requirement 3. This requirement also results from third-party editors and other maintenance tools for web site management, which need standard interfaces like WebDAV [32] to operate on the documents.

Finally, the concurrent access of many users and maintainers of the site coupled with the direct economic relevance of the managed documents, make reliable transaction management indispensable.

3.1.2 Life sciences

Another example for an application domain exhibiting requirements (1)-(4) are life sciences. There, annotated biosequences (DNA, RNA, and amino acid) are a predominant form of data.

The sequences and especially their annotations are commonly represented using XML, which makes (1) an obvious requirement. As above with auction documents, in addition to complete sequences with all annotations, a typical access granularity here is a fragment of the document, for example a single coded protein in a gene, including its annotations.

Typically, the annotated sequence data is processed by a mix of tools for generation, visualization, (further) annotation, mining, and integration of data from different sources. Existing XML tools relying on DOM or SAX interfaces need to be integrated (3).

The emergence of new scientific methods regularly requires new tools. Their rapid development is facilitated by declarative XML query languages because they render trivial the frequently recurring task of selecting sequences based on their annotation (2).

Each sequence is analyzed using several costly experiments which are performed concurrently. Their results are valuable annotations added to the sequence data. Obviously, full-fledged transaction management reduces the risk of wasting resources (4).

3.1.3 Conclusion

Requirements for large-scale persistent XML processing do not differ *in principle* from those for large-scale management of other forms of persistent data.

However, the application interfaces, data model, and query languages are very different from traditional data management approaches. The consequence is that at least a mapping layer is required when storing XML in a traditional DBMS.

The remainder of this chapter argues why such a mapping layer is undesirable both from a software engineering perspective, as it makes an application more complex and costly, and from a performance perspective, because resources are wasted.

3.2 XBMS Tasks

We elaborate on the general requirements stated above by specifying typical tasks that an XBMS must support. The task descriptions are used in the following section to explain the difficulties encountered by traditional data management paradigms when addressing the issues involved with XML document processing.

3.2.1 Document import

Due to XML's origin as a data interchange format, relevant XML documents are frequently generated outside the XBMS, and need to be imported.

As input, the XML document exists in form of a file in the file system, or as a stream of bytes, for example from a network connection.

As a result of a successful document import, the application is given an identifier which allows to address the imported document in future operations.

3.2.2 Document Object Model

Applications frequently access documents using a programming language binding of the *Document Object Model* (Section 2.4.1). DOM allows to read, update, create and delete single nodes. Navigational operations allow to traverse a document by moving from a node to sibling, parent, or child nodes.

An XBMS must allow access to stored documents using a DOM interface, including updates. Of course, updates performed using DOM have to be governed by transaction management.

The worst-case amount of used resources (CPU and memory) for DOM operations should be a function of the number of involved nodes, and not a function of the number of nodes in the document. This latter requirement is necessary to reproduce the performance behaviour of typical main-memory implementations of DOM.

In particular, in a scalable XBMS it must be possible to use DOM to process documents that are larger than the available main memory.

3.2.3 XPath Query

XPath [18] is a simple declarative query language, described in Section 2.3. XPath expressions must be evaluated in an XBMS not only to answer direct user queries, but also when performing *XSL transformations* (*XSLT* [17]), or when answering queries formulated using *XQuery* [6]. Both of these languages contain XPath as sublanguage.

Hence, efficient evaluation of XPath expressions for a set of context nodes is a very important task of an XBMS.

3.3 Existing DBMS Technology and XML

The following discussion of the adequacy of existing DBMS technology for storage and processing of XML documents is based on the general requirements and tasks from Sections 3.1 and 3.2.

As we will see, all of the approaches to implement an XBMS using an existing non-XML DBMS suffer from serious drawbacks. This insight motivates our approach to create a native XBMS, where support for XML is not simply welded on top but distributed throughout the DBMS.

3.3.1 Relational DBMS

Relational DBMSs (RDBMSs) represent the matured knowledge about large-scale data management. A natural approach to implement an XBMS is to add an XML module on top of an RDBMS.

To represent XML documents in an RDBMS, a relational schema has to be designed. Three basically different classes of schemas to store XML exist: Storing documents in *character large objects*, storing *single nodes* in separate rows, and deriving a relational schema from a *specifi c document schema*.

Each of these approaches is quite different, and the implementation of typical tasks is treated in separate sections below.

Character Large Objects

A straightforward way of modeling XML documents is to put each in a separate row of a single table. Most RDBMSs offer a *character large object*, or *CLOB*, data type whose values can be arbitrarily large strings.

Document import Simply storing the textual representation of XML documents in a field of type CLOB imports a document into the DBMS.

Neglecting administrative overhead, the document consumes about as much space inside the DBMS as in the file system, is controlled by the transaction manager, and protected by recovery. Besides, exporting the document is simple and efficient, given an efficient CLOB implementation.

DOM However, if the document structure has to be accessed, as necessary when implementing a DOM interface, there is no difference to the situation in the file system. To create a DOM tree from the textual representation, it must be analyzed lexicographically and syntactically, and a main-memory DOM representation has to be built. The amount of main-memory and CPU which are consumed to do so are a function of the document size. This is not a scalable approach.

Additionally, in multi-user environments, even if attribute-level locking is supported by the relational DBMS, a document has always to be locked in entirety. Unless the DBMS supports a suitable kind of range locking on CLOBs, concurrent updates to the same document are impossible. Suitable here means that the access and locking mechanisms need to provide logical CLOB fragment addresses that are independent from their current offset in the CLOB. Physical fragment addresses could be invalidated by inserting or removing parts of the CLOB.

If the document has been modified in its main-memory form, propagating the update to the stored representation requires to serialize the whole document into textual form. Because the RDBMS does not know about the semantics of the CLOB, without special techniques this means the whole document also has to be logged for recovery purposes. If intra-transactional savepoints are required, the application itself needs to implement support for them, because the DBMS does not know about the application's main-memory structures.

XPath Evaluation Evaluation of XPath queries requires access to the document structure, and the same scalability problems as for DOM access apply. An additional concern here is that several query compilers and evaluation engines are involved to answer a single query: The RDBMS' native relational query compiler and engine are used to gain access to the required document(s). A separate XPath compiler and evaluation engine are used to answer the query inside the document. Such an evaluation process is unnecessarily complex from a software engineering point of view.

Single Nodes

A second approach when implementing an XBMS with an RDBMS transforms a tree model for XML documents into a relational schema where each document node is represented by a separate row. Each tuple needs, in addition to its contents, information about its type, and a way to locate its neighboring nodes in the tree (siblings, children and parent nodes). The parent-child relationship can be stored using a parent foreign key reference. Since an XML document tree is an ordered tree, the order of child nodes of a parent must be materialized. This can be done by numbering siblings, or by including further references or additional relations.

Document import Document import parses the document, and stores a tuple for each node.

The per-node data is likely to take up more space than the markup in the original document. For example, the markup for a 2-character tag takes up 7 bytes (`<xx>` ... `</xx>`). The slot information for a slotted page record in an RDBMS alone probably costs as much. A parent reference and the other per-node data, such as ordering information, will blow the record up and cause an RDBMS to use up much more space than the original document, incurring additional I/O overhead. If foreign key references are used, additional unique indexes to efficiently maintain them are required, causing extra random I/O and storage space overhead.

Transaction management is also problematic. In an RDBMS, there is no locking granularity between the whole relation and a single record. Hence, every row that is created on behalf of a document has to be locked individually, requiring a large number of locks for a simple document import. The situation for recovery is similar, as an extra log record has to be generated for every document node. The created log volume is significant, as it not only includes the storage overhead as explained above, but in addition adds recovery-specific overhead, such as log record chaining.

DOM Implementing a DOM interface on top of a single-node schema requires to evaluate a query for every navigational operation in the tree. Evaluation of such queries for every navigation step is quite costly. It also results in excessive random I/O, as indices have to be consulted and the nodes are not necessarily clustered according to their document tree neighborhood.

When a DOM subtree is modified, a simple numbering scheme to represent node ordering is dangerous, as insertions in the middle of a sibling sequence require expensive renumberings. For updates, storage space and transaction management overheads mentioned for document import carry over. Even worse, concurrent updates may become impossible. To guarantee serializability, the multigranularity protocol in use by most RDBMSs [34] requires to lock the whole relation in shared mode when repeatable reads of documents or document subtrees are required, to avoid phantom problems. As a consequence, concurrent inserts of new nodes are impossible.

XPath Evaluation The storage space overhead and isolation problems also effect XPath evaluation. Other performance disadvantages occur depending on how XPath expressions are evaluated.

If an XPath processor is used that operates on the DOM interface, then for the evaluation of a single XPath expression many navigation queries are issued to perform the processor's steps in the tree. This is extremely expensive.

If XPath queries are translated into a relational query [26], the problem of multiple compilers mentioned above for CLOB storage resurfaces: The query must first be translated into an SQL query, and which has to be translated into an execution plan. Resources are wasted. Even simple XPath expressions result in costly query plans. For example, simply enumerating all descendants of a node is – without other measures – a recursive SQL query, which typically is quite expensive.

Specific Schemas

If only specific document schemas are to be supported, then it is possible to derive relational schemas, exploiting knowledge about which elements have to occur together, and whether arbitrary nesting is possible [20, 82].

This limits the application domain of the XBMS to those covered by the given schemas, unless automated translation of XML schemas to relational schemas is performed. Even then, many of the problems for the single node approach remain.

Translation of navigational operations to SQL queries is still required, and multiple translation of XPath expression only becomes more complicated.

While the specific schema approach can reduce the storage, logging and locking overhead, as long as several rows are required to represent a document and several documents share one relation, the concurrency issues related to the multigranularity locking protocol are not resolved.

3.3.2 Object-oriented DBMS

The discussion of relational DBMSs above also applies to OODBMSs. However, object-oriented or object-relational DBMSs are better equipped to manage object references and sets of object references, which can be used to optimize the *single node* approach.

Approach		Document Import	DOM Access	XPath Evaulation
File System		1,14	1,5	1,5,14
	extra index	1,13	1,5,13	1,5,13
RDBMS	CLOB		4,8,5,7(,1)	4,8,5,10
	single nodes	6,7,8,11	3,6,7,8,12	6,8,10
	specifi c schema	2,7,8	2,3,7,8,12	2,8,10
OODBMS	single nodes	6,7,9	6,7,9,12	6,9,10

Table 3.1: XBMS approach/problem matrix

By converting the tree model directly as schema for an OODBMS, for example by using the IDL description of the *Document Object Model*, more efficient support for the DOM interface is implicit. Tree edges are represented by object references, which can be traversed without index access if physical OIDs are used. Pointer swizzling further speeds up this process. Tree order can be represented by storing the list of children for each node using a collection-valued attribute. This also solves the locking problem, as now individual node sets and subtrees can be locked without affecting other documents.

Still, the storage and log space overhead and the large number of locks required for each document cause performance problems.

3.3.3 Summary

As summary, we repeat the issues encountered when dealing with the implementation of the tasks specified in Section 3.2.

1. No transaction management, in particular no recovery.
2. Only a fixed number of document schemas is supported .
3. Each navigational operation must be mapped to a query.
4. Parsing is required even if only a part of a document is accessed.
5. Scalability (only parts of documents may reside in main memory).
6. Storage space overhead and I/O overhead.
7. High log volume.
8. No concurrent updates due to mismatching lock granularities.
9. Large number of small granularity locks.
10. Multiple Compilation required.
11. Index required for efficient structure navigation.

12. Excessive Random I/O.
13. Synchronization of stores required.
14. No index structures available.

In Table 3.1, the issues are grouped by approach. It becomes obvious that no approach using existing technologies is free of serious drawbacks. Just adding an XML support module on top of an existing DBMS does not solve any of the encountered problems at all.

Instead, the lower layers of DBMSs need to be engineered for XML support. A *native* XBMS is required that incorporates XML specifics in almost any part of its architecture. Such a system is introduced in the following chapter.

Chapter 4

Natix

Architektur ist gefrorene Musik

–Johann Wolfgang von Goethe

As argued in Chapter 3, implementing an XBMS is not just a matter of adding an XML support component to a conventional DBMS. Instead, almost all components of a DBMS are affected in terms of adequacy and performance.

This chapter introduces Natix, a *native* XBMS, which is built from scratch to support management of XML documents. We give an overview of its architecture and briefly sketch the responsibilities and collaborations of the subsystems. References to other chapters or additional literature describing these components in greater detail are given where appropriate. Examples show how a native XBMS can support application development by supplying powerful constructs for persistent XML processing.

4.1 Architectural Overview

An overview of Natix's system architecture with all major subsystems is shown in Figure 4.1.

The core of the system is implemented as a C++ library, comprising the components below the ”Library Interface” line in Figure 4.1. In general, neighbouring boxes represent packages of classes with C++ function call dependencies in one or both directions.

Applications can use the library directly, or via one of the modules on top of the library interface which export language bindings other than C++, or protocols in addition to function calls.

Below, we elaborate on the subsystems. We begin with system control, which acts as a facade, implementing the library interface. Then we explain different scenarios in which applications can use Natix. Introductions to the remaining components of the XBMS library follow, and some example code using Natix concludes the chapter.

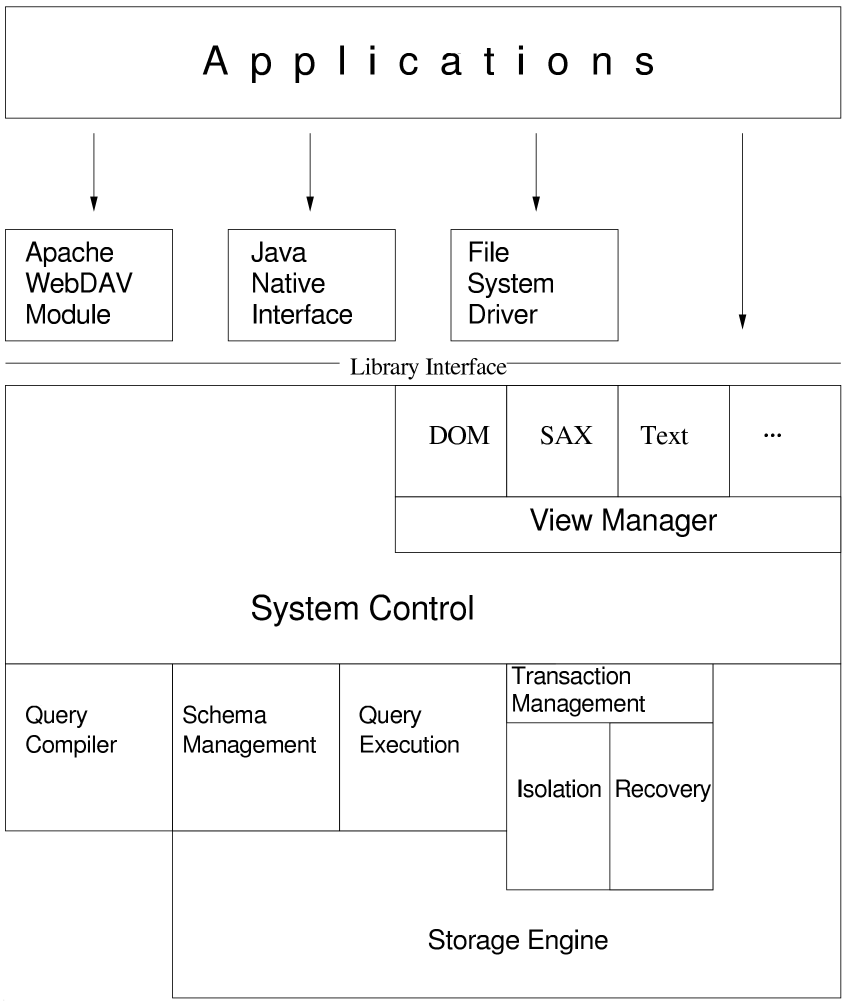

Figure 4.1: Architectural overview

4.2 System Control

The system control component provides a unified facade interface for applications to interact with the XBMS.

Applications work with Natix by sending requests to the *singleton* [30] system control object. Each time an application request is received, system control translates it into an appropriate sequence of invocations of the involved subsystems.

Requests are usually specified by constructing *request objects* (Section 4.2.1) and processed by forwarding the object to system control. Request results that are more complex than simple data types are provided in form of *views* (Section 4.2.2).

4.2.1 Requests

Typical application requests include ‘process query’, ‘abort transaction’ or ‘import document’.

In order to perform such a task with Natix, an application constructs an object of the appropriate request class, whose attributes completely specify the intended action. To invoke the action, a reference to this object is provided as argument to the `process()` method of the system control object, of which there exists only one for each instance of Natix.

When the `process()` method returns, if necessary, the result fields of the request object have been filled with values representing the answer to the request.

Request parameters and results are strongly typed. However, each parameter type can have different representations. For example, a stored document can be addressed in different ways, including a single URL, or the name of the repository and the document ID, or a pointer to the document collection and the document name.

The available request and parameter types are specified using an XML-based syntax, from which C++ classes are generated. This technique is useful to implement bindings for other languages, as explained in Section 4.3.

4.2.2 View Management

In Section 2.4, common application programming interfaces for XML and their properties are presented. The interfaces are not strictly alternatives with the same functionality. Instead, their properties are often complementary, and which API is chosen depends on the application requirements. A simple, performance-conscious application, which needs to read the whole document, may choose libxml2’s SAX binding, while more complex document transformations are better carried out with a DOM interface.

Views

To increase adoption of XBMS, they must facilitate a smooth transition from existing environments to XBMS usage, they must allow developers to choose the best suited API for a given task, and they must be prepared for the introduction of new APIs for XML. Hence, Natix does not use a single, fixed API for document access. Instead, each API is considered just one of many different *views* on the data.

Fragments

In addition, there are different ”sources” of documents. Apart from documents that are stored in the XBMS, this includes documents in the file system, documents produced as query results, and documents received through a network connection. For development convenience, and to reduce the overall complexity of the system, the view concept has to be orthogonal to the document source. By orthogonality we mean that there should be a uniform interface, to open any meaningful view on any document source, for example to open a SAX view on a query result, or a DOM view on a stored document. In Natix, this is enabled by the concept of an XML *fragment*, which is an abstraction representing some XML tree or subtree regardless of its source.

Fragments are addressed using `FragmentDescriptor`s. Requests exist to create `FragmentDescriptor`s for a variety of document sources. Possible source specifications include an OS file name, a Natix `DocumentID`, or a query expression.

Implementation

Additional request types then allow to open and close views for a given XML fragment. Each view type implements some API. There exist views for SAX, DOM, libxml2, C++ streams, and other APIs. The SAX and DOM APIs are implemented on top of the existing Xerces implementations of the APIs, and are binary compatible.

Each combination of document source and view type has a *view manager*, which translates operations on the views into operations on the implementing subsystems, such as the storage engine and the transaction manager. The view manager accesses the *schema* to determine which storage engine objects are involved. It also manages main memory for objects required by the views (such as DOM nodes), may cache created views, and/or share them between transactions.

Apart from its intuitiveness and convenience for the developer (as illustrated in the examples below), the view approach implicitly communicates the application’s access profile to the XBMS. Such access behaviour hints can then be used to optimize performance. For example, when an application requests a SAX view on a stored document, this indicates that the document is going to be scanned completely. Hence, the SAX view manager for stored documents can request a shared lock for the whole document instead of locking individual nodes or subtrees, and then escalating to a document lock.

4.3 Application Architectures and Bindings

XBMS are suited for a wide range of application domains. Their architectures, implementation languages, and protocols differ.

The classic client-server database system will not be the only architecture in which XBMS are used, and C++ is not the only language that is used to work with an XBMS. Below, we explain how Natix is applied in different scenarios.

4.3.1 Library Interface

The most flexible way to use Natix, and the one with the least overhead, is to link to the Natix library and use the control interface (Section 4.2) with C++ function calls.

The Natix API is thread-safe, and it is possible to put the data structures into shared memory, so that several processes can simultaneously work with the same instance.

Interactive Shell

An example for a simple application implemented on top of the C++ library is an interactive shell, one of the standard maintenance tools for Natix. It is a small program which links to the library and can be used to interactively work with document repositories, configure instance parameters such as buffer size, or create and assign new physical media to document collections.

All requests supported by Natix's library interface are specified in a language-independent way (Section 4.2.1). These specifications are used to generate a parser which creates request objects from strings. Requests requiring parameters that are not representable in string form are not supported by the parser.

The parser is used to implement the simple shell program that reads strings from the user, transforms them into request objects, processes the requests, and uses a standard method provided by each request to report its result to the shell's output stream.

4.3.2 Java

Using the *Java Native Interface* (JNI), it is possible to call the C++ library from Java programs [22].

Since the request types are specified in a language-independent way (Section 4.2.1), it is possible to automatically generate Java request classes and the JNI specification which binds the Java methods to C++ functions.

For some of the view types, appropriate Java-specific code needs to be written. Streams in Java, for example, have a different interface from C++ streams, and for other interfaces like DOM it is necessary for performance reasons to cache the Java representation instead of forwarding all Java-function calls to the C++ library.

4.3.3 WebDAV

The Web Distributed Authoring and Versioning Protocol WebDAV [32] is a protocol based on HTTP [28] providing read-write access to document collections.

WebDAV is a de-facto standard used by many tools for the the creation and maintenance of web sites. Supporting WebDAV enables an XBMS to be integrated into such tools without special programming.

WebDAV support for Natix is implemented as a *module* for the Apache Web Server [75]. This module translates WebDAV requests to read and write documents into requests for Natix's library interface.

4.3.4 HTTP

As a side effect, the WebDAV module allows HTTP access to documents stored using Natix. Hence, exporting a Natix repository using Apache is also possible.

4.3.5 File System

Another example for a high-level API is the file system interface [19]. It is implemented as an operating system *fi le system driver* which links to the Natix library. Documents, document fragments, and query results can be accessed just like regular files.

This allows all programs that can work with XML files, such as editors, stylesheet processors, or office applications, to work with Natix without modification.

The repository and document collection hierarchy, as well as the documents' tree structure are mapped into a directory hierarchy, which can be traversed with any software that knows how to work with the file system. Internally, the application's file system operations are translated into Natix requests for exporting, importing, or listing documents.

4.3.6 Others

Since the library interface to Natix is specified in a language-independent way, other language bindings and protocols can be easily created and maintained, as long as there is a way to call C/C++ functions from the languages, or from the servers which implement the protocols.

4.4 Storage Engine

The *storage engine* contains components for efficient XML, index 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.

An efficient storage method optimized for XML is crucial to avoid many issues raised in the previous chapter: reparsing of the document each time it is accessed, storage space overhead, efficient implementation of navigational primitives, and excessive random I/O.

Natix's storage format clusters connected subtrees of the document tree into large records and represents intra-record references differently from inter-record references, to save space. For more details, see Chapter 5.

4.5 Schema Management

Schema management is responsible for the maintenance of the application-visible logical and physical structure of the database. The primitives of schema management allow to group documents into document collections, and to group document collections into repositories.

The application uses the schema management to specify what kind of documents are allowed in document collections, how document collections are distributed onto physical media, and which kind of indexes are maintained for the collections.

Most of the schema information is representable or materialized as XML documents, to provide a unified view on data and schema. The Fragment/View mechanism explained in Section 4.2.2 provides a uniform interface to the schema.

Schema management is dealt with in Chapter 6.

4.6 Transaction Management

The *transaction management* component contains classes that provide ACID transactions. Components for recovery and isolation are located here. These components closely interact with each other and the storage subsystem.

4.6.1 Isolation

The *isolation* component is concerned with the serializability of transactions, while allowing a high level of concurrency.

This component needs to addess the lock granularity problem raised in Section 3.3. We do not elaborate on the topic here, referring to Helmer et al. [42] and Schiele [77] for details.

4.6.2 Recovery

The *recovery* subsystem closely cooperates with the storage subsystem to make transactions atomic and durable. It keeps a log of all updates to allow for transaction undo and restart recovery.

The recovery subsystem must be able to support fine-grained concurrency control, which means that transaction undo must be possible even if two transactions modify documents located on the same disk page, or if two transactions modify parts of the same document.

One example of the introduced XML-specific recovery optimizations is *subsidiary logging* (Section 7.6): Using our XML storage format outlined above, typically more than one node in a clustered subtree is modified by the same transaction, for example during document bulkload. In a conventional system, all the updates would result in individual log records, including log record headers. A large amount of log would be generated, and the log manager would become a system bottleneck. Natix's recovery subsystem tries to combine several operations into a single log record, amortizing logging costs over all nodes of the same physical record. This reduces the recovery overhead tremendously.

Designing a solid recovery subsystem requires attention to many details, as demonstrated in Chapter 7.

4.7 Query Evaluation

Evaluation of declarative queries is divided into two major subproblems. First, a declarative query has to be translated into an execution plan using a query compiler. Second, a powerful yet efficient execution engine has to be built.

Query evaluation for XML is an extensive topic on its own and beyond the scope of this work. We only take a quick glance at its integration into Natix's architecture, and provide selected pointers into the literature.

4.7.1 Query Compiler

Natix's *query compiler* builds execution plans for queries formulated using XPath and XQuery. To decide which physical operators are necessary to access the required data, it heavily relies on the schema information.

As an example for an issue of XML query compilation, we look at plans for XPath expressions. A simple plan uses nested loops to process each location step. However, XPath axes can overlap, creating duplicates. Duplicate elimination is an expensive pipeline breaker. How pipelined plans for a sizeable XPath fragment can be created is discussed in Helmer et al. [43].

To yield optimal performance, it is also possible to integrate XSLT stylesheet processors directly into the database engine [59].

4.7.2 Query Execution

The Natix *query execution* engine (NQE), which efficiently evaluates queries, is described in Fiebig et al. [27].

It is based on a hierarchy of physical iterators [33], which are parameterized using Natix Virtual Machine programs, small assembler-like statement sequences operating on virtual registers.

The physical algebra used by the execution engine includes special construction operators to create XML result documents from queries [25].

4.8 Examples

We provide examples of using Natix from C++ code, implementing the tasks from Section 3.2. Apart from illustrating how to control the basic functionality of Natix, the final

```
int main(void)

{
  NatixInstance instance("sample_instance");
  BeginTransactionRequest beginta;
  instance.process(beginta);

  // open a XML file in file system
  OpenDocumentRequest openreq(beginta.transaction(),
                              "file://catalog_summer_02.xml");
  instance.process(openreq);
  FragmentDescriptor doctoadd=openreq.fragment();

  // import file into Natix
  AddDocumentRequest addreq;
  addreq.transaction(beginta.transaction());
  addreq.inputFragment(doctoadd);
  addreq.collection("CatalogRep","SpecialCatalogColl");
  instance.process(addreq);

  CloseFragmentRequest closedoc(beginta.transaction(),doctoadd);
  instance.process(closedoc);
  CommitTransactionRequest committa(beginta.transaction());
  instance.process(committa);
  return 0;
}
```

Figure 4.2: Importing a document

code sample shows how C++ language features are used to produce terse yet expressive XBMS application code.

4.8.1 Document Import

In Figure 4.2, a system control object is instantiated and connected to a configuration file which specifies the XML base that is to be used (`sample_instance`). A transaction is started by issuing a begin transaction request. Then, a document in the file system is opened using an `OpenDocumentRequest`. A fragment handle for the document in the file system is the result.

An `AddDocumentRequest` is constructed and executed, asking to import the document into a specified collection.

The example is intentionally verbose to explicitly show all involved objects. In the next examples, we show how to use C++ language features to make application code more terse.

4.8.2 DOM Access

In Figure 4.3, we assume that there is an already implemented function `transformTree()`, operating on a Xerces DOM C++ tree. This function can be reused for a document stored in Natix as follows.

The document imported in Section 4.2 is opened, this time not in the file system, but

```
void transformTree(DOM_Node root)
{
  [...]
}

int main(void)

{
  NatixInstance instance("sample_instance");
  BeginTransactionRequest beginta;
  instance.process(beginta);

  OpenDocumentRequest opendoc(beginta.transaction(),
                              "CatalogRep", // repository name
                              "SpecialCatalogColl",  // collection
                              "catalog_summer_02.xml"); // document
  instance.process(opendoc);
  OpenDOMViewRequest openview(beginta.transaction(),
                              opendoc.fragment());
  instance.process(openview);

  transformTree(openview.view()->root());

  CloseDOMViewRequest closeview(beginta.transaction(), openview.view());
  instance.process(closeview);

  CloseFragmentRequest closedoc(beginta.transaction(), opendoc.fragment());
  instance.process(closedoc);

  CommitTransactionRequest committa(beginta.transaction());
  instance.process(committa);

  return 0;
}
/// shorter version:
int main(void)

{
  NatixInstance instance("sample_instance");
  Transaction *ta=BeginTransactionRequest(instance);

  DOMView *dv=OpenDOMViewRequest(ta,
                       "CatalogRep",
                       "SpecialCatalogColl",
                       "catalog_summer_02.xml")).view();

  transformTree(dv->root());

  CloseDOMViewRequest(ta, dv);
  CloseFragmentRequest(ta, f);

  CommitTransactionRequest(ta);
  return 0;
}
```

Figure 4.3: Accessing a DOM view

```
int printCapitals1(Transaction *ta, FragmentDescriptor f)

{
  FragmentDescriptor queryresult;
  queryresult=EvaluateXPathQuery(ta,f,"//city[@capital = ’yes’]").fragment();

  StreamView *streamview=OpenStreamViewRequest(ta,queryresult).view();
  // copy contents of stream view to stdout
  std::cout << *streamview->streambuf();

  CloseStreamViewRequest(ta,streamview);
  CloseFragmentRequest(ta,queryresult);
}

int printCapitals2(Transaction *ta, FragmentDescriptor f)

{
  FragmentDescriptor queryresult;
  queryresult=EvaluateXPathQuery(ta,f,"//city[@capital = ’yes’]").fragment();

  std::cout << queryresult;

  CloseFragmentRequest(ta,queryresult);
}

int printCapitals3(Transaction *ta, FragmentDescriptor f)
{
  std::cout << EvaluateXPathQueryTidy(ta,f,"//city[@capital = ’yes’]").fragment();
}
```

Figure 4.4: Exporting a query to a stream

from its stored location in Natix. Then a DOM view is opened for the document, and the root node of the DOM representation is given as argument to the function. The DOM view implements a binary compatible DOM interface to the stored document, which dynamically loads accessed nodes on demand.

The first `main()` function in the example is again very verbose. However, there exist constructors that not only construct a request object, but immediately call the `process()` function. The second version of the `main()` function shows how this enables the use of temporary request objects to produce more terse code. The code also makes use of the fact that a transaction knows the system control object with which it is associated. As a consequence, only the transaction is required as parameter.

4.8.3 XPath Query

As a final example, we show how an XPath expression can be evaluated and its result can be obtained as a stream (Figure 4.4).

Given the document `f` on which the expression `//city[@captial = ’yes’]` is to be evaluated, and the transaction, an `EvaluateXPathQuery` request is used to create a `FragmentDescriptor` for the query result. For that fragment, a StreamView is opened. This view allows to access the fragment as a C++ stream buffer [47], which can be copied to the standard output stream.

The second version of the function uses an overloaded output operator for fragments, which automatically opens and closes a stream view, making the code more readable.

The final version uses an `EvaluateXPathQueryTidy` request, which is a variant of the `EvaluateXPathQuery` request that automatically closes the created result fragment when the temporary request object is destroyed after the result has been written. This results in a single application code line for a rather complicated process.

Chapter 5

Storage

The people who really run organizations
are usually found several levels down,
where it is still possible to get things done.

–Terry Pratchett

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

In this chapter, we summarize the relevant techniques utilized in Natix's storage engine (up to Section 5.6). The description of the internals includes the important aspect of meta-data management. While treatment in the first subsections is more detailed than necessary for the development of efficient XML storage, the details are fundamental to the design of our recovery subsystem, which is the topic of a later chapter (Chapter 7).

We then elaborate on our novel XML storage method in Section 5.7. After placing it in the context of other approaches to XML storage, we describe the logical and physical models we use, the interfaces to manipulate them, and algorithms for bulkload and updates. A simple way to specify fine tuning parameters for our storage format completes the description. The chapter concludes with an evaluation, comparing the performance of typical RDBMS and OODBMS storage layouts with our novel format.

5.1 Architectural Overview

In Figure 5.1, the different modules of the storage subsystem and their call relationships are shown.

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

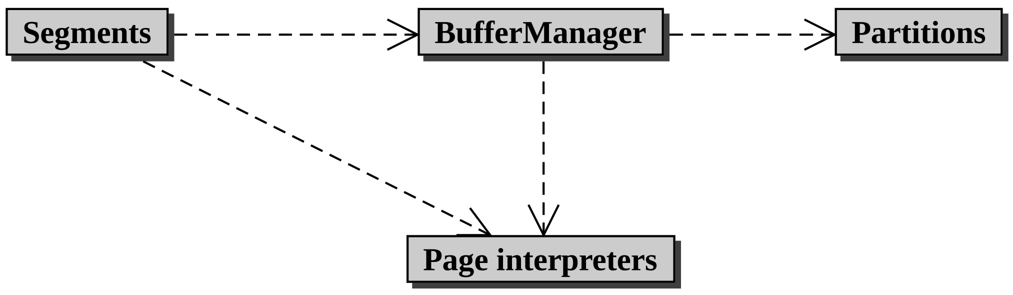

Figure 5.1: Storage Engine Architecture

contents are accessed using *page interpreters*, which are visible not only to the segments, but also to the buffer manager.

5.2 Partitions

Partitions represent an abstraction of random-access block devices.

The classes in the partition hierarchy themselves implement the basic partition interface, which is shown in Figure 5.2. Apart from reading and writing single pages (`readPage()`, `writePage()`), the interface allows to write a consecutive block of pages at once (`writePages()`), and to wait until all write operations are stored on a physical disk platter (`synchronizeWrites()`, necessary for logging, refer to Section 7.4.2). The position of the last read/write operation can be queried using `headPosition()` to allow for efficient I/O scheduling.

Partitions have a logical name (`name()`), which is specified upon creation. The page size (`pageSize()`) and total number of pages (`size()`) are constant for each partition object. In physical object addresses, partitions are identified by *partition number*s, which are 16-bit wide in Natix (`partitionNo()`). Each partition may have up to 2^{32} pages. The pages of a partition are usually addressed using their *page ID*, or *PID*, a combination of partition number and page number in the partition. Apart from translating the partition number to a partition object, no further mapping is required to determine the physical location of a page, given its PID.

Currently, there exist partition classes for Unix files, raw disk access under Linux and Solaris, and C++ iostreams.

5.3 Buffer Manager

The *buffer manager* is responsible for transferring pages between main and secondary memory. In the following, we do not reason why a DBMS needs its own buffer management separate from the operating system (refer to Stonebraker et al. [85] for detailed arguments). Instead, we limit our description to which well-known buffer management

```
class PRT_Partition
{
public:
  virtual void open(BSE_AccessMode m=BSE_READWRITE)=0;
  virtual void close()=0;

  virtual void readPage(PageNo, ptr_t buffer)=0;

  virtual void writePage(PageNo, ptr_t buffer)=0;
  virtual void writePages(PageNo start, uint32 pagecount, ptr_t buffer)=0;
  virtual void synchronizeWrites()=0;
  virtual uint32 headPosition() const=0;

  ccstring_t name() const;

  uint16 pageSize() const;
  virtual uint32 size() const=0;
  PartitionNo partitionNo() const;
};
```

Figure 5.2: Abstract partition interface

```
class BufferFrameCB
{
public:
  const PID &pid() const;
  ptr_t contents() const;
  uint32 fixCount() const;
  bool isDirty() const;
  PGE_PlainPage *page();
  const PGE_PlainPage *page() const;
};

class BufferManager {
public:

  bool fetch(const PID &p,
             BufferFrameCB *&bf,
             PageObjectFactory *fac,
             bool exclusive);
  void touch(BufferFrameCB *bf);
  void unfix(BufferFrameCB *bf);
  bool getFrame(const PID &p,
                BufferFrameCB *&bf,
                PageObjectFactory *fac,
                bool exclusive);
  void invalidateUnfix(BufferFrameCB *);

  void invalidatePartition(PRT_Partition *);
  void flush();

  BufferFrameCB *transientFrame();
  void transientFree(BufferFrameCB *bf);
};
```

Figure 5.3: Buffer manager interface

techniques [23] were selected for Natix.

In addition, we will describe the infrastructure which enables the efficient use of *PageInterpreters* to read and modify the pages' contents (`PageInterpreter`s will be introduced later in more detail, Section 5.4).

Currently there is only a single instance of the buffer manager in the system, to keep the administration effort for Natix low. Multiple instances would require trade-offs on how to distribute the available memory among them. However, the rest of the system is not based on the assumption of a single buffer manager instance. The only assumption hardcoded into the system is that a single segment (Section 5.5) uses exactly one buffer manager instance.

5.3.1 Basic Interface

The buffer manager interface can be found in Figure 5.3.

Each page frame in the buffer has an associated buffer frame control block. This BufferFrameCB contains information about which page is currently loaded into the frame (`pid()`), and where its contents are located in the buffer (`contents()`). The buffer frame also contains the number of current users of the page (`fixCount()()`), and whether it differs from the version stored on disk (`isDirty()`).

The `fetch()` call provides a BufferFrameCB for a given PID and fetches the page from disk. Using `touch()`, the buffer manager is notified that the page was modified. After work on a page has been completed, an `unfix()` call tells the buffer manager that the page is not in use any more and may be replaced. The `getFrame()` call allows for assignment of buffer space to a page without loading its current disk contents (to avoid read access for new pages). Symmetrically, `invalidateUnfix()` discards dirty pages from the buffer without writing their contents to disk (to avoid writing deallocated pages).

If a partition has been dropped and is not used any more, any remaining pages are removed from the buffer using `invalidatePartition()`. A call to `flush()` writes all modified pages to disk.

During query processing, operators used to answer the query often build some specialized main memory structures for efficient processing, for example hash tables for grouping. We want to avoid keeping a region of memory reserved for such operators even when they are not active, and to maximize the amount of memory available to the buffer manager. Therefore, such operators can allocate necessary memory from the buffer manager by calling `transientFrame()`, which returns a buffer frame that is not associated with any disk page. Such frames can be returned with `transientFree()`.

5.3.2 Page Interpreter Sharing

The buffer manager connects each page to a page interpreter. All page accesses are encapsulated by these page interpreters, and a page's contents are never accessed directly.

A page interpreter is initialized and assigned only when the page is brought into memory and afterwards shared by all users of the page until it gets replaced. This way, we avoid that every user of a page needs to create and destroy its own page interpreter, resulting in initialization and memory management overhead.

Only space for a page interpreter base class pointer is reserved in the frame control block, to avoid a dependency of the buffer manager to the specifics of the different page interpreter types, and their memory management. This pointer is initialized using *PageInterpreterFactories*, which are specified by the buffer manager users when fetching pages, and called by the buffer manager only if a page is brought into memory. If a page is already in memory, the existing page interpreter is reused.

The buffer manager notifies the page interpreters if certain events for the page occur, including an impending page flush or page replacement (More details can be found in Section 5.4).

5.3.3 Implementation

The following describes some crucial details of our buffer management implementation. While our techniques are based on established knowledge [23, 38, 66], we also highlight some of the differences.

Address translation

Translation of PIDs to buffer frames is performed using a hash table. It is possible to provide a hint in form of a buffer frame pointer from an older request for the same page, which is used without access to the hash table if it still contains the page.

Page Replacement

When selecting a page to be replaced by a newly requested page, Natix employs the Least-Recently-Used strategy in its Least-Recently-Unfixed variant [23]. An LRU queue is maintained, and when `fixCount()` indicates that the last user has unfixed the page, the associated buffer frame is inserted at the front of the queue.

Additional hints may be given during `unfix()` on whether a page reuse is unlikely in the near future, for example during a scan operation. In such cases the page is queued for prompt replacement.

Before replacing a dirty page, it must be written to disk. First, the `prepareWrite()` function (refer to Section 5.4) of the associated page interpreter is called, and then the write operation is performed. Afterwards, it is verified that the page still needs to be written, because the page interpreter may detect during `prepareWrite()` that the page contents is invalid and need not be written to disk (Refer to Section 7.5.4 for details on how this can happen).

Synchronization

The buffer manager also synchronizes page access by multiple threads. Each buffer page is synchronized by a data latch, which is part of the buffer frame control block for fast access. *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 hash table mapping page IDs to memory buffer frames is synchronized by one mutex for each hash bucket, allowing for efficient symmetric multiprocessing (SMP) with several CPUs, where several processors want to look up page IDs and modify the buffer's contents in parallel.

To avoid deadlocks, data page latches are never requested while a hash bucket mutex is held. In addition, if a buffer frame is assigned a new PID and as a consequence changes buckets, the modified bucket with the smaller index is always locked first. Otherwise, a deadlock would occur when another thread tries to move a frame between the same buckets in the opposite direction.

The data page latch synchronizes access to the contents and the page interpreter. The hash bucket mutexes synchronize the contents of the frame control blocks in the respective bucket.

The above roughly follows Mohan et al. [66], largely because we wanted to be able to implement the ARIES recovery protocol on top of our storage engine. Synchronization of the list of dirty pages is different from [66], where a separate synchronized dirty page table is used to mark pages as modified. Our approach is more lightweight because it does not use extra data structures and synchronization objects. It also makes checkpointing more efficient (see Section 7.4.6).

While the dirty flag is a member of the frame control block, it is synchronized by both the page latch and the hash bucket mutex. If a page is modified, the dirty flag must be set while the exclusive data page latch is still held. Before a page gets written back to disk, the dirty flag is cleared while holding the hash bucket mutex. This asymmetry prohibits multiple threads from attempting to flush the same page concurrently. While this could also be achieved by requesting an exclusive latch on the page before clearing the dirty flag, there is no need to block read access to the page while it is flushed to disk.

5.4 Page Interpreters

The *page interpreters* are used to encapsulate access to the pages' contents. There exist page interpreters for every type of physical page organization, for example slotted pages, B-Tree internal nodes, B-Tree leaves, and XML document storage pages.

The page interpreter classes form a class hierarchy, the base class of which (Figure 5.4) provides support for the common protocol the interpreters use with the buffer manager and the segments. From this base class, one or more classes are derived for every persistent data type.

At first glance, 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. However, the page interpreters in Natix are not merely a way to divide-and-conquer persistent data structure implementation, but are exposed as first-level citizens of the storage subsystem. This allows direct communication between other storage objects and the page interpreters.

```
class PGE_Page {
 public:

  virtual void attach(ptr_t c);
  void pid(const PID &pid);
  virtual void detach();

  virtual void attachAndFormat(ptr_t c);
  virtual void unformat();

  virtual void prepareRead(GloCB *);
  virtual void prepareWrite(GloCB *);

  bool isInvalid() const;

  virtual void format();

  void recordedFreeSpaceInfo(uint8 fsi);
  uint8 recordedFreeSpaceInfo() const;

};
```

Figure 5.4: Page interpreter base class

As it turns out, the exposed page interpreters tremendously simplify implementation of metadata management (Section 5.6) and the recovery subsystem (Chapter 7).

The methods of the page interpreter base class allow the buffer manager to associate a page with a buffer location after it has been loaded (`attach()`, `pid()`), and to remove this association before the page is replaced (`detach()`). After reading and before writing a page, `prepareRead()` and `prepareWrite()` are called to calculate/verify checksums or check bits (if page reads and writes do not correspond to atomic physical operations [62]) and possibly do data-type dependent housekeeping. Page interpreters may also indicate, using `isInvalid()`, that they have been invalidated and need not be written back to disk.

Apart from the buffer manager protocol, the page interpreter base class factorizes infrastructure needed by all segments. Pages need to be initialized, or *formatted*, before they are used for the first time, and some of the page header initialization is common to all data types (`format()`). Likewise, all segments need free space management structures to avoid excessive I/O while searching for space for new objects. Although the segments may use different schemes to manage their free space, a common storage location for free space information about a page simplifies the implementation, as explained in Section 5.6. The `recordedFreeSpaceInfo()` functions allow to manage such information.

5.5 Segments

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

Examples for segment types include Slotted Page Segments, which manage variable-

```
class SEG_Segment {

protected:
    virtual PID allocatePage(uint32 hintpages=0, uint32 flag=ALLOC_NONE);
    virtual void freePages(PID page, uint32 nopages=1);
    virtual bool canAllocatePages(uint32 nopages);

    void fetchShared(const PID &pid, BufferFrameCB *&bf) const {
    void fetchExclusive(const PID &pid, BufferFrameCB *&bf) {
    void getFrameExclusive(const PID &pid, BufferFrameCB *&bf) {
    void unfixShared(BufferFrameCB *bf) const;
    void unfix(BufferFrameCB *bf, bool dirty);
    void touch(BufferFrameCB *bf);

    virtual uint8 fsiBitsFor(PGE_PlainPage *) const;

    bool hasLogicalPageNos() const;
    LogicalPageNo logicalPageNo(PageNo physical) const;
    bool hasLogicalPageNo(PageNo physical) const;
    PageNo physicalPageNo(LogicalPageNo logical) const;
public:
    class iterator;
    iterator begin(uint8 fl,uint8 fh) const;
    iterator begin() const;
    iterator end() const;
    iterator find(const PID &pid) const;
};
```

Figure 5.5: Interface for segment base class

sized records, BTree segments which allow associative access to records based on keys, and XML segments which store XML document trees.

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.

The segment classes are organized as a class hierarchy, whose base class 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.

To illustrate the typical properties of a persistent data type interface, two of the most important segment types are discussed after briefly presenting the segment base class.

5.5.1 Segment Base Class

The segment base class primarily manages the association between the segment and its constituent pages.

This association is maintained using an extent-based system [95], which divides a partition into consecutive page groups (*extents*) of variable size (Sections 5.6.3 and 5.6.4). Each extent is either associated with a segment, or free. Using calls to `allocatePage()` and `freePages()`, a concrete segment type can request a fresh page to write to or discard old pages that no longer hold any data. The segment base class will request new extents from

```
class SEG_DirectMapSegment : public SEG_Segment
{
public:
  class iterator;

  bool  allocateSlot   (cptr_t handle, DirectMapSlotNo& slotNo, uint8& unique);
  bool  allocateSlot   (cptr_t handle, DirectMapSlotNo& slotNo, uint8& unique,
                        DirectMapSlotNo hint);
  bool  freeSlot       (DirectMapSlotNo slotNo, uint8 unique);
  uint8 unique         (DirectMapSlotNo slotNo);
  void  recycleSlot    (DirectMapSlotNo slotNo);

  bool  updateSlot     (DirectMapSlotNo slotNo, uint8 unique, cptr_t newHandle);
  cptr_t retrieveSlot  (DirectMapSlotNo slotNo, uint8 unique,
                        ptr_t target);

  bool isAllocated     (DirectMapSlotNo slotNo);
};
```

Figure 5.6: Interface for persistent arrays

the partitions free extent pool (Section 5.6.4) when all pages associated with a segment are allocated. The sizes of these new extents are governed by a grow specification given at segment creation time.

The segment base class also encapsulates the buffer manager functionality, to allow concrete segment types to export interfaces that directly operate on the buffer contents.

Intra-segment free space management is performed using a separate data structure called Free Space Inventory (FSI), which records for each page on a partition whether it is formatted and how much free space it contains. The segment base class automatically performs maintenance of the free space inventory for derived segment classes. Only refinement of the `fsiBitsFor()` function is required in the derived segment. More on FSI management can be found in Section 5.6.5.

Optionally, the segment base class can maintain a logical page number mapping, which provides a dense, segment-local page numbering that is robust against a reorganization of the segment.

Enumeration of all pages in a segment can be carried out using an STL-like iterator [47].

5.5.2 Persistent Arrays

Figure 5.6 shows an excerpt of a very simple persistent data structure interface, the *DirectMap segment*. The DirectMap segment implements a persistent array of fixed-size objects which can be addressed using an integer index, which is also called *slot number*. DirectMap segments are often used to implement object IDs, where a logical ID needs to be mapped to an object's physical location [24].

In addition to updating and retrieving array elements, or *slots*, the segment also provides memory management for slots. The caller can allocate and release slots, and freed array slots are reused in future allocations. To make it possible to detect dangling references to freed slots, each array element is assigned a small integer *unique value* (accessible via `unique()`) that is used as part of the address. Every time a slot number is reallocated, the unique number is increased. After 2^8 reallocations, the slot cannot be allocated again. By calling `recycleSlot()`, a slot number may be reused by resetting its unique number.

Access to slot contents is based on copying complete slot contents from (`updateSlot()`) or to (`retrieveSlot()`) caller-allocated buffers. This keeps the interface simple, but incurs a performance penalty which is tolerated because the managed objects in direct map segments, such as the physical object references mentioned above, tend to be small.

The direct map segment's implementation is straightforward. Dividing the slot number by the number of slots per page determines the logical page number of a slot is determined, which is then mapped to a PID using the segment base class' logical page number map. Maintenance of unique number, slot contents and free slots on each page is done by the DirectMap page interpreter.

5.5.3 Slotted Page Segments

The most important persistent data type in Natix is the variable-length record. Collections of such records are managed in *Slotted Page Segments*.

Natix basic records have the page size as their upper size limit. This is sufficient for most uses, and allows for an optimized interface, as shown below. In all cases where large records would have been necessary in Natix, special segment types were implemented instead which take the semantics of the data stored in the records into account. The primary example for this is the XML storage segment, which will be detailed in Section 5.7.

Basic Slotted Page Segments

An excerpt of the Slotted Page Segment interface is shown in Figure 5.7. There appears to be no uniform terminology for slotted pages and related techniques, despite their popularity in DBMS implementation [2, 38, 87, 95]. We call pages with slot tables *Slotted Pages*, and use the term *RID* for IDs of records on such pages. By *TID concept*, we mean the concept of overflow records with stable RIDs (see next section).

Records are addressed using *record IDs*, or RIDs, which consist of a PID and a slot number on the page. Slot numbers are mapped to the intra-page location of the record by the page interpreters, as explained in Section 5.4. Variable-length records are used in a lot of different contexts in the system, and performance of the slotted page segment heavily influences overall system performance. To allow efficient access to the records, users of the slotted page segment can directly operate on the buffer version of the records, without expensive copy operations or representation changes.

This has consequences for the interface. First, it is necessary to allow addressing of records not only with their RID, but also using their buffer location. This is done using the buffer frame from the buffer manager and the slot pointer from the respective page interpreter. A proliferation of similar method calls is the result, since the storage system also allows to give hints for the placement of new records, and since it should be possible to first address a record with its RID, and subsequently operate directly on the buffer. Second, there must be a way to notify the segment that the caller does not need direct access for a certain frame any more, which is the reason for making the `unfix()` method `public`.

Not shown in Figure 5.7 are the calls that allow to partly modify a record (insertIntoRecord, deleteFromRecord etc.).

TID Segments

A derived variant of the Slotted Page Segment is the *TID Slotted Page Segment*, which implements the *TID concept* of record addressing [2, 38, 87, 95]. In a TID Slotted Page Segment, a record may grow although there is no more space on its page. In this case, the record is moved to a different page. Even for moved records, the original RID is still a valid way to identify the record, and the original page contains a reference to the new location.

The same interface as in the basic Slotted Page Segment is used, and the complete redirection mechanism is hidden from the caller. Special calls and flags exist to allow control over when buffer frames are unfixed during redirection and record relocation.

5.6 Physical Metadata Management

This section discusses Natix's physical metadata management. *Physical metadata* is data which describes how the storage management objects such as segments themselves, are made persistent.

The most important aspect of metadata management is free space management, including search for free space on a page, search for a free page, or search for a free extent of pages. It is difficult to overemphasize the importance of efficient metadata management[1].

Our notion of physical metadata does *not* include mapping of application-controlled structures such as document collections, indexes, and document attributes such as names and identifiers to storage manager structures. We collectively call this kind of data the *Physical Schema* and deal with its management in Chapter 6.

Below, we first establish a metadata model to define more precisely what we mean by physical metadata. The implementation of its components are explained in the remainder of the section.

5.6.1 Object Model

Figure 5.8 shows a UML diagram depicting the classes of objects metadata management has to deal with.

The following subsections explain how the associations in the diagram are materialized on physical storage, as far as this is not dependant on specific segment implementations.

[1]Nearly every time some update workload exhibited unexpected performance behaviour in Natix, we found that metadata processing dominated the total required time. Simple changes in metadata management corrected the situation.

```
class SEG_SlottedPageSegment : public SEG_RecordSegment
{
public:
  enum  UpdateFlag;
  bool insertRecord(ptr_t content,
                    uint16 size,
                    RID &rid,
                    uint16 sizehint,
                    uint32 flag);
  virtual bool insertRecord(ptr_t content,
                            uint16 size,
                            BufferFrameCB *&bf,
                            PGE_RNAL_SlottedPage::Slot *&sl,
                            uint16 sizehint,
                            uint32 flag);
  virtual bool insertRecordAt(const RID& hint,
                      ptr_t content,
                      uint16 size,
                      RID &rid,
                      uint16 sizehint,
                      uint32 flag);
  virtual bool insertRecordOnFrame(BufferFrameCB *&bf,
                           ptr_t content,
                           uint16 size,
                           PGE_RNAL_SlottedPage::Slot *&sl,
                           uint16 sizehint,
                           uint32 flag);
  virtual bool insertRecordOnFrame(BufferFrameCB *hint,
                           ptr_t content,
                           uint16 size,
                           BufferFrameCB *&bf,
                           PGE_RNAL_SlottedPage::Slot *&sl,
                           uint16 sizehint,
                           uint32 flag);
  virtual void deleteRecord(const RID &);
  virtual void deleteRecord(BufferFrameCB *bf, PGE_RNAL_SlottedPage::Slot *sl);
  virtual bool updateRecord(const RID &rid,
                    ptr_t newcontent,
                    uint16 size,
                    uint32 flag);
  virtual bool updateRecord(BufferFrameCB *&bf,
                    PGE_RNAL_SlottedPage::Slot *&sl,
                    ptr_t newcontent,
                    uint16 size,
                    uint32 flag);
  virtual bool updateRecord(BufferFrameCB *bf,
                    PGE_RNAL_SlottedPage::Slot *sl,
                    ptr_t newcontent,
                    uint16 size,
                    BufferFrameCB *&newbf,
                    PGE_RNAL_SlottedPage::Slot *&newsl,
                    uint32 flag);
  virtual void locateRecord(const RID& rid,
                            BufferFrameCB*& bf,
                            PGE_RNAL_SlottedPage::Slot*& slot,
                            bool fetchExcl);
  virtual RID getOriginalRID(BufferFrameCB*& bf,
                             PGE_RNAL_SlottedPage::Slot*& slot);

  virtual void unfix(BufferFrameCB *);
};
```

Figure 5.7: Interface for slotted page segments

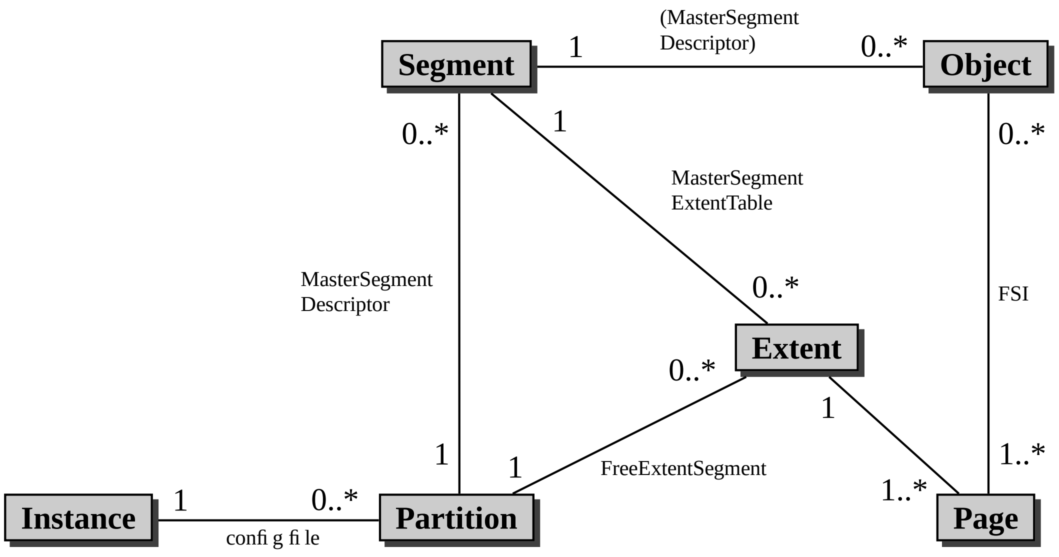

Figure 5.8: Metadata object model

First, we explain how partitions are grouped into Natix *instances*. Then, we describe the *Master Segment*, which contains information about all the segments of a partition, and how the extents belonging to each segment are stored. The Master Segment for some segments also contains information necessary to operate on the segments object collection, for example a B-Tree root node pointer. The *Free Extent Segment* is described next, which manages all the extents on a partition that do not belong to a segment. The section is concluded by a discussion of the *Free Space Inventory* (FSI), which aids the segments in distribution of objects on pages, by storing information about which pages do already contain objects and how much free space they have left.

5.6.2 Partition Information

An *Instance* of Natix is a set of partitions with pairwise different partition numbers, which satifies the condition that all physical object references stored on any partition in the set refer to other partitions in the set.

An instance is specified by an external configuration file which lists all partitions, their types, and physical locations.

Every time a new partition is created or destroyed using the partition manager, the configuration file is updated.

5.6.3 Master Segment

Every partition contains a *Master Segment*. This is a TID Slotted Page Segment whose records describe the segments available on the partition.

Records in the master segment are of one of three record types: segment descriptors, extent tables, or page maps.

Segment Descriptors

Each *segment descriptor* describes the attributes of one segment, such as the segment's name, type, and whether it is recoverable. It also contains references to the Extent Table and Page Map of the segment.

The RID of a segment descriptor may be used as a unique *SegmentID*, and segment descriptors are never moved because they only contain fixed-size attributes.

The Master Segment always contains page 0 of a partition, and the Master Segment descriptor has the RID $(p, 0, 0)$, where p is the partition number. This allows to mount a partition without having to search all of its pages for the master segment.

Different segment types may have extra attributes in the segment descriptor which are necessary to access their object collections. For example, a DirectMap segment stores its slot size in its descriptor, and a B-Tree needs to remember the root node address.

Extent Tables

Extent Table records are used to materialize the association between a segment and its pages.

Each extent table consists of a sequence of pairs (s, l), each describing an extent of length l starting at page s which belongs to the segment. The extents are sorted by the start page number. This not only allows efficient extent table modifications, but also makes it easy to scan all constituent pages of a segment in physical order.

The extent tables are of variable size and may have to be moved to separate pages when they grow. The TID concept handles such relocations.

During insertion and removal of extents, entries are merged if possible to keep the table small.

Currently, the size of an extent table in Natix is limited to the page size. For 8K pages, this means that a segment may only comprise 512 extents. Since newly acquired extents for a segment get larger with time, this is not a severe limitation. It would also be simple to devise a multi-page B-Tree like management structure for extent tables, should the need arise.

Page Map

If dense, segment-local logical page numbers are required for a segment, a logical page number map is also stored. The order of pages in the Extent Table cannot be used for logical page mapping, as extents are removed and inserted, causing new pages to appear before older pages in the extent tables.

The page number map contains pairs (l, p), which mean that the logical pages starting at l are mapped to physical pages starting at p. The table is ordered by increasing logical page number, allowing binary search for a logical page number. The size of each physically contiguous block of logical page numbers can be determined by subtracting the value of l from the l value in the following entry.

The remarks about the size limit of extent tables also apply to page number maps.

5.6.4 Free Extent Segment

The *Free Extent Segment* maintains all the extents on a partition that currently do not belong to any segment.

The Free Extent Segment is organized as a regular segment whose extents are all free extents of the partition. The contents of the segment's pages are not used. The interface allows to efficiently find and remove an extent of a given size from the segment, and to add extents to the segment once they are removed from other segments.

5.6.5 Free Space Inventory

To facilitate intra-segment free space management, a *Free Space Inventory* (*FSI*) is used. This special segment type allows to store and quickly access a small (4-Bit) value for every page in a partition. For a partition with n pages and page size p, only $\frac{n}{2p}$ pages are needed to store the complete FSI. This tremendeously reduces I/O overhead for a free space search within the segment. Most existing DBMS possess such a data structure [65, 55].

All of the FSI management is handled by the segment class hierarchy. No other system modules are involved. The actual meaning of the 4-Bit-values depends on the segment type. Usually, the FSI at least contains information about whether a page has already been formatted, or is still uninitialized.

We will now give some details on the FSI Segment itself, then describe the FSI infrastructure provided in the segment base class, and finally exemplify FSI usage in a concrete segment type, the Slotted Page Segment.

FSI Segment Interface

The *FSI Segment* has an array-like interface which allows to read and write FSI values for any given page. Apart from simple read/write access, methods are provided to search a range of pages for ones with a specified range of FSI values, and to atomically test-and-set FSI values.

Automatic Maintenance

The segment base class simplifies FSI management for concrete segment types by providing a framework (using the Strategy pattern described in Gamma et al. [30]) which only makes it necessary to refine two small methods to keep the FSI in sync with the data pages' contents.

The first method to refine is the `fsiBitsFor()` method from the segment base class (Section 5.5.1). Given a page interpreter, a concrete segment type must return the FSI value it wants to use for that page.

The other method to refine is the initialization method of the segment's PageObjectFactory. It is called by the buffer manager whenever a page was loaded into the buffer and an initialized page interpreter is needed for that page. Apart from providing the interpreter pointer as described in Section 5.3, the factory must initialize the `recordedFreeSpaceInfo()` value of the page interpreter (Section 5.4) with the current FSI value of the page. To avoid extra I/O, the current value is not read from the FSI segment, but calculated from the current page state using `fsiBitsFor()`.

After a page was modified and `unfix()` is called, the segment base class checks whether the newly calculated `fsiBitsFor()` result is different from `recordedFreeSpaceInfo()`. If so, the FSI segment is updated. This mechanism already provides the necessary synchronization. Since a page update requires an exclusive latch, the order of FSI updates is the same as the order of the original updates.

Allocation and Deallocation

To avoid unnecessary I/O, it is desirable to know which pages of a segment actually contain meaningful data. When a page is used for the first time, it is unnecessary to load it into memory, any buffer frame can be assigned to it instead. Also, if no relevant data is on a page any more, it is not necessary to write it back to disk.

While each concrete segment type could use its own scheme to detect properly formatted pages and meaningful pages, they currently all use the facilities offered by the segment base class.

The default scheme reserves two values from the 2^4 possible FSI values for each page. The value 0 designates a page that is not used and may not be considered formatted. The value 1 is used for a page that is in the process of being allocated and formatted, but may not be used yet. All other values may be used by the concrete segment implementation as desired.

Segment growth When the `grow()` method of a segment is called and a new extent is added to the segment, all FSI values for the new pages are initialized to 0.

Also, the new extent is rembered as the `freshExtent` in the segment's main memory object.

Allocation When a new page is requested by a call to `allocatePage()`, the segment looks for an unallocated page. The pages described by the segment's `freshExtent` are searched first. Then, the extents in the extent table are searched from last to first.

For each extent, the FSI segments range search is employed to find a page with FSI value 0. The FSI segments `testAndSet()` call is used to set it to 1, indicating that it is going to be formatted. If this call fails because some other thread is trying to allocate the page, the search is continued. If no page is found, a new extent is requested from the free extent pool. If the page was taken from `freshExtent`, the `freshExtent` is shrinked accordingly. The PID of the newly allocated page is returned.

A concrete segment implementation may then call `getFrame()` to obtain a buffer frame for the page, and format it. The 1 value in the FSI prohibits any other thread to get a frame and format the same page in between the first thread's `allocatePage()` and `getFrame()` calls, avoiding extra latches or locks to synchronize allocation, as needed for example by Mohan et al. [65]. If there was no special value for allocated, but unformatted pages, then other threads might think the page is already formatted, and `fetch()` it before the original thread had a chance to `getFrame()` and `format()` the page.

The domain of the `recordedFreeSpaceInfo` field is a superset of the actually assigned FSI values. Currently, Natix uses 4-Bit FSI values and `recordedFreeSpaceInfo` is 8 bit wide. When formatting a new page, the `recordedFreeSpace` info of the page interpreter is set to a value outside the range normally used by FSI management. This causes the FSI update mechanism to trigger when the newly allocated page is unfixed, and to update the FSI from 1 to the proper value. The FSI value is not updated until the page is unfixed and the allocating thread has completed its work on the new page. This avoids blocking other threads which would otherwise find the newly formatted page and try to insert records on it while the original thread is still updating it. This is especially advantageous in SMP systems.

Deallocation When a delete operation leaves behind a completely empty page, the segment may set the page interpreter's invalid flag (Section 5.4). When the `unfix()` call is invoked for an invalid page, it's FSI value is set to 0 and the buffer manager's `invalidateUnfix()` routine is called, which drops the page from the buffer without writing it.

Shrinking segments Before a segment's extent is returned to the Free Extent Segment, all its pages' FSI values are set to 1, to prevent other threads from allocating pages from the extent which is currently being removed.

Slotted Page Segment Free Space Management

The Slotted Page Segments use a simple mapping from free space on a page to FSI values. As explained above, if the page is free, the FSI value is 0. If the page is being allocated, the value is 1. If a page is formatted and more than half of the page is free, the FSI value is 2. If less than half of the page is free, but at least one fourth of the page is free, 3 is used. If at least one eigth of the page is free, 4 is used, and so on. This mapping allows to reflect large and small amounts of free space with suitable accuracy.

When a new record has to be inserted, the segment first checks the provided hint page (Figure 5.7) whether there is enough space. If not, the segments extents are scanned in last-to-first order, using the FSI segment to search for a first-fit, i.e. a page which has at least as much free space as the record size. First-fit is employed to reduce the I/O overhead until a page is found. The designated page is fetched and an attempt to insert the record is performed. If it is not successful because some other thread has used the free space, the search is continued.

The attempt to insert the record rechecks the FSI information for the page after the `fetch()` call, because it is possible that the page was deallocated in between the FSI search and the successful `fetch()` call. In this case, the page is immediately dropped from the buffer and the search continues.

The FSI search is a hot spot of the system, and all segments of a partition compete for the pages of the same FSI segment. To reduce contention the slotted page segment maintains a small *FSI cache*, which contains the FSI values of the most recently updated pages. Before searching the FSI segment, this segment-local cache is searched. When searching the cache, a best-fit search is employed that starts with the largest matching FSI value and successively decreases the desired FSI value until a page is found. This is done because the segment can expect the pages in the FSI cache to be in the buffer as they have been referenced not long ago.

While a page is fixed, it does not appear in the FSI cache, to prevent processors in SMP systems from simultaneously trying to access the same page, causing one of them to block.

Usage of the FSI cache not only reduces the load on the FSI segment, but also increases locality of page accesses, as it selects pages for update that have recently been in use.

Callers of the Slotted Page Segment may limit the free space search effort by providing flags to the update calls. For example, the caller can request that a new page is to be allocated for a new record, without doing free space search. It is also possible to limit the search to the FSI cache before allocating a page, instead of performing a full-blown search in the FSI segment.

5.7 XML Storage

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

Flat streams In this approach, the document trees are serialized into byte streams, for example by means of a markup language. For large streams, some mechanism is used to distribute the byte streams on disk pages. This can be a file system, or a BLOB manager in a DBMS [4, 14, 52]. This method is very fast when storing or retrieving whole documents or big continuous parts of documents. Accessing the documents' structure is only possible through parsing [1].

Metamodeling A different method is to model and store the documents or data trees using some conventional DBMS and its data model [20, 29, 49, 56, 79, 82, 89, 91].

In this case, the interaction with structured databases in the same DBMS is easy. On the other hand, reconstructing a whole document or parts of it is slower than in the previous method. Other representations require complex mapping operations to reproduce a textual representation [82], even duplicate elimination may be needed [20].

Mixed In general, the meta-modeling approach introduces additional layers in the DBMS between the logical data and the physical data storage, slowing the system down. Consequently, there are several attempts to merge the two ”pure” methods above. In *redundancy-based approaches*, to get the best of both worlds, data is held in two redundant repositories, one flat and one metamodeled [96] (A similar approach is also proposed in Florescu et al. [29] to speed up document export). Updates are propagated either way, or only allowed in one of the repositories. This allows for fast retrieval, but leads to slow updates and incurs significant overhead for consistency control. In *hybrid approaches*, a certain level of detail of the data is configured as threshold. Structures coarser than this granularity live in a structured part of the database, finer structures are stored in a ”flat object” part of the database [7].

Natix native storage Natix uses a novel **native** storage format with the following distinguishing features: (1) Subtrees of the original XML document are stored together in a single (physical) record (and, hence, are clustered). Thereby, (2) the inner structure of the subtrees is retained. (3) To satisfy special application requirements, their clustering requirements can be specified by a *confi guration matrix*. Performance impacts of different clusterings are evaluated in Section 5.8.

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

5.7.1 Logical Object Model and Segment Interface

In the interface and implementation of our XML storage method, we use a simple labelled tree model similar to existing semi-structured data models [13, 56, 70]. Our model simplifies the implementation of our segment type and increases performance, as we replace some string comparisons by integer comparisons. Since the mapping from XML to our internal model is a simple injective function, there is very little overhead.

After describing our logical object model and the segment interface, we will show how to map XML document nodes and tag names onto our model.

Logical Document Object Model

The XML segment's interface allows to access an unordered set of trees. New nodes can be inserted as children or siblings of existing nodes, and any node (including its induced subtree) can be removed.

The individual documents are represented as ordered trees with non-leaf nodes labeled with a symbol taken from an alphabet Σ_{Tags}. In the current implementation, we use the set of integers $0 \ldots 2^{16} - 1$ as Σ_{Tags}.

Leaf nodes can, in addition to a symbol from Σ_{Tags}, be labeled with arbitrarily long strings over a different alphabet. In the current implementation, leaf nodes may be labeled using Unicode strings.

Segment Interface

The interface used to access the labeled tree forest in an XML segment is shown in Figure 5.9.

Nodes can be addressed using a *Node ID*, or *NID*, which is a physical identifier consisting of a page number, a slot number and an offset into the physical record. NIDs are not stable against modification of the document tree.

To work with documents, the segment provides *TreeNodeHandles*, which represent references to tree nodes. The data type used for tree labels is called `DeclarationID`. Apart from navigation inside a tree, and content access, the segment also supports enumeration of all documents in the forest using STL-like iterators [47].

Upon creation of a new document (using the first form of `create()`), the caller has to assign a logical document ID which can then be efficiently retrieved for every node. The second form of `create()` adds a node to an existing document by providing a reference node and a relative insertion location (add as last child, as right sibling or as left sibling of the reference node). The implementation of this function is detailed in Section 5.7.5. If a document is modified, all existing *TreeNodeHandle*s for the same document become invalid, with the exception of the handle used to denote the point of insertion and the handle of the newly created node.

5.7.2 Mapping XML to the Logical Object Model

A small wrapper class is used to map the XML model with its node types and attributes to the simple tree model and vice versa. A sample tree for a document fragment is shown in Figure 5.10.

Mapping XML Document Nodes to Logical Nodes

Elements are mapped one-to-one to tree nodes of the logical data model. Attributes are mapped to child nodes of an additional *attribute container* child node, which is always the first child of the element node the attributes belong to. Attributes, PCDATA, CDATA nodes and comments are stored as leaf nodes, using reserved integer values as node label.

External entity references are expanded during import, while retaining the name of the referenced entity as a special internal node.

Mapping XML Tags to Tree Labels

The wrapper uses a separate segment to map tag and attribute names to integers, which are used as Σ_{Tags}. All the documents in one XML segment share the same mapping, which

```
class SEG_XMLSegment : public SEG_SlottedPageSegment
{
public:
  class TreeNodeHandle ;

  TreeNodeHandle open(NID, bool exclusive);
  static void release(const TreeNodeHandle &n);

  NID nid(const TreeNodeHandle &) const;
  const DocumentID &documentID(const TreeNodeHandle &) const;

  bool isValid(const TreeNodeHandle &) const;
  bool isEmpty(const TreeNodeHandle &) const;
  bool isLiteral(const TreeNodeHandle &) const;
  bool isLargeLiteral(const TreeNodeHandle &tnh) const
  TreeNodeHandle child(const TreeNodeHandle &);
  TreeNodeHandle child(const TreeNodeHandle &, uint32 n);
  TreeNodeHandle lastChild(const TreeNodeHandle &);
  TreeNodeHandle next(const TreeNodeHandle &);
  TreeNodeHandle nextPreorder(const TreeNodeHandle &root,
                              const TreeNodeHandle &tnh);
  TreeNodeHandle nextPreorder(const TreeNodeHandle &root,
                              const TreeNodeHandle &tnh,
                              int32 &navigation);
  TreeNodeHandle prev(const TreeNodeHandle &);
  TreeNodeHandle parent(const TreeNodeHandle &tnh);
  void markModified(const TreeNodeHandle &tnh)
  bool remove(TreeNodeHandle &);
  DeclarationID declarationID(const TreeNodeHandle &tnh) const
  uint32 contentSize(const TreeNodeHandle &tnh);
  ptr_t contents(const TreeNodeHandle &tnh) const;
  void contents(const TreeNodeHandle &tnh, ptr_t dest);

  class documentIterator ;
  documentIterator beginDocuments(bool exclusive, uint32 prefetch=0);
  documentIterator endDocuments();

  TreeNodeHandle create(const DocumentID &docid,
                        DeclarationID id,
                        bool leaf,
                        uint32 contentsize,
                        cptr_t content,
                        uint32 sizehint);

  TreeNodeHandle create(TreeNodeHandle &,
                        InsertionLocation l,
                        DeclarationID,
                        bool leaf,
                        uint32 contentsize,
                        cptr_t content,
                        uint32 sizehint);
};
```

Figure 5.9: XML Segment Interface

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

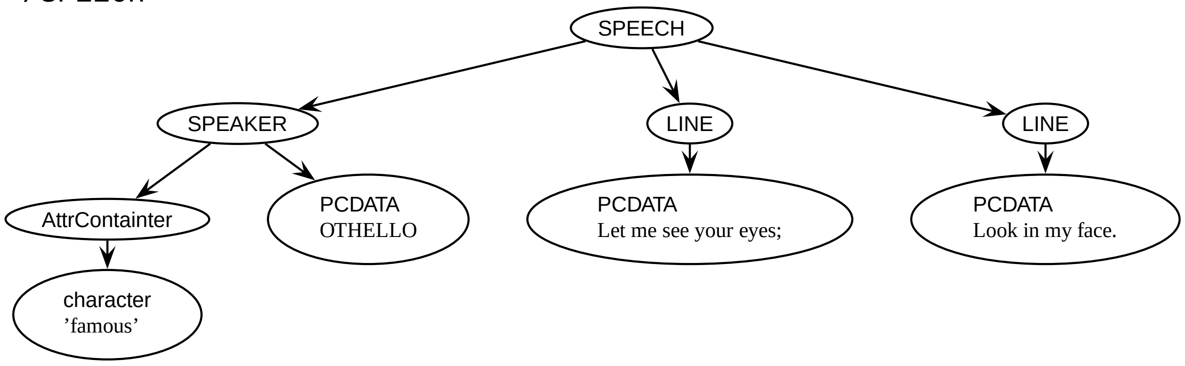

Figure 5.10: A fragment of XML with its associated logical tree

makes query evaluation simpler and more efficient because the possible integer values for a given tag or attribute name can be resolved once per query and stay the same for all documents in the segment. We call the integer labels `DeclarationID`s.

The interface of this so-called *Declaration Table* segment is shown in Figure 5.11. It is derived from the regular slotted page segment and contains a relation with schema $(qualifiedname, entrytype,$ `namespaceID`, `baseID`), where

qualified name is the tag name, possibly including a namespace prefix,

entry type indicates whether the entry describes an element type, attribute type, or a namespaces,

namespaceID contains the `DeclarationID` of the namespace this entry belongs to, and

baseID contains the `DeclarationID` of an entry with the same tag name, but without a namespace prefix.

In addition to simple insertions and lookups, the declaration table allows small queries that result in `DeclarationID` sets. Such queries can return all `DeclarationID`s for a given tag name, namespace, and/or entry type. Wildcards can be specified in form of null values, for example to express the query ”return all elements in a given namespace”.

Namespaces Namespaces can also be entered into the declaration table. There is one entry for each namespace. The `DeclarationID` of this entry identifies the namespace. Every element or attribute entry in the table contains this `DeclarationID` as `namespaceID`. For namespace entries, the namespace URL (see Section 2.2) is used as a name, and the `namespaceID` is unused.

```
class SEG_DeclarationTable : public SEG_SlottedPageSegment
{
public:
  class entryIterator;
  enum EntryType
  {
    DET_UNDEFINED,
    DET_ELEMENT,
    DET_ATTRIBUTE,
    DET_NAMESPACE
  };
  class DeclarationTableCB;
  class TableEntry
  {
    friend class SEG_DeclarationTable;
      cstring_t qname_mutable() const;
    public:
      DeclarationID declarationID() const;
      DeclarationID baseID() const;
      ccstring_t qname() const;
      EntryType entryType() const;
      DeclarationID namespaceID() const;
  };

  DeclarationTableCB *provideCB();
  void releaseCB(DeclarationTableCB *cb);

  DeclarationID addEntry(DeclarationTableCB *cb,
      DeclarationID namespaceid,
      ccstring_t qname,
      EntryType entrytype
      );

  void removeEntry(DeclarationTableCB *cb);

  TableEntry *resolve(DeclarationTableCB *cb, DeclarationID id);
  DeclarationID getEntry(DeclarationTableCB *cb,
      DeclarationID ns, ccstring_t qname, EntryType entrytype);
  DeclarationID lookupEntry(DeclarationTableCB *cb,
      DeclarationID ns, ccstring_t qname, EntryType entrytype);

  class DeclarationSet;
  DeclarationSet *queryBaseName(DeclarationTableCB *cb,
      DeclarationID ns, ccstring_t qname, EntryType entrytype);
  void releaseDeclarationSet(DeclarationTableCB *cb, DeclarationSet *cb);
};
```

Figure 5.11: Declaration table interface

Due to the prefix notation for namespaces, every tag name can appear with several different qualified names, even in the same document. For each such combination of qualified name and namespace, a separate entry in the table exists because the exact appearance of a document must be recreatable. For every combination of unqualified tag name and namespace, there is exactly one normalized entry, which is the version of the tag without prefix. All entries for that combination refer to this normalized entry with their `baseID` field.

Implementation Mapping from `DeclarationID` to entry is performed using a simple array. Tuples of $(qualifiedname, namespace, entrytype)$ are mapped to a declarationID using a hash table.

Queries are evaluated using scans of the tables. Since the buckets for the hash tables are based on the prefixless part of the names, queries for all `DeclarationID`s of a certain tag in a certain namespace can be answered by simply scanning one hash bucket.

The result of such declaration table queries are sets of `DeclarationID`s. The predominant operation on such a set is a containment query, asking whether a certain `DeclarationID` is present in the set. This is required for example when processing an XPath node test (Section 2.3), where nodes qualify only when they have a certain tag name. Since the tag name can appear with several different namespace prefixes, a node test translates into a containment test for a `DeclarationID` set.

The sets are represented as simple integer arrays. Typical small ID sets occupy only a single processor cache line. Thus, the common containment tests can be answered very fast.

Caching Translation between names and IDs is a hot spot, especially during bulkload or other update-heavy transactions, and during document export. The resulting dynamic memory management and synchronization can easily dominate the processing time.

As a remedy, to interact with the declaration table, a caller must first obtain a Declaration Table control block (`provideCB()`). All dynamic memory management is tied to this control block. Result pointers point to objects in a special memory area which is valid only until the control block is released. This reduces the burden on the dynamic memory manager.

The declaration table also builds control-block-local translation tables in main memory, which can be accessed without synchronization from a thread. Only if a local lookup fails, the global tables are accessed using synchronization[2].

This technique decreases the time spent for ID translation into insignificance.

[2]Removing single entries from the declratation table is not supportet yet. To do so, some kind of reference counting would have to be introduced to ascertain that there are no documents any more using the deleted entry. This is very expensive, and hence the declaration table is only completely erased when there are no documents at all.

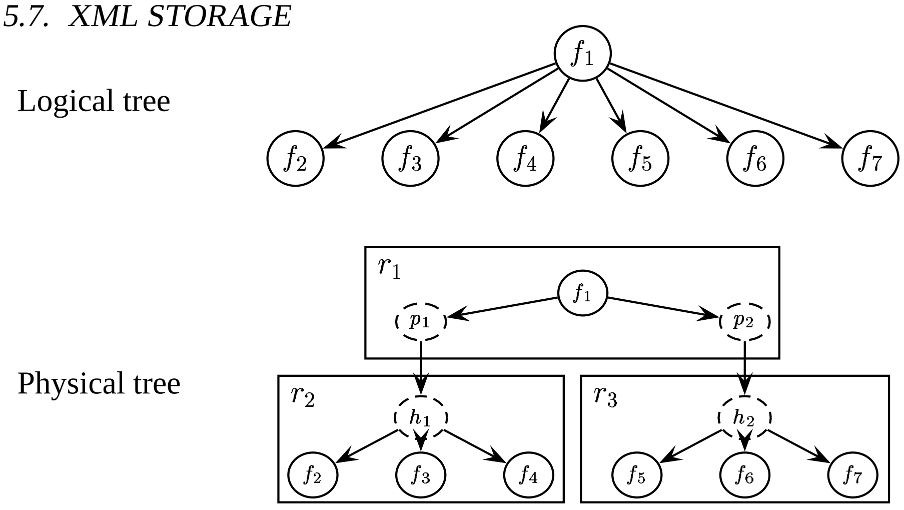

Figure 5.12: One possibility for distribution of logical nodes onto records

5.7.3 Physical Object Model

Typical XML trees may not fit on a single disk page. Hence, document trees must be partitioned. Typical BLOB (*binary large object*) managers achieve this by splitting large objects at arbitrary byte positions [4, 14, 52]. We feel that this approach wastes the available structural information. Thus, we *semantically split* large documents based on their tree structure.

We partition the tree into subtrees, in which *Proxy nodes* are used to refer to connected subtrees not stored in the same record. Their contents is the RID of the record containing the subtree they represent. Substituting all proxies by their respective subtrees reconstructs the original data tree.

A sample is shown in Figure 5.12. To store the given logical tree (which, say, does not fit on a page), the physical data tree is distributed over the three records r_1, r_2 and r_3. Two proxies (p_1 and p_2) are used in the top level record. Two helper aggregate nodes (h_1 and h_2) have been added to the physical tree. They group the children below p_1 and p_2 into a tree. Proxy and helper aggregate nodes are only needed to link together subtrees contained in different records.

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

The sample physical tree is only one possibility to store the given logical tree. More possibilities exist since any edge of the logical tree can be represented by a proxy. Sections 5.7.7 and 5.7.5 describe how to partition logical trees into subtrees fitting on a page. The following section will explain how the individual subtrees are materialized.

5.7.4 XML Page Interpreter

To store a logical tree in our XML segment, we partition it into subtrees (see Section 5.7.3). Each subtree is stored in a single record and, hence, must fit on a page. This section describes the interface and storage format used for subtrees on a page, as implemented in our *XML page interpreters*.

We first present a model of the subtree data, specify an interface to manipulate such trees, and then provide details about the storage format.

Physical Subtree Model

Each subtree represents part of a logical tree as defined in Section 5.7.1. In addition to leaves labelled with strings, physical subtrees also contain another kind of leaf node, which is labelled with references to other subtrees.

Every subtree also has two additional attributes. A *parent record* RID points to the parent subtree (if it exists), and a logical document ID field allows to determine which document this subtree belongs to.

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

Aggregate nodes represent inner nodes of the logical tree.

Literal nodes represent leaf nodes of the logical tree and contain text strings.

Proxy nodes are subtree leaf nodes which point to other records. They are used to link trees together that were partitioned into subtrees (see 5.7.3).

Interface

In the XML Page Interpreter Interface (Figure 5.13), subtree nodes are addressed using a combination of a `Slot` pointer, which denotes the subtree on the page, and a pointer to the current main memory buffer location of the node. A direct pointer is used instead of an offset into the slot to avoid extra computation steps when accessing the node. The slot pointer could be derived from the node pointer, but is still required as argument to avoid a lookup in the slot table of the page.

Mapping from slots to slot numbers, iterating over a page's records and other elementary page functions are inherited from the Slotted Page interpreter, from which the XML page interpreter is derived.

Updates The interface allows to create new subtrees (`createStandalone()`), create nodes in subtrees (`createEmbedded()`), and remove nodes or complete subtrees (`remove()`). New nodes are specified using their ContentType (Literal, Aggregate or Proxy, see above), their LogicalType (node label), a facade flag (see next section), and optionally some contents for leaves. When creating a new subtree, an arbitrary `documentID` to which the subtree belongs and the parent RID may also be specified. While in general, no updates are allowed except removing and reinserting nodes, subtree pointers form an exception to allow relocation of subtrees. Both the parent

```
class PGE_XMLPage : public PGE_RAL_SlottedPage
{
public:

  typedef uint8 ContentType;
  enum PhysicalNodeType ;
public:
  virtual void format();
  virtual ptr_t createStandalone(Slot *&s,
                                 const RID &parent,
                                 const DocumentID &id,
                                 bool facade,
                                 LogicalType lt,
                                 ContentType ct,
                                 uint16 contentsize,
                                 cptr_t content=0);

  virtual ptr_t createEmbedded(Slot *s,
                               ptr_t &node,
                               InsertionLocation il,
                               bool facade,
                               LogicalType lt,
                               ContentType ct,
                               uint16 contentsize,
                               cptr_t content=0);
  virtual bool remove(Slot *s, ptr_t &node);
  virtual void standaloneParent(Slot *,const RID &r);
  virtual void proxyTarget(Slot *s, ptr_t,const RID &r);

  const DocumentID &standaloneDocumentID(Slot *) const;
  const RID &standaloneParent(Slot *) const;

  ContentType contentType(Slot *, ptr_t) const;
  LogicalType logicalType(Slot *, ptr_t) const;
  bool hasContents(Slot *s, ptr_t) const;
  bool isAggregate(Slot *s, ptr_t) const;
  bool isEmbedded(Slot *s, ptr_t) const;
  bool isFacade(Slot *s, ptr_t) const;
  uint16 nodeSize(Slot *s, ptr_t) const;

  ptr_t contents(Slot *s, ptr_t) const;
  uint16 contentSize(Slot *s, ptr_t) const;
  const RID &proxyTarget(Slot *s, ptr_t) const;

  uint16 offset(Slot *,ptr_t) const;
  ptr_t  node(Slot *, uint16) const;
  ptr_t rootNode(Slot *) const;

  ptr_t parent(Slot *s,ptr_t) const;
  ptr_t next(Slot *s,ptr_t) const;
  ptr_t nextPreorder(Slot *s,ptr_t) const;
  ptr_t nextPreorder(Slot *s,ptr_t, int32 &navigation) const;
  ptr_t prev(Slot *s,ptr_t) const;
  ptr_t child(Slot *s,ptr_t) const;
  ptr_t findProxyFor(Slot *s,ptr_t,const RID &);

};
```

Figure 5.13: XML page interpreter interface

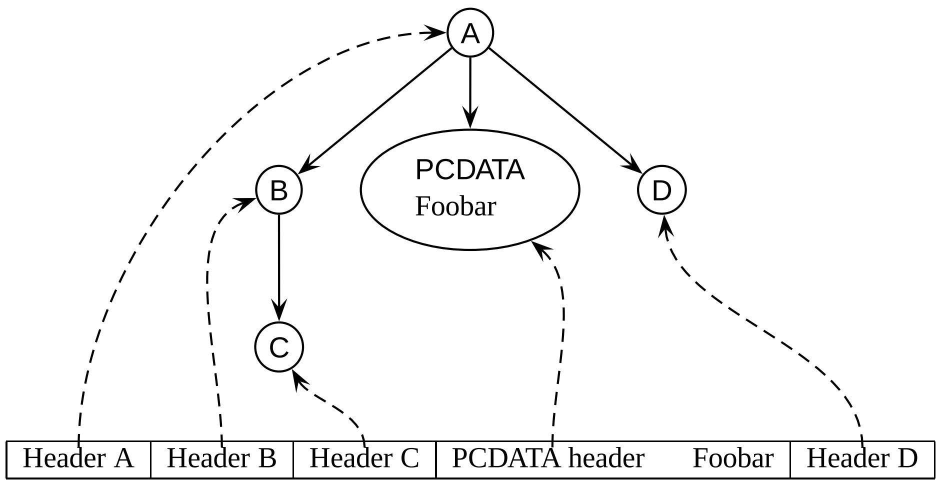

Figure 5.14: A small logical tree and its record representation

pointer (`standaloneParent()`) and proxy targets (`proxyTarget()`) may be modified.

Attributes Access methods for subtree-level attributes (`standaloneParent()`, `standaloneDocumentID()`) and node-level attributes exist.

Addressing The functions `offset()` and `node()` allow translation of current buffer addresses to intra-record offsets to create NIDs. The `rootNode()` function returns the address of a subtree's root node.

Navigation The navigation functions allow to access the neighbors of a node in the tree structure and to scan a subtree in preorder. The `findProxyFor()` function returns the proxy with a given pointer value.

Implementation

The storage format for subtrees needs to be as compact as possible. Since we augment the original XML data with additional structural information to allow navigation, there is the danger of increasing the space requirements to a point where the positive effect of the structural information is outweighed by the increased I/O overhead.

To keep our representation small and efficient, it is based on two main ideas.

First, we *embed* descendants into their ancestor nodes (Figure 5.14), and keep them sorted in document order. As a result, we do not need to store child pointers, because an internal node's first child is located at the beginning of the internal node's contents. We also do not need additional ordering information, as the nodes are already sorted in preorder, which is at the same time a very common access order when processing queries. Next sibling pointers are also not required, as adding a node's size to its start address gives the next node in preorder. If that node points to the same parent, its the next sibling, otherwise the node is the last child of its parent.

Second, for the individual subtrees, distances between nodes have an upper limit, the page size. This optimizes the representation of the remaining intra-subtree pointers, as the node size and parent pointers only consume 2 bytes each, if a typical page size smaller than 64K is used. Hence, a node header in Natix consists of a 3-Bit ContentType, a 2 byte node size, a 2 byte parent pointer, and a 2 byte logical type. The remaining bits of the byte used to hold the content type can be used to store the facade flag and other flags (for example, the *fresh flag* for Subsidiary Logging, see Section 7.6). Together with an unused extra padding byte this results in a header size of 8 byte for an embedded node.

Note that storing vanilla XML markup with only a 1-character tag name (`<X>`...`</X>`), for example, already needs 7 bytes! On average, XML documents inside Natix consume about as much space as XML documents stored as plain files in the file system.

5.7.5 Updating Documents

In this section, we present Natix's algorithm for the dynamic maintenance of physical trees, i.e. the implementation of the segment's second `create()` call from Figure 5.9. The principal problem addressed is that a record containing a subtree grows larger than a page. In this case, the subtree has to be partitioned into several subtrees, each fitting on a page. Scaffolding nodes (proxies and maybe aggregates) have to be introduced into the physical tree to link the new records together.

To better understand the 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 records as nodes. Each record contains a part of the logical data tree. In the example in Figure 5.15, 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 nodes.

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

Algorithm for Tree Growth

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

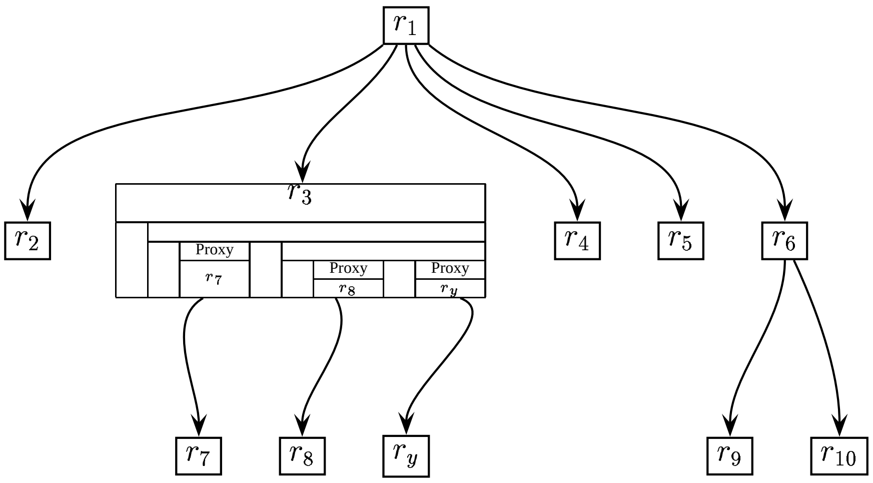

Figure 5.15: Multiway tree representation of records

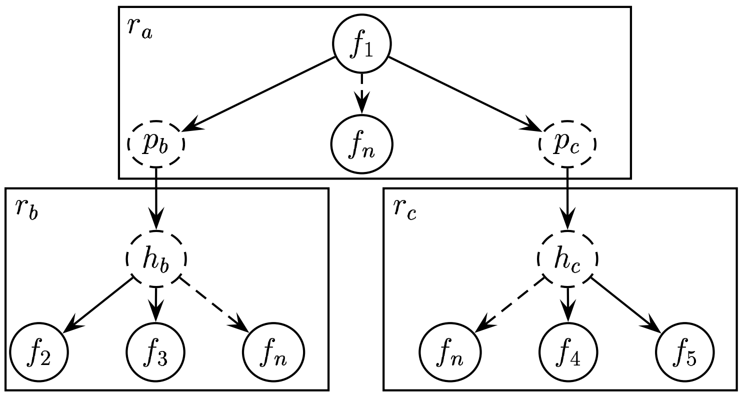

Figure 5.16: Possibilities to insert a new node f_n into the physical tree

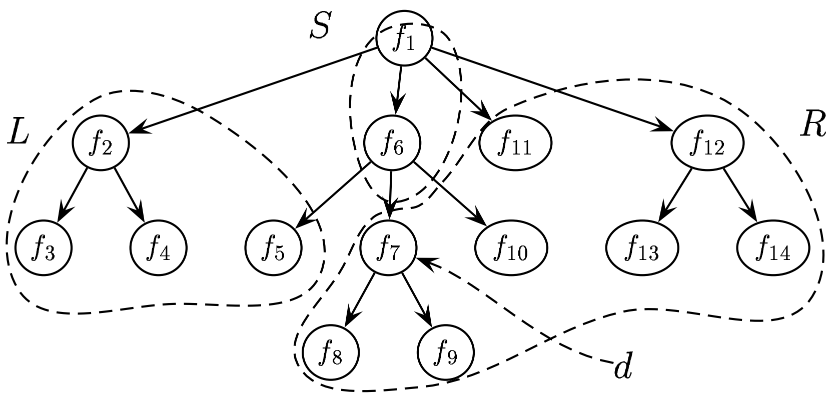

Figure 5.17: A record's subtree before a split occurs

1. Determining the Insertion Location To insert a new node f_n into the logical data tree as a child node of f_1, it must be decided where in the physical tree the insert takes place. In the presence of scaffolding nodes, there may exist several alternatives, as shown by the dashed lines in Figure 5.16: the new node f_n can be inserted into r_a, r_b, or r_c. In Natix, this choice is determined by the split matrix (see below).

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

(a) Determining the separator Suppose that in Figure 5.16 we add f_n to r_b, which cannot grow. Hence, r_b must be split into at least two records r'_b and r''_b, and instead of p_b in the parent record r_a, we need a *separator* with proxies pointing to the new records to indicate where which part of the old record was moved.

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

Figure 5.17 shows the subtree of one record just before a split. It is partitioned into a left partition L, a right partition R, and the separator S. This separator will be moved up to the parent record, where it indicates into which records the descendant nodes were moved as a result of the split operation.

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

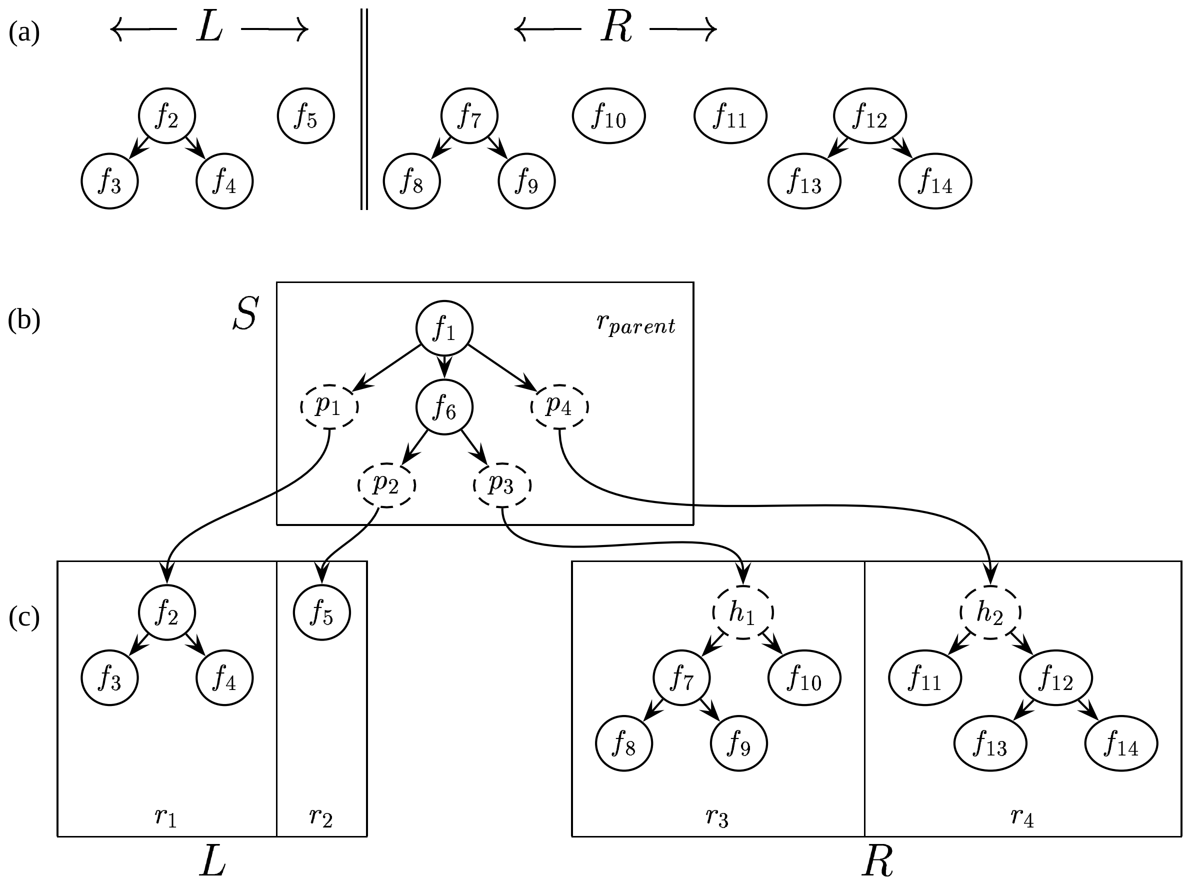

Figure 5.18: Record assembly for the subtree from Figure 5.17

Hence, the split algorithm must find a node d, such that the resulting L and R are of roughly equal size. Actually, the desired ratio between the sizes of L and R is a configuration parameter (the *split target*), which can, for example, be set to achieve very small R partitions to prevent degeneration of the tree if insertion is mainly on the right side (as when creating a tree in pre-order from left to right). Another configuration parameter available for fine-tuning is the *split tolerance*, which states how much the algorithm may deviate from this ratio. Essentially, the split tolerance specifies a minimum size for d's subtree. Subtrees smaller than this value are not split, but completely moved into one partition to prevent fragmentation.

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

In the example in Figure 5.17, the size of the subtree below f_7 was smaller than the split tolerance, otherwise the algorithm would have descended further and made $d = f_7$ part of the separator.

(b) Distributing the nodes onto records Consider the partitioning implied by node $d = f_7$ (Figure 5.17). The separator is removed from the old record's subtree, as in Figure 5.18(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 5.18(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 — by recursively calling the insertion procedure — where it replaces the proxy which referred to the old, unsplit record. If there is no parent record, as in Figure 5.18(b), the separator becomes the new root record of the tree. The edges connected to the nodes in the partition records are replaced by proxies p_i. Since children with the same parent are grouped in one scaffolding aggregate, for each level of the separator a maximum of three nodes is needed, one proxy for the left partition record, one proxy for the right partition record, and one separator node.

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

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

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

5.7.6 The Split Matrix

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

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

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

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

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

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

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

We also use the split matrix as the configuration parameter for determining the insertion location of a new node (see Section 5.7.5): When a new node x (label j) should be inserted as a child of node y (label i), then if $s_{ij} = \infty$, x is inserted into the same record y. If $s_{ij} = \mathbf{other}$, then the node is inserted on the same record as one of its designated siblings (wherever more free space exists). If $s_{ij} = 0$, x is stored as the root node of a new record and treated as described above.

Finally, the Split Matrix is also obeyed during bulkloads. New subtrees are not only created when the current subtree is full, but also when the Split Matrix dictates that the new node has to be stored in a separate subtree.

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 [48].

Another aspect we will not detail here is that concurrency control is based on physical records [77], and by forcing certain elements or attributes to be stored in separate records, they are effectively available as a granularity of locking.

5.7.7 Bulkloading Documents

The insertion of large amounts of data which is already available in a non-DBMS format is called a *bulkload* operation. In conventional DBMS, bulkloads are often used to initialize a database, for example when introducing an application to DBMS usage, or when converting data from a different DBMS or storage format.

In an XBMS, importing documents from external sources is a very frequent operation, as XML was designed as a data exchange format. Since a document consists of a large number of individual nodes, every document import is essentially a small bulkload operation. Hence, an efficient bulkload support is crucial for XBMSs.

Bulkload Design

We base our design of the bulkload component on three goals, all of which are performance-related.

1. The interface should closely match the typical output of XML parsers.

   Since an XML parser is the most common source of imported XML documents, we do not want to waste resources by requiring to change the data representation before or while accessing the bulkload component, in addition to potential representation changes for the actual transfer to the persistent storage format.

2. The mechanism should not require main memory proportional to the document size.

   Linear memory usage would prohibit import of documents larger than main memory. As a generalization, the total amount of concurrently importable documents would be limited by available physical memory. Although today's virtual memories are large, using virtual memory would result in thrashing, negating the benefits of concurrency.

3. The produced storage layout should be efficient for typical workloads on documents.

   We identify three subgoals.

   (a) A dominant access pattern for document trees is the preorder traversal of subtrees induced by inner nodes. It is used when exporting documents and document fragments to their textual representation. Query evaluation on XML documents typically also relies on preorder traversals, such as the evaluation of XPath `descendant` and `descendant-or-self` axes. The default bulkload strategy therefore is to create a layout which adequately supports preorder traversal.

(b) Given a set of children, we assume that the access frequency of sibling nodes decreases with their order. Typically, the leftmost children are accessed more often than the last children. For example, to reach any child by position in its sibling sequence, in Natix storage format, all left siblings of the target node need to be visited. Hence, the likelihood of being stored in the same record as the parent node should be higher for left siblings.

(c) Finally, the split matrix (Section 5.7.6) contains some information about typical workloads. This information should be exploited.

4. The produced storage layout should have minimal space requirements.

The goals imply some kind of clustering algorithm that partitions a tree into a minimum number of subtrees with limited size, which can then be used as Natix XML subtree records.

There are efficient clustering algorithms applicable to weighted tree structures [51, 54] which consider the problem of creating a clustering of a tree which minimizes the number of generated clusters. However, the generated clusters always have the following properties: (1) The weight of each cluster has an upper limit, which is a parameter of the algorithms. The weight of a cluster is the sum of the weight of its nodes. (2) All nodes of a cluster are connected.

Unfortunately, our storage format does not match well with these constraints on clusters, because in our case (1) the storage cost of a cut edge is not 0, as a cut edge causes overhead in the form of a proxy node and a new physical record header, and (2) it is possible to put several different siblings into a single cluster, creating nonconnected partitions of the tree.

There is another algorithm (Schkolnick [78]) which partitions hierarchical structures based on access patterns. However, it does not enforce a size limit for clusters, and does not consider nodes of different weight.

The clustering algorithm employed by Natix is described below.

Interface

Figure 5.19 shows the bulkload interface for XML segments.

The document tree to bulkload is ”described” to the segment in form of a sequence of ”visit events” resulting from a depth-first search of the tree. The bulkload user signals these events to the bulkload component by calling appropriate functions each time a node is visited.

This corresponds directly to parser interfaces such as SAX or libxml as described in Section 2.4. These generate parsing events which correspond to a depth-first search of the abstract syntax tree. Clients need to register callbacks with the parser which are invoked when the associated event occurs. Each SAX event can be directly translated into a single call of the bulkload interface[3].

[3] Attributes are an exception, as they are delivered as a list together with the parent element. This is a design error in the SAX interface.

```
class SEG_XMLSegment : public SEG_SlottedPageSegment
{
public:
[...]
  class BulkloadContext;
  BulkloadContext *beginBulkload(const DocumentID &doc, DeclarationID logt,
                                uint32 childcount, uint32 sizehint);
  void beginInternalNode(BulkloadContext *context, DeclarationID lt, uint32 children);
  void endInternalNode(BulkloadContext *context);
  void addLiteralNode(BulkloadContext *context, DeclarationID lt,
                      uint32 contentsize, ptr_t content);
  NID endBulkload(BulkloadContext *context);
[...]
};
```

Figure 5.19: XML segment interface

The first visit to the document root node initializes the bulkload (`beginBulkload()`), and the second visit (`endBulkload()`) terminates the bulkload and returns the NID of the stored root node. The `beginBulkload()` call allows to specify a size hint for the document. For small documents, this allows to fit the document into a matching gap on an already used page.

When visiting nonliteral nodes (`beginInternalNode()`) for the first time, the caller may specify how many children the internal node has, if known. After all descendants of the node have been added, `endInternalNode()` is called.

When visiting leaf nodes which are labeled with strings, `addLiteralNode()` is called.

Implementation

Our algorithm is based on the one by Kundu et al. [51], which creates a clustering of a tree with weighted nodes, where each cluster is connected and has at most weight k, and where the number of clusters is minimal. We first describe the algorithm, and then explain how it is modified for use in Natix.

Algorithm of Kundu et al. The algorithm pursues a bottom-up approach, successively assigning clusters to nodes. A node is processed only after its sons have been processed. *Processing* a node x guarantees that the weight of the subtree rooted at x is smaller than k. The *weight of a subtree* is the sum of all weights of those nodes in the subtree which have not been assigned to a cluster. While the subtree weight is larger than k, new clusters are created for sons of x, each containing the subtree including the son and all descendant nodes that are not yet assigned to a cluster. Partitions are created for the sons in order of subtree weight, with the heaviest subtrees first. Once the subtree rooted at x has a weight less than k, processing of x is finished. When this algorithm has reached the root node of a tree, the resulting clusters are smaller than k, and a minimum number of clusters has been generated (Refer to [51] for a proof).

Suitability as bulkload algorithm Document bulkload is easily translated into a problem instance for the algorithm above. Document tree nodes have a weight proportional to their space usage, clusters are stored as physical records, and the limit for the size of a physical record is the system page size. The algorihm generates physical records in a bottom-up manner, so that subtrees induced by some inner nodes are in as few physical records as possible. This prepares preorder traversals of document fragments, as required when exporting or traversing such subtrees when evaluating queries.

However, a bulkload algorithm for Natix needs to address some additional issues as explained above:

1. We do not want to keep the whole document tree in memory.
2. There is an overhead weight associated with a physical record, because the standalone header and the proxy node in the referring record occupy space.
3. Neighbouring siblings can be assigned to the same physical record, amortizing the overhead weight over several subtrees.
4. The leftmost siblings should have a higher probability of being clustered with their parent.

The first item can easily be addressed, since the algorithm's bottom-up approach does never change a node's assignment to a cluster. Hence, once a cluster has reached the record size limit, it can be stored in a physical record on disk and the constituent nodes need not be retained in main memory.

The second item has to be taken into account in the weight calculations of the algorithm (see below).

The third item introduces another degree of freedom when processing nodes. Instead of choosing the heaviest son first when creating new subtrees, it is now possible to create an ”artificial” heaviest son by grouping consecutive siblings together into physical records. This can be used to address item 4, to make clustering of leftmost sons with their parent more likely. We can store some of the rightmost sons together in a separate physical record, while keeping a heavier son further to the left in the same cluster as its parent.

The second and third items break the optimality proof of the algorithm when applied to Natix, demoting it to a heuristic with respect to minimum number of records generated. It is not clear how the bottom-up algorithm can be modified to retain optimality. If at all possible, a combinatorical approach seems likely that would have to select the least costly of all possible partitionings of records into clusters. We were not able to find an algorithm to do this efficiently. Since efficiency is of great importance for document import, and the heuristic algorithm explained below generates satisfying clusterings in all observed cases, we consider an suboptimal clustering acceptable.

Natix bulkload algorithm We now explain the variant of the above algorithm used in Natix. Instead of choosing the heaviest son to be assigned to a separate cluster from the parent, Natix combines some of the *rightmost*, unassigned, consecutive children of the

```
void SEG_XMLSegment::beginInternalNode(BulkloadContext *context, DeclarationID id)
{
  context->current()->appendNode(new BulkloadNode(id));
}
```

Figure 5.20: Code for `beginInternalNode()`

currently processed node and clusters them in physical records smaller than the size limit. This amortizes the record overhead over several nodes. It also increases the likelihood of the leftmost children to be clustered with the parent node.

The algorithm maintains a main-memory tree which consists of nodes that have not been assigned to a cluster yet. The main-memory tree nodes are stored using native C++ pointers for parent references, and sets of child pointers in each node. The main-memory tree also includes main-memory versions for proxies referencing subtrees which have already been assigned to clusters and moved to physical records. The worst-case size of this main-memory tree is proportional to the height of the document tree, i.e. the maximal path length from the root node to a leaf node in the document. This property is guaranteed by keeping only a fixed weight limit of nodes on each level (see below).

In the beginning, bulkload starts with an empty main-memory tree. Every call to the interface functions to construct the document either results in a new main-memory node, or transfers some of the main-memory nodes to secondary memory by assigning them to a cluster, or both.

To simplify the exposition, we only consider treatment of the `beginInternalNode()` and `endInternalNode()` functions. Calls to `addLiteralNode()` can be regarded as calls to `beginInternalNode()` immediately followed by `endInternalNode()`.

The `beginInternalNode()` code simply adds the new node to the main memory tree (Figure 5.20). The node is buffered in this main-memory tree since it only can be processed until its complete subtree has been described using the bulkload interface.

When `endInternalNode()` is called (Figure 5.21), the current node's subtree has been completely visited by the depth-first traversal, and it can be processed. The function `pruneCurrentCluster()` is called to guarantee that the node's subtree is smaller than the physical record size limit. Then, the parent of the current node becomes the new current node, and its weight is adjusted by the currently processed node's subtree weight. Finally, if the main memory tree below the current node has reached a certain constant limit, we start to create physical records to reduce the amount of memory occupied by the bulkload.

Figure 5.22 shows code for the pruning of the main-memory tree. If the subtree below the current node does not fit on a page when the standalone record header is taken into account, then the children of the node are clustered into physical records until the size of the main memory subtree falls below the limit. The `IGNOREMATRIX` identifier is explained below.

During pruning of the tree, physical records are created which contain subtrees of the main-memory tree. These main-memory subtrees are replaced with main-memory proxy

```
void SEG_XMLSegment::endInternalNode(BulkloadContext *context)
{
  BulkloadNode *processed=context->current();
  pruneCurrentCluster(context);
  context->current(processed->parent());
  context->current()->addWeight(processed->weight());
  if(context->current()->weight() > memoryLimit())
    pruneCurrentCluster(context);
}
```

Figure 5.21: Code for `endInternalNode()`

```
void SEG_XMLSegment::pruneCurrentCluster(BulkloadContext *context)
{
   BulkloadNode *current=context->current();

   if(current->weight() + clusterOverhead() > recordSizeLimit())
      clusterChildren(context, IGNOREMATRIX);

   while(current->weight() + clusterOverhead() > recordSizeLimit())
      clusterChildren(context, PARTITIONPROXIES);
}
```

Figure 5.22: Code for `pruneCurrentCluster()`

nodes. Therefore, even after creating clusters and removing the nodes from the main-memory tree, the remaining proxy nodes may still cause the subtree to be larger than the record size limit. Hence, the proxy nodes themselves are grouped into clusters and physical records are created for them, possibly in several layers, until the subtree fits into the size limit.

The `clusterChildren()` function (Figure 5.23) determines the cluster boundaries, moves clustered subtrees into physical records, and replaces the subtrees with proxies in the main-memory tree. Note that the grouping of child nodes into clusters proceeds from right to left, to address item 4 by first selecting the rightmost nodes to put into a separate record.

Instead of showing code, we will only briefly describe the lower-level functions required by `clusterChildren()`. The `findClusterBoundRight()` and `findClusterBoundLeft()` functions determine the interval of those children of the current node that are to be included in a new physical record. `findClusterBoundRight()` looks for nodes satisfying a predicate that depends on the `mode` parameter. The search starts at the second argument `lastsplit` and continues to the left siblings. If `mode == IGNOREMATRIX`, then the predicate is true for all non-proxy nodes. Otherwise, any node qualifies.

`findClusterBoundLeft()` moves further right starting from the rightmost node of the new partition. It includes nodes into the interval while they satisfy the same predicate as above, and while the closed interval of subtrees bounded by `firstsplit` and

```
void SEG_XMLSegment::clusterChildren(BulkloadContext *context, ClusterMode m)
{
  BulkloadNode *current=context->current();
  BulkloadNode *lastsplit=current->lastChild();

  lastsplit=findClusterBoundRight(context,lastsplit,mode);

  while(lastsplit!=0 &&
        current->weight() + clusterOverhead() > recordSizeLimit() )
  {
    BulkloadNode* firstsplit;
    firstsplit=findClusterBoundLeft(context,lastsplit,mode);
    RID target=createRecord(context,firstsplit,lastsplit,false);
    BulkloadNode* nextsplit=firstsplit->leftSibling;
    replaceWithProxy(context,current,firstsplit,lastsplit,target);
    lastsplit=nextsplit;
    lastsplit=findClusterBoundRight(context,lastsplit,mode);
  }
}
```

Figure 5.23: Code for `clusterChildren()`

`lastsplit` still fits into a physical record.

`createRecord()` is straightforward and creates new subtree records from the main-memory representations. If main-memory proxy nodes are included in the subtree, they are inserted into the physical record, and their target record's parent pointer is updated to refer to the new physical record.

The last frame on which records were inserted is kept in the `BulkloadContext` structure to avoid unnecessary free space searches (see `insertRecordOnFrame()` in Section 5.5.3). If a new page is required, free space search is limited to the FSI cache to keep bulkload free space searches efficient (Section 5.6.5).

`replaceWithProxy()` removes the main-memory representation of the subtrees that have been moved to a record and inserts a proxy instead.

Memory management The main-memory representation consists of a large amount of small objects. In the case of literals, these are even of variable size.

In spite of this, memory management is not expensive during bulkload. Memory is allocated for the nodes during a depth-first traversal. In depth-first preorder, all nodes of a subtree form a consecutive interval of nodes. This makes it possible for the bulkload component to use a special memory management technique. The special memory manager requests memory in blocks of constant size from the operating system, adding nodes to blocks in depth-first preorder as they are delivered to the bulkload component. The order in which the blocks are used is maintained in a list. When a subtree's main memory representation is no longer used, the interval of blocks which only contains nodes of this subtree can be deallocated in a per-block fashion, without regarding the individual nodes on the blocks.

Split matrix The split matrix (Section 5.7.6) can easily be integrated into the algorithm.

First, we need an additional mode for the `findClusterBoundRight()` and `findClusterBoundLeft()` functions, called `OBEYMATRIX`. Here, only nodes qualify which do not have to be clustered to the parent node according to the split matrix, i.e. for which $s_{ij} \neq \infty$. In `pruneCurrentCluster()`, an additional call to `clusterChildren` with this new mode is added before the first `clusterChildren` call (Figure 5.22). For nodes which have to be clustered with the parent, this causes the bulkload algorithm to split them into a separate physical record only if all other nodes have been split already, and the subtree is still too large.

Second, we check at the end of the `endInternalNode()` function whether the current node has to be stored in a separate physical record from its parent, i.e. for which $s_{ij} = 0$. In this case, we immediately create a new physical record for the node using `createRecord()` and only insert a proxy to it into the main memory tree.

5.8 Evaluation

A systematic and comprehensive performance review of Natix, including comparisons with other systems, while desirable, is beyond the scope of this work. However, in this section we present experimental results to allow a first assessment of Natix overall performance.

We compare the performance of Natix with different settings of the Split Matrix, and with some existing performance results from the literature. Since only few numbers can be found about other systems so far, and few is known about the systems and approaches that produced these numbers, they have to be interpreted with care.

In this section, we elaborate only on numbers measured for a configuration without logging and recovery, please refer to Section 7.9 for experiments with enabled recovery subsystem.

5.8.1 Environment

Hardware

The following measurements were performed on two different machines. The first was equipped with an 1.5 GHz AMD Athlon processor and 512MB of RAM (Machine 1), while the other had a Pentium III running at 600 MHz, also with 512MB RAM (Machine 2). The I/O subsystems employes Ultra Wide SCSI in both cases.

Software

As operating system, SuSE Linux with kernel version 2.4.18 was used. The Natix executable was created using gcc 3.2 with full optimization.

Natix Configuration

A single Natix partition was used, with 256000 pages of 8K each. It was accessed using *direct I/O*, which means that the all operating system caching, readahead and disk scheduling were disabled, and DMA to the Buffer Manager's memory was enabled. The Buffer Manager was configured to use 32000 pages of main memory, resulting in a buffer size of 256 MB.

Two different settings of the Split Matrix were examined. The first setting format uses the value **other** for all entries of the split matrix, the second 0 (see Section 5.7.6). The former clusters documents whereas the latter yields a storage format that stores one node per record. This is somewhat similar to mappings of XML documents to relational and object-oriented systems.

Document Collections

Two document collections were used in the performance measurements, and two additional kinds of synthetic documents were used for special bulkload experiments.

The first collection is an XML rendering of the plays of William Shakespeare [8]. It comprises 37 documents of total size around 8 MB, each about 200K in size. A scaling factor of 6 was used, to enhance the measurability of queries with very short running time. Hence, in total, 48MB of XML in 222 documents were used.

The second collection contains annotated peptide sequences of close to 5000 organisms, with one document for each organism. Depending on how much is known about the organisms and its complexity, the documents vary in size between 700 bytes and 15MB, with a total size of 135MB. The average document size is 30K, the median document size 3K.

For the bulkload scalability measurements, we used synthetic documents, consisting of a single tag name (`test`) which was nested 5 layers deep with a constant fanout for each document. Different fanouts were used to create documents of varying size. The elements of the lowest layer each had one text node as son, each containing a 58 byte string.

The comparison with other systems used a synthetic document of size 100MB, which is detailed in Schmidt et al. [80].

5.8.2 Bulkload

Document Size Impact

In Table 5.1, the elapsed import time for bulkload of synthetic douments with various sizes on machine 2 is shown. The total elapsed time was measured, including a full buffer flush after importing the document. It does not include the time spent to initialize the database structures.

The fanout for the document described in Section 5.8.1 was set to values between 5 and 16, resulting in documents between 200K and 75MB.

Figure 5.24 gives a graphical representation of the results.

Fanout	Document Size	Elapsed time
5	227K	0.329s
6	560K	0.772s
7	1203K	1.622s
8	2336K	3.069s
9	4195K	5.535s
10	7085K	9.729s
11	11386K	15.289s
12	17562K	23.136s
13	26167K	33.826s
14	37856K	53.416s
15	53393K	72.269s
16	73659K	105.899s

Table 5.1: Import scalability

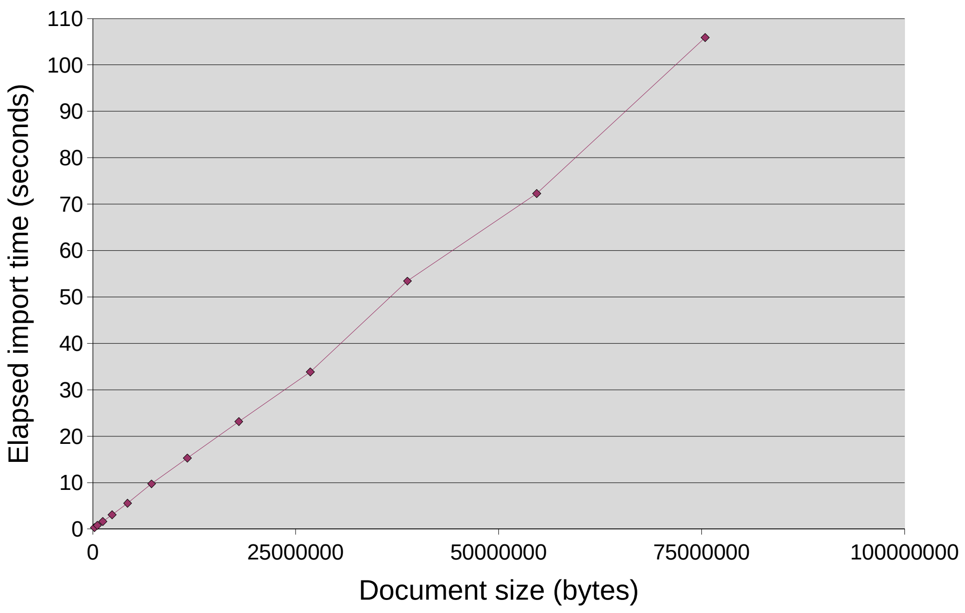

Figure 5.24: Import scalability graph

Collection		$s_{ij} = \mathbf{other}$	$s_{ij} = 0$
Bioml	Elapsed Time	150.98	253.41
	CPU Time	136.33	175.97
	CPU%	90%	69%
	Segment Size	26639	77932
Shakespeare	Elapsed Time	29.12	50.26
	CPU Time	20.31	36.14
	CPU%	70%	72%
	Segment Size	9028	26571

Table 5.2: Import performance (Times in seconds, sizes in pages)

We observe a nearly linear time behaviour of the bulkload algorithm, with an import speed of about 700K/s. The bulkload algorithm appears to be scalable for a large range of document sizes.

Storage Format Impact

Table 5.2 shows bulkload times and sizes for the two nonsynthetic document collections from Section 5.8.1, taken on machine 1. Again, the times include a system shutdown after import, to observe the time spent writing the dirty buffers to disk. The times do not include system setup.

We make four main observations. First, the clustered format takes less total elapsed time to load in all cases. Second, the clustered format consumes much less disk space. The savings factor approaches three. Third, compared to the documents' textual size, the documents in the clustered format take about 1.5 times as much space (8K pages were used). Fourth, the difference in elapsed time is not explained by the increased I/O load for the single node format alone, as the CPU time needed to bulkload is also significantly higher. The increased CPU time is caused by the larger amount of record insertions, since in each case, the buffer manager must be accessed, including synchronization and address translation overhead.

The I/O performance in the bulkloads is comparably low, between 8 and 16 MB per second. This is due to the fact that Natix currently does not have a writing mechanism which reorders writes to sequential order, and the measurements were performed using direct I/O, bypassing the system I/O scheduler. Hence, although they could have been sequential, most of the buffer writes resulted in random I/O, impacting the I/O rate.

Comparison with other systems

Published performance results for XBMS systems are rare and far between. The only comparable numbers we could find were in Schmidt et al. [80]. They compare bulkload performance for a 100MB synthetic XML document on various anonymous mass-storage systems. We repeat some of their results in Table 5.3.

System	Bulkload time (Seconds)
System A (from [80])	414
System B (from [80])	781
System C (from [80])	548
Natix	215

Table 5.3: XML Bulkload Times for various systems

We limit our comparison to the disk-based systems, omitting their numbers for purely main-memory based systems. The remaining systems are relational DBMS, and called ”System A”, ”System B” and ”System C” in the paper. No details about the employed mappings from documents to relations are given, except that systems A and B do not require a DTD, while system C requires to manually generate a relational schema from a DTD.

Table 5.3 also includes a measurement of Natix's bulkload performance for the same document. We used machine 1, which is very similar to the one described in Schmidt et al. [80], except that it has less main memory (512MB compared to their 1 GB), and a slightly faster processor (600Mhz compared to their 550Mhz). In this case, the Natix configuration included full optimized transaction support (see Section 7.9 for more Natix experiments with enabled recovery), and the time measured does include commit processing and log flushing, but not the flushing of the buffer manager.

Although Natix outperforms the relational systems by factors between 1.9 and 3.6, few is known about the exact configurations and techniques used to store XML in the relational systems. Hence, it is unclear to what extent the numbers are comparable.

5.8.3 Queries

Queries

We evaluate three queries on both the Bioml and Shakespeare document collections. We use the XPath language to specify the queries and explain them below.

Query 1 In XPath notation, the query evaluates the expression

```
/child::X/child::Y/child::Z/child::U
```

on all documents in the collection. As element names `X,Y,Z,U` we used specific names for each collection. For the Shakespeare collection, `X`=PLAY, `Y`=ACT, `Z`=SCENE, and `U`=TITLE. Hence, the query returns the titles of all scenes in each play. In the Bioml collection, `X`=bioml, `Y`=organism, `Z`=subunit, `U`=label. Here, the query selects the internal names of all peptide sequence subunits for all organisms.

Query 2 This query evaluates the expression

```
strval(/child::X/child::Y/child::Z[pos()=last()])
```

			Storage Format			
			$s_{ij} = \mathbf{other}$		$s_{ij} = 0$	
Collection	Query		Cold Cache	Hot Cache	Cold Cache	Hot Cache
Shakespeare	Query 1	Elapsed Time	12.65	0.46	21.19	3.25
		CPU Time	0.77	0.46	4.15	3.24
		CPU%	6%	100%	20%	100%
	Query 2	Elapsed Time	3.28	0.02	6.18	0.05
		CPU Time	0.04	0.02	0.13	0.05
		CPU%	1%	100%	2%	98%
	Query 3	Elapsed Time	14.73	3.74	59.36	42.25
		CPU Time	4.05	3.73	43.43	42.22
		CPU%	27%	100%	73%	100%
Bioml	Query 1	Elapsed Time	9.99	0.23	16.66	1.24
		CPU Time	0.41	0.21	1.69	1.24
		CPU%	4%	94%	10%	100%
	Query 2	Elapsed Time	8.91	0.97	15.63	2.30
		CPU Time	1.16	0.96	2.81	2.28
		CPU%	13%	99%	17%	99%
	Query 3	Elapsed Time	27.78	4.97	84.47	84.64
		CPU Time	5.94	4.94	53.91	53.42
		CPU%	21%	99%	64%	63%

Table 5.4: Query performance (Times in seconds)

on all documents. The same tag names as above are used. This query returns the text contents of the last scene, or the last subunit, for each play/organism.

Query 3 `count(/descendant-or-self::V)`

This query counts elements that occur very frequently in the document collection. For the Shakespeare collection, it counts all (V=LINE) lines of all speeches given in the play. In the Bioml collection, all `db_entry` elements are counted, which represent literature annotations for subsequences of the organism.

Execution Plans

This subsection briefly explains how the queries specified above were executed. We directly wrote execution plans in the NVM internal language, without using a query compiler.

The execution plans below navigate between nodes by using primitives which operate directly on the buffered representation of the tree in secondary storage. No representation change to a main memory representation is performed.

Query 1 The query was executed using a leaf operator scanning over a list of root nodes of the document collection (using the Document Directory, see Section 6.4). Then, a chain of *UnnestMap* operators enumerated the children of each node which matched the given tag name.

Without further detailing the involved operators, the execution plan has the effect of five nested loops, where the first loop enumerates the document roots, the second iterates over all children of nodes from the first loop which have an X tag name, and so on.

Query 2 The execution plan is similar to the previous query. However, instead of enumerating all Z nodes, and retrieving their children, only the last Z children node is selected. For this final node, a further *UnnestMap* loop then performs a depth-first traversal of its subtree and concatenates the values of all text nodes found.

Query 3 Here, only a single *UnnestMap* operator is performed after the initial enumeration of all document root nodes. It performs a depth-first traversal of the whole document tree, returning all matching nodes, which are then counted by an additional operator.

Results

The evaluation times for the query were measured both with an empty cache, and with the cache filled by a previous execution of the same query.

We emphasize some features of the collected data. First, if the clustered Natix data format is used, query evaluation is always faster compared to the format where every node is stored in a separate record. The clustered format outperforms the single node format by an order of magnitude in some cases, and it is still 50% faster in the worst case. Likewise, no matter which query or document collection, and whether the cache is hot or cold, the clustered format requires less CPU time than the single node format. For CPU usage alone, the advantage of the clustered format can be higher than an order of magnitude. The I/O portion of the query runtime is lower for the clustered format, even for Query 2, where only a few nodes per document are traversed.

Note that the Bioml collection stored as single nodes did not fit in the cache due to the high space requirements of separate records for each node. Hence, its performance is the same for hot and cold caches.

Chapter 6

Schema

Es gibt keine Ordnung der Dinge a priori.

–Ludwig Wittgenstein

Schemas are used to specify the logical and physical organization of an XML base.

Constraints for the logical structure of individual XML documents are specified using *Document Type Defi nitions* (*DTDs*), or XSchema documents. Both of these languages are dicussed in Section 6.1.

Natix allows to group documents into application-defined logical hierarchies. Just as relational databases organize data into databases and relations to store tuples, Natix provides *repositories* and *document collections* to store documents. Section 6.2 explains Natix's logical schema model, which describes how such collections and repositories are specified.

Section 6.3 presents the physical schema model. The physical schema controls the mapping of document collections on physical media, including indexes and the split matrix.

The chapter concludes with Section 6.4, where we show how the schema model elements are materialized using the primitives provided by the storage engine (Chapter 5).

6.1 Document Schemas

By *document schema*, we mean a set of constraints on the contents of individual documents. If a document satisfies the constraints put forth by a document schema, we say it is *valid* with respect to the schema. Hence, a document schema defines a class of documents, namely the class of documents that is valid with respect to the schema.

Meaningful constrains on documents can be classified by the degree of freedom they constrain:

tag names It is often desirable to limit the allowed tag names in a class of documents.

attributes It is possible to limit the kinds of attributes elements with a certain tag name may have, and to specify whether the attributes are mandatory or optional, and if they have a default value when not explicitly given.

	DTD	XML Schema
tag names	$\surd$	$\surd$
attributes	$\surd$	$\surd$
content models	$\surd$	$\surd$
reuse	–(1)	$\surd$
inheritance	–	$\surd$
referential integrity	$\surd$	$\surd$
no. of key domains	1	n
data types		
attributes	$\surd$(2)	$\surd$
text nodes	–	$\surd$
ranges	–	$\surd$
inheritance	–	$\surd$

Table 6.1: Document schema features

element content models These specify the allowed parent-child and sibling associations which certain element types may form. For example, the content model may require elements of type person to have a `name` and one ore more `address` elements as children.

referential integrity Referential integrity constraints allow to specify that certain attribute or element values are *keys* or *key references*. In valid documents, key references must contain values that are existing keys in the document.

data types In certain attributes or `PCDATA` nodes, only a specific data type or range of values, may be meaningful. A `price` tag should only contain a positive fixed-point number, for example, and a `weekday` attribute may only contain names of week-days.

There are two standardized schema specification languages, *Document Type Defi nitions* (DTDs) [10] and *XML Schemas* [90, 5].

Table 6.1 shows which classes of constraints these languages support. The elementary constraints can be expressed using DTDs, which are based on a non-XML syntax for backwards compatibility to SGML. Parts of content models often need to be reused (1), for example both <`short_description`> and <`long_description`> tags use the same content model to describe text layout. To factor commonalities between tags, XML Schema allows to define reusable content model fragments, and allows tags to inherit and extend content models of other tags. In DTDs, this can only be approximated using *parameter entities*, a textual replacement mechanism. The closest DTDs come to the concept of a data type is the NMTOKEN attribute type (2), which specifies accepted value sets for attributes. XML schema allows to specify data types for all literals in a document, and data types can be also be numeric, inherit from each other, or be value ranges of a supertype.

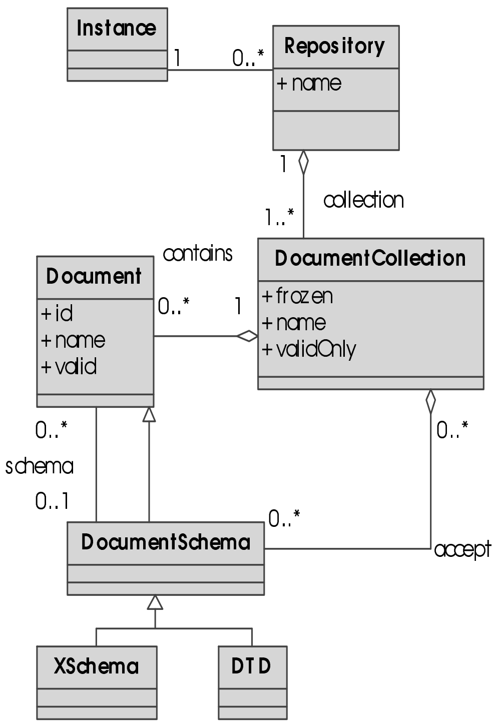

Figure 6.1: Logical schema model

We do not go into further details about document schemas, as currently the database engine is not involved in document validation, but uses external validators for this purpose. Natix only validates documents when they are added to a repository and does not check during updates whether the resulting document is still valid. Revalidation of documents must be triggered manually. Online validation during incremental updates is left as future work. A basic idea in this context is the materialization of the validation automaton's state information in our storage engine, which would limit the necessary revalidation to small granularities of a document.

6.2 Logical Schema Model

Figure 6.1 shows a UML model of the concepts used to structure the set of documents in a Natix instance logically.

Each instance manages a set of repositories, where a *repository* is a set of document collections. Usually, applications will use repositories to group together document collections which are semantically related.

A *document collection* contains documents with similar contents. The number of different schemas that can have conforming documents in the same collection is typically rather small. Document collections are often used as input sets for queries.

Below, these concepts are elaborated on.

6.2.1 Repository

A *repository* typically contains documents relevant for a single application domain.

For example, in a Web Shop System, the product catalogs make up one repository, the business reports are stored in a different repository, and the documents and templates used as basis for the web appearance are stored in a third.

Typically, an application only accesses one document repository. In our example, catalog editors only access the catalog repository, as do tools used to generate special catalogs for special opportunities, like seasonal sales. Web authoring tools work exclusively with the web documents, while a management information system is primarily concerned with business reports.

A few applications use more than one repository. In our example, this would be the software generating the web pages in their final form that is sent to the customers. It needs to access both the template documents and product catalogs and merges them to create viewable pages.

Repositories are a granularity for authorization, concurrency control, and backup. For example, only managers may access business reports, and web designers may only read, but not update product catalogs.

All repositories in an instance have a unique `name`.

6.2.2 Document Collection

Document collections are unordered sets of documents. Typically, the documents in one collection have a similiar structure and markup, although they do not need to conform to the same document schema. Applications use document collections to group together documents that are frequently processed as a unit, for example when evaluating queries. As explained in Section 6.3, document collections also are a granularity of physical placement control.

Document collections have a `name` which is unique among the document collections of a repository.

In Natix, applications can announce to the XBMS which document schemas the documents added to the collection will conform to (`accepts` association in Figure 6.1). If the *validOnly* flag is set, then documents are only added to the collections if they conform to one of the specified schemas. Otherwise, all well-formed documents can be added to the collection. It is required to store invalid documents, since there are applications of XBMS, like web crawlers [11], which need to deal with many different sources of documents, some of which may not use tools that enforce validity. Although invalid, such documents are interesting for the users of the application.

Another property of document collections is whether they are *frozen* or not. A *frozen* document collection may not have new document schemas associated with it. The *frozen* property implies *validOnly*. This allows for the XBMS to select an optimized storage format for a collection. For example, only for frozen document collections, techniques can be applied which transform a fixed document schema into a relational schema [20, 29, 82].

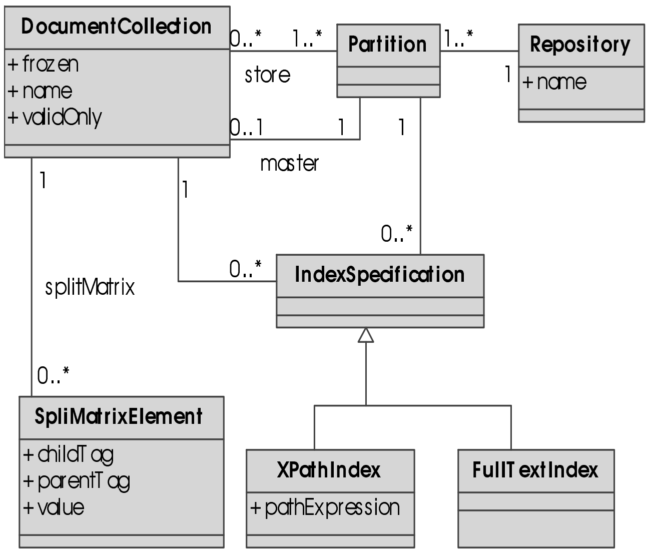

Figure 6.2: Physical schema model

The transformation has as consequence that no documents that do not conform to such a schema can be stored, and that schema evolution is expensive.

Natix does not use any of the cited methods, but has its own flexible XML storage layout as outlined in Section 5.7. However, we feel that the conceptual schema model must be flexible enough to incorporate schema-specific storage formats if necessary.

6.2.3 Documents

Individual documents have a `name` which is unique within their collection, and a stable ID which is unique in the whole instance and does not change.

A document may have an associated document schema, to which it may conform or not (`valid` flag).

In addition, a document is associated with a set of *properties*. Each property has a name and a value. The name of a property is unique in the document's set of properties. The value can be a text string, or a well-formed XML document. Some of these properties are system-maintained, such as creation date, while others are user-defined and can be modified arbitrarily.

An example for a user-defined property is the *MIME type*, which is a classification of the document contents. Natix's WebDAV module (Section 4.3.3) annotates stored documents with their MIME type, to make it available to the clients which can select suitable tools to display or process the documents.

6.3 Physical Schema Model

Repositories and document collections can be associated with partitions to control the physical placement of the data. Further, by specifying a split matrix, clustering of document parts can be controlled. Indexes can be specified to increase access speed.

A model for physical schema specification in Natix is presented in Figure 6.2. The following subsections provide details on the components of the diagram.

6.3.1 Repository

Each repository has a *master partition*, which is the one on which it was originally created. On this master partition, metainformation is stored including the available document collections and their distribution on partitions.

Additional partitions may be added and removed from the repository (see below), but the master partition may never be removed from the repository.

While a partition may contain documents of several document collections, each partition holds only data from exactly one repository. Thus, data from different repositories is thus separated physically, to enhance concurrency and security.

6.3.2 Document Collection

For each document collection, it is possible to specify on which partitions its documents may be stored. This concept is akin to the *tablespaces* used in many relational DBMS [16]. If all partitions assigned to a document collection are full, but further space is needed for update operations, these operations fail until the administrator mounts and assigns additional partitions.

6.3.3 Split Matrix

For every document collection, a split matrix may be specified. Its effect is explained in detail in Section 5.7.6.

The split matrix is stored as a set of objects which describe those coordinates in the matrix which deviate from a value of **other**. The coordinates are given in form of element or attribute names.

Hence, if no `SplitMatrixElement`s are given for a document collection, the update algorithms have full control over the placement of nodes. By introducing `SplitMatrixElement`s, some nodes can be clustered or separated from others.

6.3.4 Indexes

It is possible to create indexes for document collections, which accelerate access to document nodes satisfying certain predicates.

Natix allows to specify two kinds of indexes, *XPathIndex* and *FullTextIndex*. A current limitation of Natix is that indexes cannot span partitions. Instead, it must be specified for every index on which partition it is located. In addition, indexes are not maintained automatically, but reindexing has to be triggered by applications or administrators.

XPathIndex

An XPathIndex is similar to an attribute index in a relational DBMS. Instead of a table and a set of columns of that table, an XPathIndex is specified on a document collection and an XPath expression.

For a given string k, an XPathIndex based on the XPath expression X allows for quick retrieval of all the nodes x for which the XPath expression $k = string(x/X)$ is true. By x/X we mean the result of evaluating X with context node x. The result is converted to a string, to have a single data type which can be used as the key for the index. Otherwise, since XPath expressions can have arbitrary result types, including sets of nodes, it would be difficult to use and implement the index structure.

Full Text Index

Given a word k, a full text index can generate a list of text nodes and/or attributes which contain the word k.

6.4 Detailed Design and Implementation

The storage subsystem primitives available in Natix (Section 5) provide adequate means to materialize instances of the schema models, as explained in the following.

6.4.1 Overview

In Figure 6.3, the data structures used to implement the logical and physical schemas are shown.

To represent document collections, several segments are required. Their purpose and their distribution on partitions is explained in Section 6.4.2. To quickly access documents by their ID, a *document directory* data structure is used to map `DocumentID`s to their physical location (Section 6.4.3).

A repository's schema is materialized using a *catalog* document collection (Section 6.4.4), which also stores the association between collections and the constituent segments, and information about which schemas are accepted by a collection (Section 6.4.5).

6.4.2 Document Collection

Document collections are stored using a number of segments.

document tree The document tree itself is stored in an XML Segment. There is one XML Segment for each collection on every partition assigned to the collection.

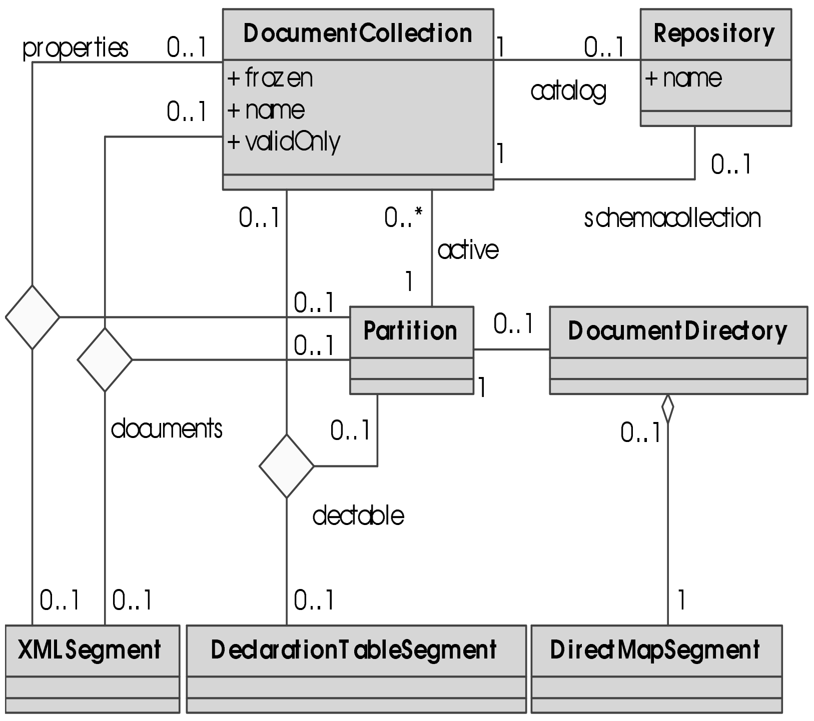

Figure 6.3: Implementation model

declaration table For every XML Segment storing a collection's documents, there also exists a declaration table segment on the same partition (`documenttable`), containing the mapping between the tag names and the integers used to label the tree nodes (Section 5.7.2). Separate declaration tables for each partition cause document storage on each partition to be self-sufficient, making it possible to access and export documents without having to access other partitions. While this results in a duplication of declaration table entries, the different declaration tables do not need to be synchronized, as there is no need for `DeclarationID`s for the same tag name to be identical across partitions.

The design requires resolution of the used tag names to integers only once per partition, amortizing the resolution overhead over many documents.

properties The properties of a document are stored in a separate XML Segment on each partition assigned to the collection. It contains exactly one property document tree for each document, in which every property is stored as a subtree containing the properties value. Thus, the efficient storage format for XML can not only be used for the document contents, but for for XML-valued properties as well.

The document's name is also considered a property and is stored in the property document.

The property document also contains a backpointer to the document ID. Hence, all documents of a collection on a specific partition can be efficiently enumerated by scanning the property documents, which are much smaller than the documents themselves.

The last partition on which documents for this collection have been stored is called the `active` partition. When adding a new document to a collection, Natix attempts to store it on the active partition first. If there is not enough space, then the other partitions assigned to the collection are tried.

When adding a document, it is first inserted to the document directory (see below) to obtain a `DocumentID`. Then, a property document is created, and finally, the document is bulkloaded as explained in Section 5.7.7. If it is not already, the partition is then made the `active` partition for the collection.

6.4.3 Document Directory

There is a *document directory* on every partition which contains XML document trees. It contains information about all documents in the partition, including their physical location, schema, validity and other properties. The `DocumentID` is used as a key to access a document's entry in the directory. All records in XML Segments contain a backpointer to the `DocumentID` of the encompassing document (Section 5.7.4).

The design decision to have one document directory per partition, and not per repository or per document collection, is based on the desire to localize the data structures used to materialize documents. Repositories and document collections can be distributed over partitions. If the data structure mapping `DocumentID`s to physical locations was distributed arbitrarily, then even simple read access to a single document could result in cross-partition I/O. If each document collection had its own document directory, or directories, then the collection would have to be known and loaded to access a document. As consequences, an additional level of indirection would be introduced, causing extra disk accesses to reach a document given its ID, and the collection identifier would have to be included to address a document in indices or other document references, wasting space.

The directory is implemented as a DirectMap segment (see Section 5.5.2). The slot and unique numbers in the DirectMap segment, together with the partition number, are used as `DocumentID`. Hence, the `DocumentID` is sufficient to locate a document within a Natix instance. Even if a document is deleted, dangling external references can be detected using the differing unique number.

Each DirectoryEntry contains the following fields

`entryType` Possible entry types are

- **document** describes a document in this partition.
- **schema** a schema document in this partition.
- **redirect** a document that was once located on this partition, but has been moved to another partition.

`segmentID` Contains the `SegmentID` (see Section 5.6.3) of the XML Segment in which the document is stored.

`documentNID` Contains the NID of the document's root node.

`propertyNID` Contains the NID of the property document tree's root node (explained in Section 6.4.2).

`schemaID` Contains the `DocumentID` of the document's schema. May be 0.

In case of a redirect entry, this field contains the new documentID.

`validity` Is `true` when the document is valid with respect to the schema.

`declarationTableID` contains the `SegmentID` of the declaration table used to encode the document (Section 5.7.2).

The **redirect** entry type is used for a TID-like concept (see Section 5.5.3), when a document grows but there is no longer enough free space on its partition. Except for this case, a document directory entry only needs to be modified after creation only when it becomes invalid due to an update operation, or when it is deleted.

6.4.4 Repository Catalog

The *repository catalog* is a document collection describing a repository's schema.

It contains one document for each document collection in the repository, storing its attributes (`frozen, name, validOnly`) and its associated segments' `SegmentID`s. Thus, the repository catalog materializes the `collection` association from the logical schema (Figure 6.1).

The `SegmentID`s stored in the document represent the `properties`, `documents`, and `dectable` associations from Figure 6.3. They also implicitly materialize the `store` association from Figure 6.2, the set of partitions assigned to the collection. If a new partition is assigned to a collection, the document, property and declaration table segments are created even if no documents are stored on the partition yet.

In addition, the repository catalog documents contain information about the `split-Matrix` (Figure 6.2) configuration of the collections, and the `active` partition (Figure 6.1). The `accept` association from the logical schema is also stored in the catalog (see Section 6.4.5).

The *root document* which describes the repository catalog itself, must be stored on the master partition. It is always the first entry in the document directory for the master partition, which makes it easy to locate. All other partitions assigned to collections of the same repository contain as first entry in their document directories a redirect entry pointing to the root document.

When opening a repository, all mounted partitions are scanned, dereferencing the first document directory entry, until the root document of the repository is found.

This design is lightweight, robust and scalable for large instances. New partitions and repositories can be introduced without exclusively locking some global data structure. Documents describing individual collections only need to be locked when partitions are added or removed. Further, and more important, even if some of a repository's partitions are not currently mounted, because of maintenance or hardware errors, work can continue on the accessible data. Only the master partition needs to be readable to open a repository.

6.4.5 Document Schemas

Document schemas are treated like regular documents and are stored in a special collection, as indicated by the `schemacollection` association in Figure 6.3.

As `schemaID` in the document directory, the schema documents use special reserved `DocumentID` values indicating whether a DTD or an XML Schema document is present.

DTDs are first converted to an XML representation. Internal DTD fragments are also converted to XML documents, but in addition are redundantly stored verbatim as a special text node in the document for easier export.

The association `accept` from Figure 6.1 is materialized in the repository catalog document for the document collection. For increased access speed, this information is cached in the main memory objects used to access the document collections currently in use.

6.4.6 Indexes

Each index is stored using a single segment (hence the limitation to single partitions). An XPathIndex is implemented using a BTreeSegment, while a FullTextIndex is implemented using a special segment type [58].

6.4.7 Example

As an example for a materialized repository schema, we now list segments and objects required to store a sample logical and physical schema for a repository called ’PaperNews’, designed to manage documents for a small magazine.

The logical schema contains two document collections, called `MyArticles` and `ResearchMaterial`. `MyArticles` may only hold documents conforming to the DTD of the local authoring system, `articlewriter.dtd`, while there is no particular schema associated with the `ResearchMaterial` collection.

There are two partitions in the physical schema, `part1` and `part2`. Both are assigned to `ResearchMaterial`, while documents for `MyArticles` may only be stored on `part2`. `part1` is the master partition of the repository.

In the `MyArticles` collection, only a single article document (`newarticle.xml`) is stored, which is valid with respect to `articlewriter.dtd`, while `ResearchMaterial` contains two (`stockquotes04072002.sqx` and `ciafactbook.xml`), both of which are not associated with a document schema.

To materialize the repository schema, Natix creates a `#catalog` document collection and a `#schema` document collection as described in Sections 6.4.4 and 6.4.5. By default, these are assigned to the master partition `part1`.

The required segments to materialize the schema are listed in Figure 6.4. The segment names comprise a type prefix (dd for document directory, and xd for XML data), the associated collection’s name, if any, and the repository name. The objects stored in the document

No	Part.	Type	Name	Purpose
0	part1	DirectMap	dd:PaperNews	DocumentDirectory for `part1`
1	part1	XML	xd:#catalog:PaperNews	Catalog document contents
2	part1	XML	pd:#catalog:PaperNews	Catalog document properties
3	part1	DecTable	dt:#catalog:PaperNews	tag names for 1+2
4	part1	XML	xd:#schema:PaperNews	Schema document contents
5	part1	XML	pd:#schema:PaperNews	Schema document properties
6	part1	DecTable	dt:#schema:PaperNews	tag names for 4+5
7	part1	XML	xd:ResearchMaterial:PaperNews	`ResearchMaterial` documents
8	part1	XML	pd:ResearchMaterial:PaperNews	`ResearchMaterial` properties
9	part1	DecTable	dt:ResearchMaterial:PaperNews	tag names for 7+8
10	part2	DirectMap	dd:PaperNews	DocumentDirectory for `part2`
11	part2	XML	xd:ResearchMaterial:PaperNews	`ResearchMaterial` documents
12	part2	XML	pd:ResearchMaterial:PaperNews	`ResearchMaterial` properties
13	part2	DecTable	dt:ResearchMaterial:PaperNews	tag names for 11+12
14	part2	XML	xd:MyArticles:PaperNews	`MyArticles` documents
15	part2	XML	pd:MyArticles:PaperNews	`MyArticles` properties
16	part2	DecTable	dt:MyArticles:PaperNews	tag names for 14+15

Figure 6.4: Segments required for example schema

Part.	Segment	Objects
part1	dd:PaperNews	Entry for the `#catalog` meta document
		Entry for the `#schema` meta document
		Entry for the `ResearchMaterial` meta document
		Entry for the `MyArticles` meta document
		Entry for `articlewriter.dtd`
		Entry for `ciafactbook.xml`
part1	xd:#catalog:PaperNews	`#catalog` schema document tree
		`#schema` schema document tree
		`ResearchMaterial` schema document tree
		`MyArticles` schema document tree
part1	xd:#schema:PaperNews	`articlewriter.dtd` document tree
part1	xd:ResearchMaterial:PaperNews	`ciafactbook.xml` document tree
part2	dd:PaperNews	Redirect entry to `#catalog` collection document
		Entry for `stockquotes04072002.sqx`
		Entry for `newarticle.xml`
part2	xd:ResearchMaterial:PaperNews	`stockquotes04072002.sqx` document tree
part2	xd:MyArticles:PaperNews	`newarticle.xml` document tree

Figure 6.5: Segment contents

segments and the document directory segments are shown in Figure 6.5. We do not show the contents of the property and declaration table segments. The property documents contain just the document names and no other properties by default, and the declaration tables' contents are straightforward.

Chapter 7

Recovery

A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a peon, a canal – Panama!

–Guy Steele, Jr.

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

To be an effective tool for enterprise-level applications, Natix therefore must provide transaction management for XML documents with the above-mentioned properties.

This chapter deals with the realization of the atomicity and durability properties of transactions.

We state our goals for design and implementation of our recovery subsystem in Section 7.1. This includes our assumptions about the used concurrency protocols because it is impossible to design a recovery method without knowing what the allowed degrees of concurrency are. The design and implementation of the concurrency controller itself is beyond the scope of this work.

Natix uses an extended version of the well-known ARIES protocol [66] for recovery, which is summarized afterwards, together with more recent extensions for two-level object recovery (Section 7.2).

After outlining the architecture of the recovery subsystem (section 7.3) and showing how it interacts with the storage subsystem, we thoroughly discuss the design details and implementation issues in Section 7.4. Although much progress has been made on the theory of database recovery, and the standard recovery protocols have been well-studied in the last decades, implementing a recovery subsystem remains a complicated and easily underestimated task. This becomes especially apparent when we discuss Natix's *metadata recovery* in Section 7.5, a topic that is usually ignored in the recovery literature.

Finally, we introduce some novel techniques which exploit certain opportunities to improve logging and recovery performance. These techniques prove to be particularly effective with our XML storage format, although they are applicable in many other environments.

The new techniques are called *subsidiary logging*, *annihilator undo*, and *selective restart* and are dealt with in Sections 7.6, 7.7 and 7.8, respectively.

7.1 Goals

Apart from the basic functional requirement to provide atomic and durable transactions to applications, there are a number of goals that were important factors in the design of Natix's recovery subsystem.

They fall in two categories: Performance goals, and software engineering goals.

High performance is always of utmost importance for any database management system, which is why performance related goals receive highest priority. When dealing with recovery, there are several subgoals to consider, which have in common that it is easy for the recovery subsystem to jeopardize performance improvement techniques in other parts of the DBMS.

small overhead Recoverable operations put a strain on the DBMS because to allow recreation of previous states requires that they must be stored in some way.

No matter if this is done using an operation log or using shadow copies [35], it means that all data modified by a transaction must be copied to at least two locations, the regular one and a location used by the recovery system to remember the old state.

By careless design, it is very easy to introduce additional representation changes and copy operations on the data until it reaches its final locations.

Since such copy operations and representation changes are very expensive, we avoid them wherever possible.

high concurrency Recovery should allow highly concurrent access to the data. A transaction should not block other transactions if they do not conflict. We can identify the following subgoals:

- Allow record-level locking. Note that even document-level locks may turn out to be record-level locks because there may be several documents on a single database page.
- Allow commutative operations on multi-page data structures to proceed concurrently. This is not only important for index structures, as in traditional DBMS, but also for XML documents, if it becomes necessary for the system to support several transactions which work on the same document concurrently.

parallel processing The previous goal is related to the concurrent execution of transactions. In addition, concurrent threads of execution should be supported efficiently, not only because multi-threaded execution is a straight-forward way to implement concurrent transaction, but also because the computing power of symmetric-multiprocessing (SMP) systems can only be exploited if concurrent threads can operate efficiently.

For example, access to the above-mentioned storage of old states (log or shadow copy) must be synchronized. If the design of the recovery system does not allow for high parallelism on the data structure used to store old states, all efforts undertaken in the rest of the system to allow a high degree of update parallelism are futile because the degree of update parallelism on the data structure is an upper bound for update parallelism of the rest of the system.

XML support As we want to build a native XML DBMS, the typical access patterns when dealing with XML documents must be supported efficiently.

Recalling the storage technique for XML presented in Section 5.7, this means we must be able to deal with a few large records on each page, and sequences of updates on the same record by one transaction. This contrasts with typical traditional DBMSs, which deal with lots of small records on each page, and each transaction only modifies a record ideally at most once. This also emphasizes the previous goal: Since the typical update granularities for XML are documents and subtrees, which are larger than the typical small relation rows, additional copy operations and representation changes will considerably slow down an implementation.

On the other hand, the conventional access pattern must still be supported well, as metadata and indices reside in more traditional structures.

In addition to the performance goals, there are goals which relate to the quality and maintainability of the system. Transaction processing has a higher need for quality than the rest of the system. Errors in the rest of the system are supposed to have less impact because affected transactions can be aborted, and their changes are logged and can be undone. Recovery subsystems, on the other hand, are complicated and very subtle in their collaboration with the rest of the system.

Good software engineering can produce higher quality and maintainability. Therefore we will reiterate some software engineering goals and their relation to Natix's recovery system:

extendibility The recovery subsystem should have an architecture that allows to introduce new data types and recovery techniques. XML processing is still a young discipline, and new storage and recovery techniques are bound to emerge.

loose coupling Too many dependencies between subsystems make software more difficult to understand and less robust against changes and extensions.

By limiting the dependencies of the storage system to the recovery system, we facilitate nonrecoverable versions of Natix, allowing to stabilize the core functionality first, without worrying about recovery effects.

By limiting the dependencies of the recovery system to the storage system, we support extendibility and more general code which is easier to understand.

By limiting the dependencies of the recovery system to the concurrency controller to assumptions about the allowed degree of concurrency, we decrease the recovery systems complexity, and increase its applicability in other environments. A prime example for such a complicated dependency between recovery and concurrency controller can be found in Mohan et al. [65].

reuse Reducing the total amount of code is another way to decrease the complexity of the system. By factoring code and making it more general, it becomes simpler.

For recovery processing, a concrete example is regular processing and transaction recovery. If a transaction aborts and rolls back all its changes, this can be transformed into a transaction that decided to apply inverse operations for all changes it introduced, and then commits. The same code can be used for regular prcoessing and undo processing. In Natix's, we leverage such factorizations as much as possible.

The mentioned goals are contradictive. In database systems, it is often worth compromising the architecture for a significant performance improvement (Mohan et al. [63] provide an excellent example for tight coupling of recovery and concurrency management which tremendously improves performance). We will strive to find an architecture and interfaces that fulfill the software engineering goals in spite of performance requirements, either by evolution of the architecture, or by finding equally well performing alternatives that result in less perturbation of the architecture.

7.2 Recovery Method Introduction

The recovery subsystem's job is to ensure the atomicity and durability properties of the DBMS. *Atomicity* means that either all or none of the modifications of a transaction are reflected in the database state. *Durability* means that once a transaction commit has been returned to the application, the modifications done by the transactions are not lost even if the system crashes and the contents of main memory or disks are lost.

Among the different approaches to achieve atomicity and durability, the *ARIES method* (Algorithm for Recovery and Isolation Exploiting Semantics [66]) has been established as a reliable, extensible and maintainable tool to perform recovery. The amount of literature [69, 60, 76, 67, 61, 68, 64] on extensions and variations of ARIES corroborate the superiority of its approach.

We give only a brief overview of ARIES and begin by establishing a model of our system, defining the relevant vocabulary. Basic techniques for page-level recovery are treated next. The concluding subsection explains how ARIES (and its generalizations) deal with multi-page data structures that need recovery while allowing concurrent access by more than one transaction.

7.2.1 System Model

We identify some properties of our storage engine (Chapter 5) and requirements which motivate why ARIES is a suitable foundation for a recovery method. We list a number of different types of failure the storage system may suffer from, needing to be addressed by recovery.

Application

From the view of the recovery system, the application program issues requests to begin and terminate transactions, and to read or modify persistent data structures on behalf of transactions it started. The application does not issue more than one request at a time per transaction, and the requests may depend on results of previous requests.

Database

Haerder et al. [37] classify database management systems according to some dimensions which affect the way they have to recover from failures.

Buffering We call the contents of the pages as stored on partitions the *stable database*. When we consider the *buffered database*, we mean the pages' contents as stored in the buffer manager for buffered pages, and the contents of the pages on partitions for pages not in the buffer.

Nonatomic, *in-place* updates The Natix buffer manager has a $1 : 1$ mapping between page numbers (*PIDs*) and physical storage locations. If a modified page is dropped from the buffer, the new version is written directly to the original partition. Hence, it is not possible to transfer updates of a transaction to the stable database in an atomic way.

noforce Noforce means that we do not write all pages modified by a transaction to disk when this transaction commits. This would cause a very high random access I/O load on the buffer manager, which contrasts with our goal to achieve high performance. In addition, the buffer manager does not keep track which pages were modified by which transaction. Since we use fine-grained locking, this would require dynamic storage management for an $m : n$ mapping of transactions to modified pages, again harming performance.

steal In our storage engine, it is possible that some modified pages of not-yet-committed transactions are written to the stable database. Our buffer manager does not base its replacement decisions on the transactions that have accessed certain pages. Doing so would make the replacement algorithm much more complicated and would impair I/O performance because scheduling of disk arms would be less effective when some pages may not be written.

Failures

With a storage organization as explained above, we have to consider several types of failure which must be addressed by recovery.

transaction failure A transaction may be aborted, for example because the user has pressed some kind of ”cancel” button, or because a programming error in the application has caused its thread of execution to be terminated, or because access conflicts like deadlocks have caused the DBMS to terminate the transaction.

main memory loss If the system crashes, due to a power loss or a programming error inside the database management system for example, the contents of main memory, the buffered database, is lost, while the stable database is unharmed.

secondary memory loss A device error, a hard disk head crash for example, may cause the stable database to be lost.

Recovery from each of the above-mentioned situations is assigned a level from 0 to 2:

R0 recovery considers how to undo a single transaction’s effects. This may require to access both the stable and the buffered database, since updates of a transaction may be stored on partitions before the transaction commits.

R1 recovery guarantees (1) that updates of transactions that did not commit before the system crash are removed from the stable database, and (2) that all updates of committed transactions are present in the stable database.

Special care must be taken of the fact that R1 recovery may be interrupted itself by a main memory loss. To work properly, R1 recovery must be idempotent, i.e. it must be possible to run R1 recovery several times on a crashed system state.

R2 recovery To recover from storage device failures, backups of the stable database must be regularly made, and incoporated into the recovery subsystem.

We do not dicuss R2 recovery in the following to limit the amout of detail that has to be presented.

7.2.2 Recovery for L0 Operations

We explain how ARIES implements R0 and R1 recovery in the case where objects do not span several pages, and the granularity of locking is the object. In particular, this means that objects are not modified by a transaction before all the transactions that have also modified the same object have committed.

In this case, a transaction can be undone by applying reverse-order, page-level inverse operations for all operations the transaction has performed. We call them *L0 operations*, since the application of inverse operations occurs on the lowest level, the page.

Overview

ARIES records all updates made by every transaction in a log. This log is an ordered collection of log records, as described below. Since we employ the *steal* policy in our storage engine, the log needs to contain information about how to undo each update, to provide R0 recovery, and during R1 recovery to remove modifications that were stored on stable storage by transactions that did not commit before a system crash. Because we employ a *noforce* policy, during R1 recovery ARIES must be able to incorporate effects of committed transactions that were not stored in the stable database before a crash. Hence, the log must also include information about how to redo each update.

During R0 and R1 recovery, ARIES reads and interprets the log entries and reapplies or reverses updates. A detailed decsription of the employed data structures and algorithms follows.

The Log For each execution of an update operation on an object, a *log record* is created.

Log records are assigned a strictly monotonic increasing number called *log sequence number*, or *LSN*. A special Null LSN value exists which is not used for any log record.

Log records are placed in a volatile *buffer* upon creation. At certain times, this *buffered log* is transferred to nonvolatile storage. The part of the log that is stored on nonvolatile storage is called the *stable log*. Transferring the buffered log to the stable log is called *flushing the log*. A log record may only be flushed if all log records with a smaller LSN are also part of the stable log afterwards.

ARIES log records contain the following fields

Type can be a regular *update log record*, *compensation log record* (*CLR*), *commit log record*, *abort log record*, or checkpoint related log records (see below). An additional log record type *L1 subcommit log record* is introduced in Section 7.2.3.

TransID Identifier of the transaction to which the log record belongs.

PrevLSN LSN of the previous record written by the same transaction, or Null if it is the transaction's first log record.

PID Identifies the page on which the update occured for update or compensation log records.

UndoNxtLSN In compensation log records, it describes the next log record that has to be undone after this log record was undone.

Data Contains information depending on the log record type.

For update and compensation log records, it contains redo information describing which object on the page is affected, and how to redo the update. This information may be stored either in a physical manner, by using an *after image* of the page or parts of the page, or in a logical manner, by only storing the applied operation and its parameters (”increase field A of record r by 3”). The latter kind of update log record, which is physical to a page, but logical within the page, is called *physiological log record*.

For regular update log records, the data also describes how to undo the update, i.e. how to apply the inverse operation.

The log contains all the information necessary to make transactions durable because if not all operations of a transaction have been stored in the stable database, the missing updates can be redone using the stable log information. The log also contains all the information to make transactions atomic because all stable database updates by partly executed transactions can be undone using the information in the stable log.

Pages Each database page in ARIES contains a `pageLSN`. This field contains the LSN of the most recent operation that modified the page. Since LSNs are monotonically increasing, this allows for ARIES to determine the state of a page, i.e. whether certain updates are applied to a page or not.

Dirty pages ARIES requires a *dirty page table* in main memory, containing the set of pages that are different in the buffered database with respect to the stable database. For each page that has been modified in the buffered database and that is not yet included in the stable database, the dirty page table contains an entry (`PID`,`RecLSN`). The `RecLSN` contains the first update operation on a page since it was last written to the stable database.

Modes of Operation To explain the operation of ARIES, it is helpful to divide the actions of the DBMS into several *modes of operation*. The modes of operation are not *phases* because some of them may be in effect concurrently. The modes of operation are not *modules*, because support for them is distributed throughout the recovery subsystem.

The modes of operation are

Forward processing Regular operation, where transactions start, manipulate data, and end.

Checkpointing When taking a checkpoint, the system records information in the log to speed up restart processing.

Analysis When restarting the system, the log is analyzed to determine whether the system has been properly shut down the last time round. If not, some transactions may not have completed, and their changes need to be undone, while some completed transactions may not have all of their modifications reflected in the database. Such cases are detected by analysis, in which case R1 recovery is initiated, using some of the information gathered during analysis.

Redo processing If R1 recovery is necessary, ARIES first performs redo processing, during which the state of the database as of the time of the crash is reestablished. All updates by all transactions in the log which have not been stored on disk are redone. This includes all updates of incomplete transactions as well. This *repeating of history* is central to the correct operation of ARIES.

Undo processing During undo processing, the system applies inverse operations for all logged operations of one or more transactions. This is necessary for both R0 and R1 recovery, to ensure the atomicity of transactions. Undo processing is invoked after redo processing during restart to remove the effects of transactions that did not run to completion, and during normal operation if a single transaction aborts.

Each of the modes of operation is the topic of a separate section below, explaining algorithms and data structures necessary.

Forward Processing

We call regular system operation *Forward Processing*.

When a transaction starts, it is entered into the transaction table, and its `lastLSN` and `UndoNxtLSN` fields in the transaction table are set to Null.

Every time a transaction performs an update to a page in the buffered database, a log record describing the update is written to the buffered log. Performing and logging the update must be atomic, i.e. it must be impossible that another transaction may access the updated page before the update has been logged. The log record's LSN is recorded in the transaction's `lastLSN` and `UndoNxtLSN` fields in the transaction table, and also in the `pageLSN` field of the updated page.

If an operation op modifies a page p that was not in the dirty page table before, p is inserted into the dirty page table. As `RecLSN`, the LSN of op's log record is used.

Before a page is written back to disk, the log is flushed up to the `pageLSN` to ensure that all updates contained in the stable database have associated log records in the stable log which describe how they can be undone. This is called *Write-Ahead-Logging*, or *WAL* for short[1].

Before a transaction commits, it writes a *commit log record* and flushes the log up to that log record's LSN. The transaction may only be considered comitted after the flush operation is completed. This ensures durability of committed transactions.

A transaction is aborted by invoking Undo processing (see below) and writing an abort log record. The log does not need to be flushed in this case.

Checkpointing

A *checkpoint* is taken by the system in regular intervals, after restart, and before shutdown. During a checkpoint, information about the current system state is written to the log. When the system restarts, log analysis (see below) uses the information from the last completed checkpoint to limit the amount of log that has to be scanned to perform R1 recovery.

When a checkpoint is taken, ARIES first creates a begin checkpoint log record. Then, log records describing the current dirty page table and the current transaction table are written. The checkpoint is completed by an end checkpoint log record, and flushing the log up to this record.

Finally, the LSN of the begin checkpoint record is stored at a well-known, absolute position on stable storage.

[1]There are ambiguous defi nitions of WAL in the literature. Sometimes it includes flushing of the log before a transaction commits, but in most cases WAL only refers to log flushes that precede writes of dirty buffer pages. We use it in the latter sense.

While the checkpoint is processed, the normal operation of the system proceeds. Transactions may continue to update pages, be created, or be terminated.

Analysis

After the system is restarted, the log is analyzed by scanning it from the most recent checkpoint to the end. The LSN of the begin checkpoint record can be found in an absolute, well-known location on stable storage.

Using the information from the checkpoint log records, the table of active transactions and the dirty page table are reconstructured.

Then, for every log record, they are modified as follows: If an update log record or a compensation log record is found for a page that is not yet in the dirty page table, the page is inserted into the dirty page table, using the log record's LSN as `RecLSN`. If a log record is found for a transaction that is not yet contained in the transaction table, the transaction is assigned an entry in the transaction table. If a commit or abort log record is encountered, the associated transaction is removed from the transaction table. For regular log records, the `UndoNxtLSN` field of the transaction is set to the log record's LSN. For compensation log records, it is set to the log record's `UndoNxtLSN` value.

As a result, the analysis produces the dirty page table and transaction table as of the time the system crashed. Updates on pages in the dirty page table may have been lost because they have not been transferred to the stable database. Those updates may have to be redone. All the transactions in the transaction table after analysis are incomplete transactions which did not commit, and are called *loser* transactions. Updates of those transactions have to be undone.

Redo Processing

Redo processing is invoked if the dirty page table is not empty after analysis.

The minimum of all `RecLSN` values in the dirty page table is used as `redoLSN`, and determines the oldest log record that may have to be redone.

The log is scanned from that log record, and every time a regular or compensation log record is read, it is first checked whether the affected page is in the dirty page table. If not, updates on that page do not need to be redone. Otherwise, the log records LSN is verified to be greater or equal to the `RecLSN` of the page in the dirty page table. If this is not the case, we know that this log records update is already contained in the stable database. If the update may have to be redone, the page is brought into the buffer, and it is checked whether the `pageLSN` is greater than the log record's LSN. If so, the update is already on the page.

Only if all the checks (dirty page table containment, `RecLSN` check, `pageLSN` check) indicate the update has to be redone, it is reapplied to the page, and the `pageLSN` is set to the log record's LSN.

After the redo pass, all updates in the stable log are guaranteed to be contained in the buffered database.

Undo Processing

If a transaction aborts (R0 recovery), or if there are active transactions in the transaction table after analysis (R1 recovery), then undo processing is required.

R0 Recovery When a single transaction needs to be undone, then ARIES scans its log records in reverse order and undoes the changes.

The reverse scan of the log is a loop that examines the log record the transaction's `UndoNxtLSN` field points to, which is always the next log record that needs to be processed.

Regular log records contain undo information necessary to apply the inverse operation to the modified page. In addition to applying the inverse update, a log record is written. In contrast to forward processing, this log record is not an update log record, but a compensation log record without undo information. The `UndoNxtLSN` field of the compensation log record is set to the undone log record's `prevLSN` value. The `pageLSN` of the affected page is set to the compensation log record's LSN. The transaction's `UndoNxtLSN` is set to the log record's `prevLSN`, and undo continues.

When encountering compensation log records, the log record does not contain undo information and does not need to be undone. The transaction's `UndoNxtLSN` is set to the compensation log record's `UndoNxtLSN` field, and undo continues.

The reverse chain of log records of the transaction is followed until the transaction is completely undone, which is indicated by the pointer to the next log record to undo being Null.

R1 Recovery After Redo processing, ARIES performs Undo processing for all loser transactions. This Restart or R1 Recovery Undo processing is nearly the same as R0 Recovery Undo Processing as explained above. The only difference is that to undo all transactions which are still in the transaction table, only one combined reverse log scan is performed. Hence, in every step of the undo loop described above, processing continues with the transaction whose `UndoNxtLSN` value is greatest.

7.2.3 Recovery for L1 Operations

We explain how ARIES can be extended to deal with *L1 operations*, which allow concurrent modification of objects by several transactions.

Motivation

For some data structures in certain applications, the limited concurrency allowed by L0 operations is inacceptable. For example, suppose that every transaction which modifies a B-Tree indexs needs to lock the index tree, or parts of the index tree, until the transaction completes. Then all other transactions accessing the same index or part of the index are blocked. For important, frequently used indices, this effectively results in a serialization of the transactions.

The strict limits on concurrency imposed in the previous section are not necessary to guarantee the serializability of the transactions. There are several possibilities to employ locks beyond page-level read/write locks. These techniques exploit knowledge about the object-level operation semantics [50, 81, 83] to allow a higher degree of concurrency while still retaining serializability. For example, if two transactions T_0, T_1 each add one key-value pair to the index each, those operations commute as long as they do not affect the same key. The end result, no matter in which order the two pairs are added, is an index containing both key-value pairs. Both operations can be undone independently, and the order of their application is not relevant.

Unfortunately, this is only true on a logical level. If such operations were allowed to be performed concurrently, the L0 recovery method described in the previous section would no longer work, since the B-Tree index is a dynamic multi-page data structure. Suppose transaction T_1 adds the key-value pair (k_1, v_1) to the index leaf page p. Afterwards, T_2 tries to insert (k_2, v_2), which causes p to be split into two pages, restructuring the inner nodes of the tree and moving (k_1, v_1) to a newly allocated page p'. If T_1 now aborts, the update log record written for the insertion of (k_1, v_1) can no longer be processed because undoing it would try to remove (k_1, v_1) from page p, where it no longer resides. Even worse, if another transaction T_2 had inserted (k_3, v_3) on page p' after the split, then an abort of T_1 would also remove T_2's update because p' would be deallocated as part of undo processing for T_1. T_2's update would be lost.

To allow increased concurrency on structures as B-Trees, we need to make the recovery subsystem aware of the ”logical” level on which the operations commute. The operations on the multi-page data structure level are called *L1 operations*, complementing the L0 operations on single pages.

Higher levels of concurrency could be supported if the recovery subsystem made it possible to logically undo L1 operations. Logical undo means that, upon undo, the multi-page data structure is examined to determine which operations on which pages are necessary to undo the update now, taking into account updates to the data structure that have occured after the original modification, but before the undo.

Overview

Introducing L1 operations to the recovery subsystem is possible with only a limited amount of additions to the methods outlined in Section 7.2.2.

Essentially, each L1 operation on a multi-page data structure is treated as a small subtransaction, which uses page-level (L0) log records to describe its updates. In its commit log record, called *subcommit log record,* the subtransaction contains information about how to logically undo the L1 operation. In addition, the subcommit log record points, using its `UndoNxtLSN` pointer, to the last log record that was written *before* the L1 operation was performed.

Now, the log contains enough information to redo the L1 operation, namely its constituent L0 operation log records. It also contains the information to logically undo the operation, in the L1 subcommit log record. The logical undo supersedes undo of the L0 operations which make up the L1 operation. Which log records have been superseded and must be skipped by regular L0 undo can be determined by using the subcommit log record's `UndoNxtLSN`.

A similar concept was already mentioned in the original ARIES paper [66] under the name of *nested top actions*, for the case where the logical undo operation is the identity. This was extended to B+-Trees in the ARIES/IM paper [67]. A more general treatment of concurrency control and recovery on multiple levels can be found in Weikum et al. [93]. A very elaborate exposition of the topic is part of Weikum et al. [94], on which much of our terminology in the subsequent sections is based.

We now elaborate on the modifications necessary to make L1 recovery work in the different modes of operation.

Forward Processing

Upon execution of an L1 operation, its page-level updates are logged as regular log records. When the operation completes, a subcommit log record is written, containing information about how to undo the L1 operation on the logical level.

For example, a transaction inserts a record r on database page p, writing a log record with LSN l_1. It then updates an associated, unique index by adding a key-value pair (k_1, v_1). To do this, it first performs a search in the B+-Tree to find the page q on which key k_1 has to be placed. Since q has no free space left, it has to be split. The split occurs and regular L0 log records are written to describe it. Then, (k_1, v_1) is inserted on the now only half-full q, and a regular log record for this insertion is written. Finally, a subcommit log record for the insertion of (k_1, v_1) is written. Undoing the insertion means to delete (k_1, v_1) from the index. Since it is an unique index, k_1 uniquely identifies the key-value pair. k_1 is sufficient to find the key-value pair even if it has been moved to another page by the time an undo is required. Hence, k_1 is contained in the subcommit log record as logical undo information. The `UndoNxtLSN` pointer of the subcommit log record is set to l_1.

The log does not need to be flushed to the subcommit log record because the subtransaction only needs to be durable if the parent transaction commits, in which case the log is flushed anyway.

Checkpointing, Analysis

Recovery for L1 operations can use the same checkpoint and analysis algorithms as L0 recovery.

Redo Processing

The L0 records written during execution of the L1 operation already contain all the information necessary to redo the L1 operation. Therefore, during redo, the L1 subcommit log records are ignored, and the L0 records are redone as before.

Undo Processing

Undo processing for L0 log records proceeds as explained above.

When encountering an L1 subcommit log record, instead of undoing it on the page level, a high-level inverse operation is invoked to undo the effects. For example, in our B-Tree example from Forward processing above, the high-level inverse could simply be the B-Tree delete operation for k_1. This operation then performs a new search on the tree to find where the key-value pair associated with k_1 is located, and deletes it. This works even if k_1 was moved in the meantime.

As with L0 undo, the inverse operations need to be logged. However, the inverse operations required for L0 undo can be regarded as atomic, i.e. either an operation has been undone or not. For L1 operations, this is not the case. The inverse operation for an L1 operation is not necessarily atomic and may affect more than one page. This needs to be taken into account in the undo algorithm because a system crash may leave a partially performed inverse L1 operation behind.

undo partial inverse L1 operation It must be possible to undo partially executed inverse L1 operations. Hence, not compensation log records, but regular log records are written for the page-level operations performed on behalf of the inverse L1 operation.

redo of incomplete inverse L1 operation If undo of an L1 operation was incomplete before a crash, and the partial undo was itself undone by regular L0 recovery, the inverse L1 operation must be redone on the logical level.

This requires that the first L0 log record written as part of the execution of the inverse L1 operation has to be chained, using its `nextUndoLSN` field, to the subcommit record of the L1 operation that is being undone.

L1 restart idempotence Because restart itself may be interrupted by a crash, it must be able to run several times without undoing the same L1 operation more than once. This is achieved by using compensation log records, similar to L0 undo. The compensation log records for L1 operations are the subcommit log records of the associated inverse L1 operation and must be linked to the operation that precedes the forward L1 operation in the `nextUndoLSN` chain.

Note that undo of L1 operations with the above method violates a property of ARIES: The log volume for repeated restarts is not bounded any more. Suppose a crash occurs during undo of some L1 operation. The partial effects of the inverse L1 operation are undone and the logical undo is retried, again causing log records to be written. If the undo crashes again, then during the next restart there will again be compensation log records and log records for the retried inverse L1 operation, possibly ad infinitum. This is not acknowledged anywhere in the literature, and a solution does not yet exist.

7.3 Natix Recovery Architecture

The responsibilities for recovery are distributed over the system, and it is difficult to draw a sharp line which separates the storage subsystem from the recovery subsystem. Some classes perform tasks that are important for both subsystems, some are exclusively needed for recovery and not for storage, and some are completely unaffected by recovery.

The following subsections will describe the involved classes' responsibilities and how they collaborate to provide correct and efficient recovery.

The description of each class or class hierarchy is structured according to the modes of operation introduced in Section 7.2, namely Forward Processing, Checkpointing, Analysis, Redo Processing, and Undo Processing. After the distribution of the responsibilities has been outlined, the individual classes and interfaces are described in more detail, including implementation aspects.

This structure is different from typical texts on database recovery, which group descriptions not by class, but strictly by mode of operation. The classical way of exposition is better suited to bring across recovery *concepts*, while we want to describe a recovery *architecture*.

One of the main contributions of the first sections of this chapter is exactly this separation of the many different issues which have to be resolved for recovery into implementable and understandable parts that have clear interfaces and can be extended and maintained. The section's structure was chosen to emphasize this important part of the design. When introducing new concepts, in the final sections of this chapter, our focus will shift from the architecture to a top-level view of the system operation, again following the traditional outline of recovery.

The subsections for the individual classes and hierarchies are preceded by a short overview which is intended to provide better orientation in the following text.

7.3.1 Overview

An overview of the Natix components that are involved in providing atomicity and durability of transactions is shown in Figure 7.1, which depicts the classes, class hierachies and their call relationships in UML notation. UML class boxes whose name is followed by a $*$ represent base classes of important class hierarchies.

The **Segments** are large data structures used by the **application** to hold all persistent user data. The segment operations are mapped on page-level operations which are executed by calling **Page Interpreters**. For recovery, special page interpreters are used. During normal operation they create log records, and during undo, they interpret them. Page contents are transferred between main memory and secondary memory in the **partitions** using the **buffer manager** which also logs information about which pages are currently buffered. The **Transaction Manager** manages the active transactions in the system[2]. Apart from the segments, it is the only component directly called by **the application**, which uses the transaction manager to group user operations in transactions. The transaction manager is also used as a central storage location for per-transaction data. Finally, the **LogManager**

[2]It also serves as a storage manager for per-transaction data for all modules. This explains the dependency of the segments on the transaction manager, which causes a dependency cycle. This is not problematic as the segments only access the per-transaction data.

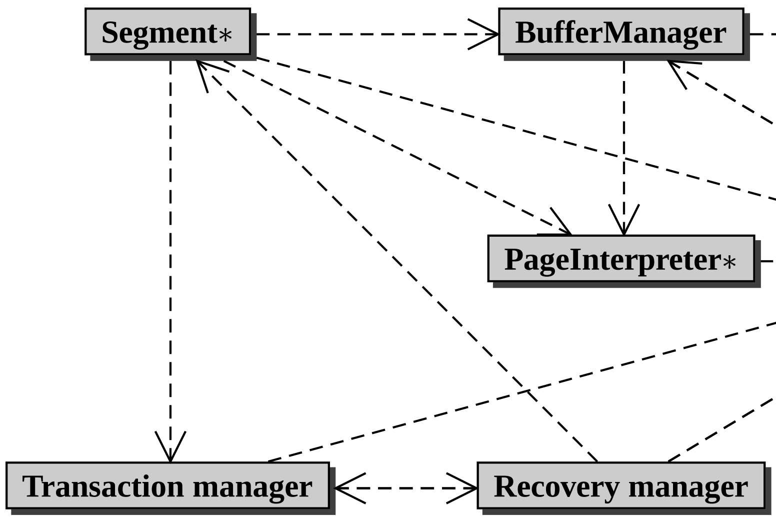

Figure 7.1: Recovery Components

manages read and write access to the recovery log, storing all log records in special partitions.

After briefly explaining the threading model of Natix recovery, we present the recovery responsibilities of those classes that also belong to the storage subsystem and whose duties for nonrecoverable operations were already detailed in Section 5. The additional classes necessary exclusively for transaction management follow. The section is concluded by sequence diagrams illustrating typical processes.

Explanations of critical dependencies between the classes are marked in the margin for ease of reference.

7.3.2 Threading Model

The design of Natix's recovery subsystem is based on the assumption that there is a $1 : 1$ relationship between concurrent threads of execution and update transactions. Each thread may only perform updates for one transaction, and each transaction has exactly one thread doing its updates.

If several threads were executing on behalf of the same transaction, their access to transaction-local data structures would have to be synchronized. Additional communication between the threads would be another consequence, as the threads would have to be coordinated if one of them decided to terminate the transaction, or to take a savepoint.

On the other hand, it seems a reasonable constraint to have application designed such that different threads always perform updates on behalf of different transactions.

Please note that in this model, for read-only transactions it is still possible to have more than one thread. Synchronization of threads for read-only transactions is simpler than for updaters. Since the recovery subsytem is not used by read-only transactions, we do not go into further detail here.

7.3.3 Segments

Implementation details in section 7.4.4

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.

Application $\longrightarrow$ Segments

The data structures provided by the segments are typically larger than a page. The segments map operations on these data structures to operations on single pages, operate on page interpreters to perform the page-level operations, and employ the buffer manager to transfer the pages between main memory and disk storage.

Segments $\longrightarrow$ PageInterpreter

Segments $\longrightarrow$ BufferMgr

From the perspective of the recovery architecture, there are three kinds of segments:

Simple Recoverable Segments Logging and recovery for operations on pages is dealt with by the page interpreters (see Section 7.3.4), which means that there is nearly no recovery-specific code to implement for recoverable segment types. Hence, the designer of a new multi-page persistent data structure can write a recoverable Natix segment type for it without having to worry about recovery. Only the underlying page interpreter must be capable of creating and interpreting log records.

Recoverable Segments With High Concurrency In Section 7.2.3, we explained why sometimes the desire for a high degree of concurrency requires to have operations on segments whose inverse operations are not described by the page-level inverse operations of the original action's execution. This kind of operations, called *L1 operations*, requires special recovery code in the segments.

Nonrecoverable Segments are typically used as storage location for intermediate results during query processing. Their contents need not to be recovered in case of a system crash, and they never need to survive the transaction which created them.

Still, operations on nonrecoverable segments can involve the recovery subsystem, if the segment is located on a partition together with recoverable segments. In that case, the resources used by the nonrecoverable segment have to be released when the creating transaction is undone. Section 7.5 deals with the necessary recovery mechanisms.

Nonrecoverable segments are treated in Section 5.5. The responsibilities and collaboration of the recoverable kinds of segments during the different modes of operation are as follows:

Forward Processing

During forward processing, a recoverable segment must use a page interpreter class that is capable of doing logging and recovery. Disregarding L1 operations, this is the only difference between segments that support recovery and segments that do not.

If the segment provides L1 operations, during forward processing it has to parenthesize every L1 operation with calls to the transaction manager, announcing the begin and end of the L1 operation. This allows to do housekeeping in the transaction control block. During the execution of the operation, page-level log records are generated as usual. In addition, the segment must write an L1 subcommit log record containing logical undo information after the operation has finished.

Segments $\longrightarrow$ TransactionMgr

Checkpointing

Segments are notified when a checkpoint takes place, allowing them to write information to disk that up to then was only maintained in main memory data structures for efficiency reasons.

Examples include administrative information like logical page tables and free space management information.

Redo Processing

The recovery manager forwards all log records that are associated with a segment and may have to be redone to the corresponding segment object. Although possible, the recovery manager does not forward the page-level log records directly to the page interpreters, but delegates page-level redo to the segments instead. As a result, page updates are handled in a uniform way during redo, undo and forward processing, including buffer management, free space inventory updates and maybe additional main-memory cache management for objects and metadata. This makes metadata recovery much simpler (see 7.5).

RecoveryMgr $\longrightarrow$ Segments

In case of a page-level log record for a page belonging to the segment, the segment checks the on-disk version of the page whether it already contains the update, and if not, forwards the log record to the corresponding page interpreter for redo.

L1 operations do not need special treatment during redo. Their effects are described by page-level log records written during their execution, which are redone using regular redo processing as above. The additional L1 subcommit records are only reuired for undo and ignored during redo.

Undo Processing

Undo processing for segments is similar to redo processing, in that log records to be undone are forwarded to the segment by the recovery manager.

For page-level log records, the only difference is that undo is unconditional: If an undoable log record is encountered while traversing the `nextUndoLSN` chain, it is not possible that the page already contains the undo operation. So undo log records are always executed by forwarding them to the page interpreter.

L1 subcommit log records contain the information necessary to logically undo the L1 operation using segment operations. To undo an L1 operation, the segment just invokes the inverse operation on itself, using the regular forward-processing functions to do so.

In Section 7.4.4 we describe how this causes correct log information to be written for the inverse L1 operations during undo.

7.3.4 Page Interpreters

Imple-mentation details in section 7.4.5

The page interpreter classes are responsible for page-level physiological logging and recovery. They create and process all page-level log records.

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.

Forward Processing

Upon a call to an update method of the page interpreter, the operation is performed and a log record describing it is written by calling the log manager.

PageInterpreter $\longrightarrow$ LogManager

The order of perfoming and logging the operation is arbitrary, as long as the buffer manager latch on the data page is not released in between. Logging and performing the operation may even be interleaved for performance reasons, for example to copy a before image directly from the object to the log record, then modify the object, then copy the after image directly from the object. The data page latch makes the whole action atomic from the perspective of other users of the same page, and no further synchronization is required.

Page interpreters also have to perform write-ahead-logging: The buffer manager notifies the page interpreter before a page is written to disk, so that the page interpreter can make sure the stable log contains all the log records associated with the page. The buffer manager could directly instruct the log manager to flush the log, but since it invokes a dynamic function on page interpreters before writing a page anyway (Section 5.4), we can avoid introducing addititonal assumptions, dependencies and code for write-ahead-logging in the buffer manager. Instead, we delegate them to the page interpreter, where a dependency to the log manager and knowledge about recoverability of the page already exist. In addition, sophisticated recovery techniques may require actions in addition to just flushing the log before writing a page to disk, see Section 7.6.

BufferMgr $\longrightarrow$ PageInterpreter

Redo Processing

If a page-level operation has to be redone during restart recovery, the corresponding log record is provided to the page interpreter via the segment. The page interpreter then reexecutes the operation coded in the log record, but without writing a log record as in forward processing.

The decision whether a log record needs redo is reached by the recovery manager and the segment, and not by the page interpreter itself. This avoids duplicate code.

Undo Processing

Undo of a modifying operation proceeds similar to redo. The page interpreter is told by the segment to undo a log record. The data necessary to apply the inverse operation is extracted, and the inverse operation is executed using the regular methods that are also employed during forward processing.

As result of using the regular forward-processing code to undo operations, log records are written during undo. The log manager knows that undo is in progress and performs proper `nextUndoLSN` chaining for compensation log records. Again, no code duplication is necessary.

Imple-mentation details in section 7.4.6

7.3.5 Buffer Manager

BufferMgr $\longrightarrow$ Partition

The buffer manager controls the transfer of pages between main and secondary memory in the form of partition object. Although ARIES is independent of the replacement strategy used when caching pages, the buffer manager facilitates recovery by notifying other components about page transfers between main and secondary memory, and by logging information about the buffer contents during checkpoints. In contrast to ARIES, the set of currently dirty pages, which have been modified compared to their on-disk version, is not maintained in a separate data structure during forward processing. In Natix, the information equivalent to ARIES's *dirty page table* is maintained directly in the buffer frame control blocks (Section 5.3).

Forward Processing

The buffer manager maintains the association between buffer contents and the corresponding page interpreter (Section 5.3). In the case of recoverable page interpreters, this automates synchronization of operation execution and log record creation.

Before flushing a dirty page to disk, the buffer manager notifies the corresponding page interpreter of the impending write operation to ensure write-ahead-logging.

BufferMgr $\longrightarrow$ PageInterpreter

The buffer manager also provides and logs information about dirty pages to assist redo recovery and log truncation.

Checkpointing

When the recovery manager notifies the buffer manager of a checkpoint, the buffer manager writes log records specifying the pages that are currently dirty.

BufferMgr $\longrightarrow$ LogMgr

Imple-mentation details in section 7.4.8

7.3.6 Recovery Manager

The recovery manager orchestrates system activity during system restart and checkpointing. After the application invokes the recovery manager in order to restart or checkpoint the system, the recovery manager initializes the data structures and manager objects necessary for the operation of the recovery subsystem.

Forward Processing

The recovery manager is not involved in regular system operations.

Checkpointing

A checkpoint is initiated by calling the checkpoint method of the recovery manager. The recovery manager then notifies the buffer manager, partition manager, segment manager, and transaction manager that a checkpoint is taking place, and each of them may write log records to describe their current state.

RecoveryMgr $\longrightarrow$ All

The recovery manager also writes begin and end checkpoint log records and instructs the log manager to persistently store the checkpoint LSN.

Analysis

During restart analysis, the recovery manager follows the ARIES restart analysis algorithm to determine the set of loser transactions and the set of dirty pages. This involves reading the log anchor and performing a log scan starting at the last checkpoint.

RecoveryMgr $\longrightarrow$ LogMgr

The transaction manager is notified of the loser transactions, which is necessary for undo processing.

RecoveryMgr $\longrightarrow$ TransactionMgr

Redo Processing

Restart redo is initiated by the recovery manager during restart. Following the standard ARIES procedures, the systemRedoLSN is determined, a log scan starting at that LSN is performed and all actions that are not recorded in permanent storage are redone.

The actual execution of redo actions is left to the segments (see above). The affected segment is determined from the log record and opened. The log record is further processed by forwarding it to the segment object. In case of page-level redo records, the recovery manager checks whether the affected page is contained in the dirty page set before calling the segment, avoiding to load segments and pages only to detect that the changes are stored on disk already.

Undo Processing

During transaction and restart undo, the recovery manager performs a backward log scan of the transaction(s) that has (have) to be undone.

Analogous to redo processing, the actual execution of the undo operations and writing of compensation log entries is delegated to the segments.

In contrast to redo handling, where the log records are filtered according to the dirty page set, the undo algorithm does not read page IDs from log records, but routes all log records that affect segments to the appropriate segment object. Proper actions for both segment-level (logical) undo and page-level undo are then taken by the segment.

Imple-mentation details in section 7.4.2

7.3.7 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 part of the log buffered in main memory using a special log buffer manager, and employs special partitions, log partitions, to store log records.

LogMgr $\longrightarrow$ Partition

The log manager maintains the mapping of log records to LSN (and its inverse), and persistently stores the LSN of the most recent checkpoint.

LogMgr $\longrightarrow$ BufferMgr

LogMgr $\longrightarrow$ BufferMgr

LogMgr $\longrightarrow$ TransactionMgr

Log truncation (see 7.2.2) is a responsibility of the log manager. Log truncation is the process of marking log pages as no longer used, and hence reusable. It explains the call dependencies to the buffer manager and the transaction manager, which are called to determine how much of the log is required by a restart.

Forward Processing

During forward processing, the log manager is called whenever log records have to be created. The log manager reserves space for the record in the log buffer and assigns an LSN (log sequence number) which identifies the log record. Using information from the transaction control block, the log manager chains records of the same transaction together to allow efficient undo.

The address of the log record in the log buffer and its LSN are returned to the caller and can then be used to write the log records content and to update any `pageLSN` fields etc.

The log manager provides calls to flush the log up to a certain LSN (as necessary for force-at-commit and write-ahead-logging).

Checkpointing

The log manager maintains an anchor record, an absolute storage location where it stores the LSN of the youngest checkpoint record and some additional administrative data.

Flushing of the anchor record to disk is triggered by the recovery manager.

Analysis, Redo Processing

During analysis and redo processing, the log manager is employed by the recovery manager to perform a forward scan of the log.

Undo Processing

The log manager allows to access log records by specifying their LSN, which is necessary when traversing the `nextUndoLSN` chain(s) during undo.

Log records are written during undo processing as well, since undo operations need to log compensation log entries. Instead of chaining a log record to the previous log record of the same transaction, the log manager detects if a transaction is rolling back, and in that case properly points the `nextUndoLSN` field to the operation that has to be undone next, as required by the ARIES protocol. As in forward processing, it is possible to override the log manager's choice for `nextUndoLSN` in certain situations, like L1 log records or nested top actions.

The automatic `nextUndoLSN` chaining by the log manager supports code reuse in recoverable page interpreters. They do not need additional code for performing and logging operations during undo. Instead, they may use the regular forward processing functions to undo operations, automatically generating compensation log records.

7.3.8 Transaction Manager

Imple-mentation details in section 7.4.7

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

Application $\longrightarrow$ TransactionMgr

The transaction manager maintains control structures for active transactions, called *transaction control blocks*. Application programs use the transaction manager to group their operations into (possibly nested) transactions. To indicate on behalf of which transaction an operation is performed, applications have to give a transaction control block as one of the arguments to their database operations.

The initiation of necessary actions when the state of a transaction changes is another responsibility of the transaction manager.

Each active transaction has one transaction control block, serving as a central repository for per-transaction information of the various transaction management components.

Apart from fields that contain the LSNs of the first and last written log records of the transaction, each control block includes a pending actions list.

Pending actions in Natix are more generic than those of ARIES [66], which only allow actions to be performed *after commit*. In Natix, the pending actions list contains sequences of operations for a variety of changes in transaction state, for example *before* a transaction commits, or before it establishes a savepoint. The pending actions list is a main memory structure and is used to maintain other recovery components' main memory structures. It may not be used to store information necessary to undo operations of the transaction that have been made durable (as the pending actions list may be lost in a crash). Examples for its use include metadata recovery (see for example Section 7.5.3) and subsidiary logging (Section 7.6).

The transaction manager also maintains a *system transaction*. This transaction is always active, and is never rolled back. It is used in those places where a transaction control block is required for operations that are not associated with any application transaction, for example when writing checkpoint log records. While it would be possible to implement a second code path for all those components which can be used without a transaction control block, doing so for only a few special cases seems to be a less elegant version than simply using a system transaction control block.

Forward Processing

Application programs use the transaction manager to begin, commit and abort transactions, to establish savepoints and perform partial rollbacks. The transaction manager writes log records accordingly, and if a rollback is requested, it employs the recovery manager to

TransactionMgr $\longrightarrow$ LogMgr

TransactionMgr $\longrightarrow$ RecoveryMgr

recreate a previous transaction state by means of the recovery manager's undo routines.

Checkpointing

During a checkpoint, the transaction manager will log the contents of the control blocks of all active transactions, to allow for recreating them after a crash. The contents of the pending actions list is not recorded in the transaction manager's log records.

Analysis

As specified in the ARIES protocol, when encountering commit or abort records during restart analysis, or when encountering a previously unknown transaction's first log record, the recovery manager instructs the transaction manager to allocate or release transaction control blocks accordingly.

RecoveryMgr $\longrightarrow$ TransactionMgr

The recovery manager also forwards the checkpoint log records describing active transactions to the transaction manager to allow for reconstruction of transaction control blocks for the transactions that were active during the checkpoint.

As a result, at the end of restart analysis, all transactions that were still active when the system crashed, and only those, will have a transaction control block.

Redo Processing

The transaction manager is not involved in redo processing, Since no log records or locks are created. However, most low-level components require properly initialized transaction control blocks to operate. To allow these components to be used during redo, the system transaction's control block is used.

Undo Processing

If a transaction was completely rolled back during redo processing, but no abort record was found, the recovery manager properly aborts the transaction using the transaction manager, resulting in an abort record being written.

All transaction control blocks are released at the end of the restart undo phase.

7.3.9 Control Flow Examples

To clarify control flow through the recovery subsystem, we present UML sequence diagrams showing the involved components and their interactions in three situations, forward processing, transaction undo (R0 Recovery), and system restart.

Forward Processing

Figure 7.2 depicts the insertion of a record into a slotted page segment.

The application calls the segment (1). The segment finds a potential candidate page for insertion in its main memory cache, which it fetches (2). However, the candidate page is not

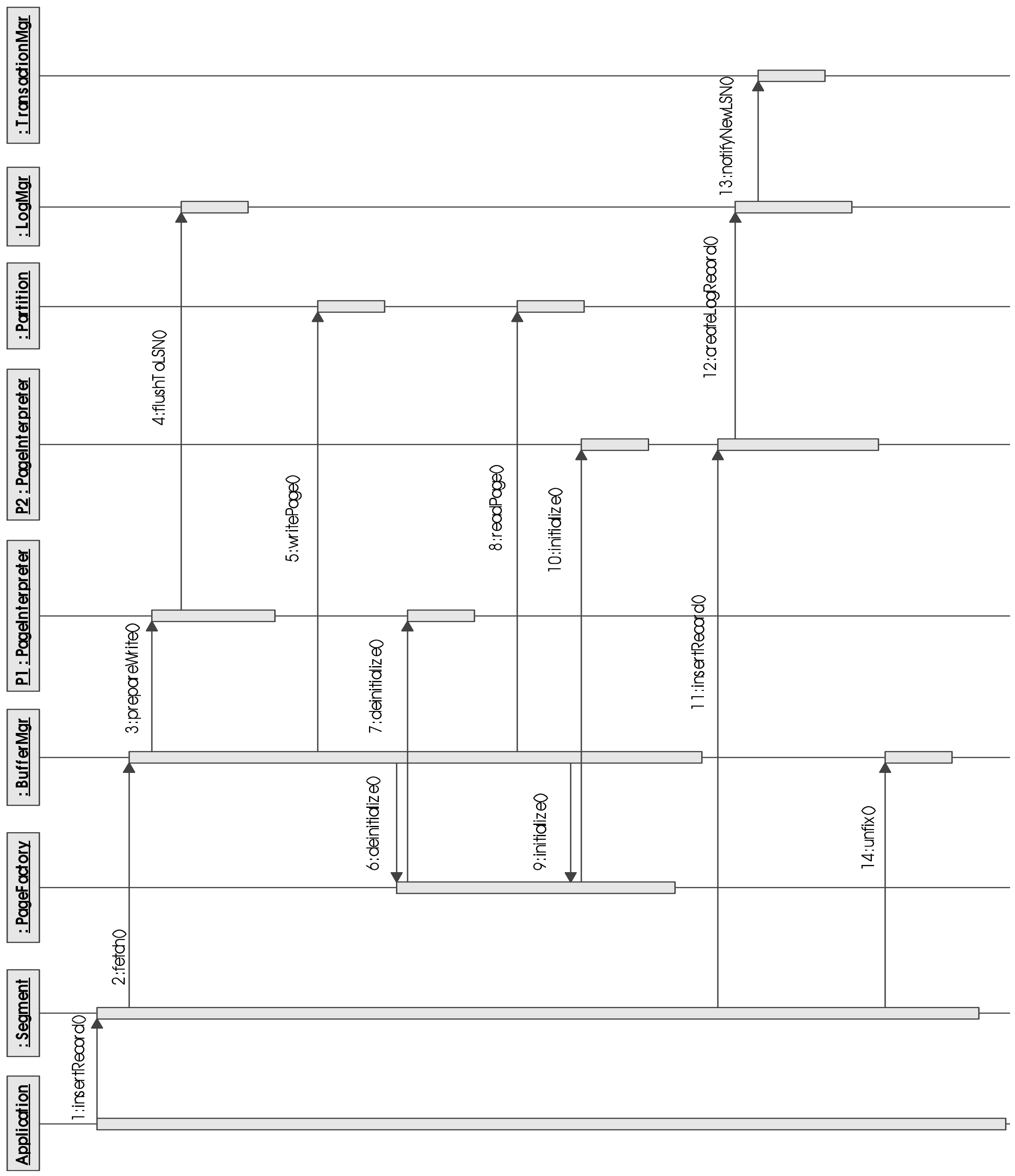

Figure 7.2: Forward processing sequence diagram

in the buffer. Hence, another page must be replaced. In this case, the replaced page happens to be a dirty page of the same segment. The buffer manager notifies the associated page interpreter (3), which performs write-ahead-logging for the page (4). The page is written to disk (5). Using the segment's page factory, the page interpreter is deinitialized (6), (7). The new page's contents are read (8) from disk, and the segment's page factory is called to provide a recoverable page interpreter (9), which is initialized (10). This completes the fetch operation. The segment now inserts the record on the page (11). The recoverable page interpreter, in addition to modifying the page's contents, creates a log record (12) using the log manager, which notifies the transaction manager to access the transaction information and maintain the undo LSN chain (13). The segment unfixes the page (14) and returns.

Transaction Undo

In Figure 7.3, the transaction from the previous example is aborted.

The applications requests the transaction abort (1) from the transaction manager, which uses the recovery manager to actually undo the transaction's updates (2). The recovery manager performs a backward scan of the log records written by the transaction, starting with the last written log record (3). It identifies the involved segment, and forwards the log record to it (4). The segment fetches the affected page (5), which is already in the buffer. The log record is further forwarded to the appropriate page interpreter (6), which calls its regular forward processing routines to perform the inverse operation (7). A compensation log record is automatically written (8), and the undo chain information for the transaction is updated (9). The segment unfixes the page (10). For a longer transaction, the steps would be repeated from (3), but in our example, only one log record needs to be undone. Finally, the transaction manager cleans up the relevant per-transaction resources and returns.

System Restart

In the final example (Figure 7.4), we show some of the steps executed during a system restart.

System restart is requested by system control (1), which uses the recovery manager to perform the necessary actions. The first phase is restart analysis (2), during which a log scan traverses the log from the starting point (3). The first log record reveals a transaction which was formerly unknown to the recovery system. Hence, the transaction is registered with the transaction manager (4). Further log records are scanned (5), (6), and the analysis phase adapts its dirty page accordingly. Here, it turns out that the final operation of the aborted transaction from our previous examples may not be stored on stable storage. At the end of the analysis phase, the page (P2) is in the dirty page table, and the LSN of the log record produced in step (8) of the previous example is the first log record that may need redo. During redo (7), a second log scan starts at that log record (8), and it is forwarded to the segment (9). The segment fetches the page into the buffer (10). Although the page is not in the buffer yet, we leave out the steps required to provide a page interpreter for the page to keep the diagram small. They are the same as in Section 7.3.9. After the buffer manager returns, the segment verifies that the `pageLSN` is smaller than the log record's

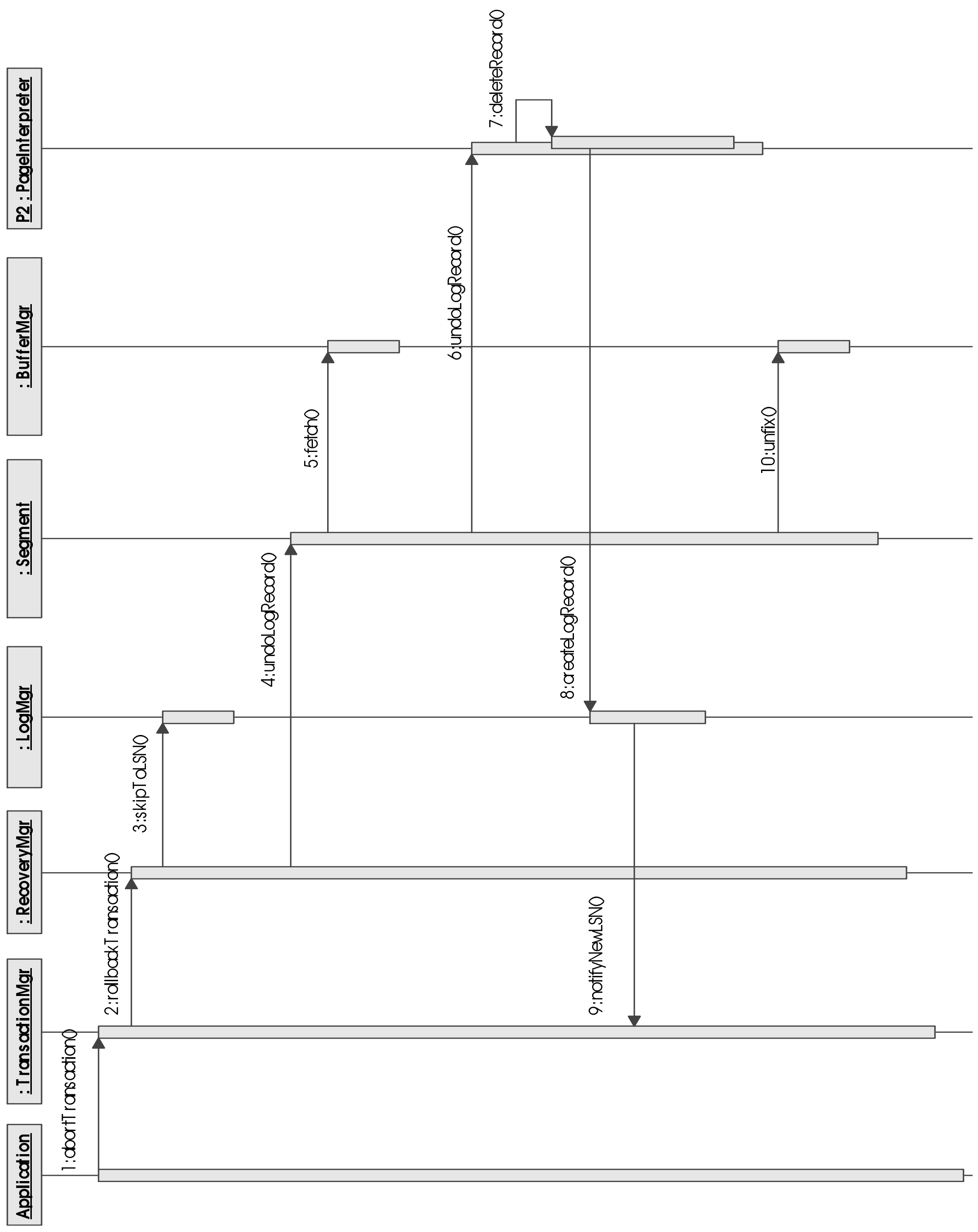

Figure 7.3: Transaction undo sequence diagram

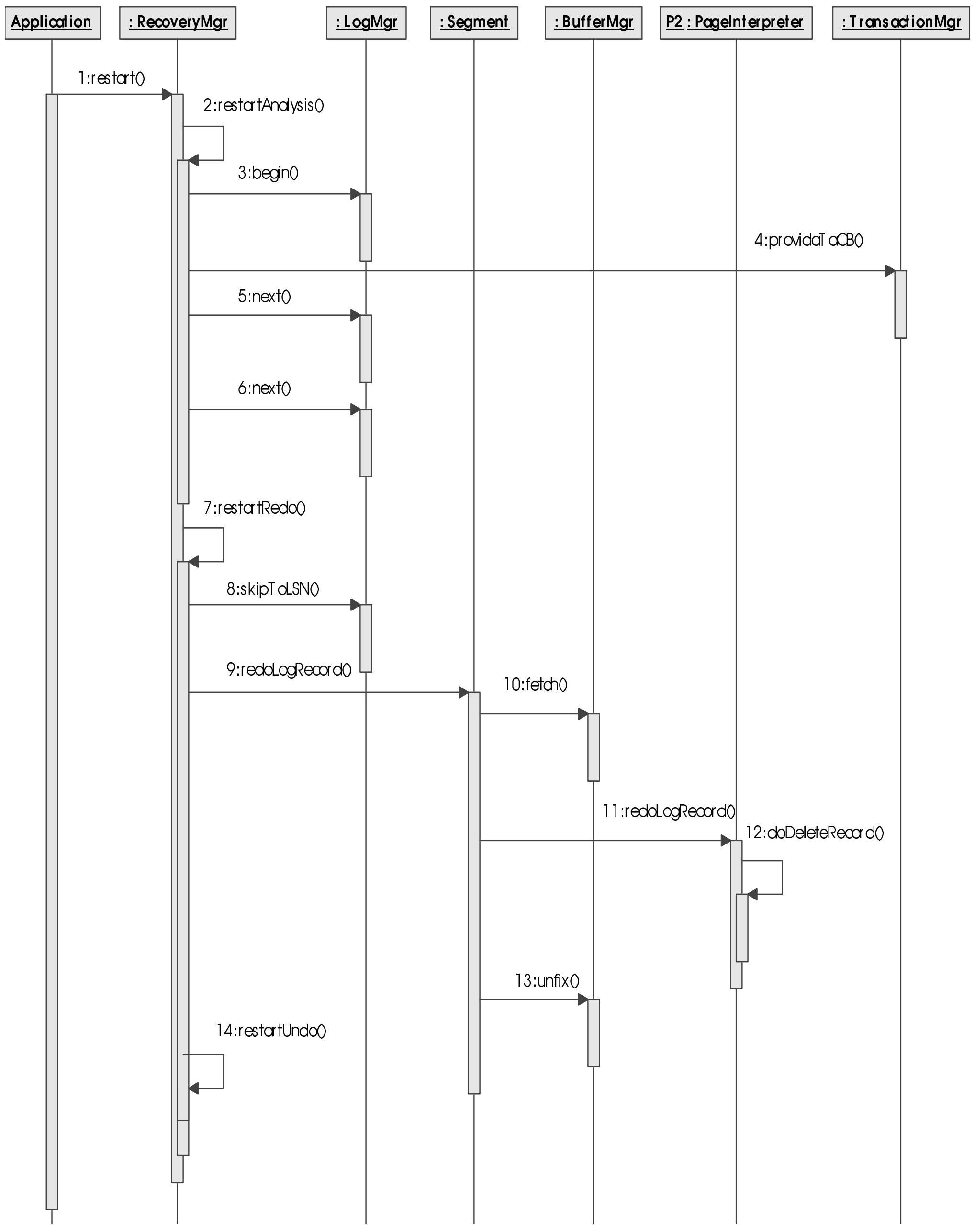

Figure 7.4: Restart sequence diagram

LSN. It is, and the log record is forwarded to the page interpreter (11). The page interpreter uses a non-logging call to redo the operation (12), and updates the `pageLSN`. The page is unfixed (13). No further log records need to be processed, and restart continues with undo processing. However, since the single active transaction already was aborted, nothing remains to be done.

7.4 Detailed Design and Implementation

The overall distribution of responsibilities and collaborations for recovery processing to the different system components has been outlined above. In the following, we provide details on interfaces and how the data structures and the communication of the classes are implemented to yield optimal performance. There is few implementation-oriented recovery literature. A lot of interesting details are not previously documented anywhere, making a complete discussion of all issues a daunting task. We focus on the more unusual and subtle parts and provide the basic material necessary to present them clearly.

Each subsection deals with one major class or class hierarchy in the recovery subsystem, explains how the goals outlined in 7.1 affect the class, and identifies additional requirements if they exist. General design principles of the class or hierarchy are derived, and a detailed description of the C++ interfaces follows. A discussion of implementation details conclude some subsections.

7.4.1 Log Records

Natix writes a recovery log which describes the actions of all update transactions using *log records*.

Log Sequence Number

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 records physical location on disk. The mapping is maintained by the log manager (Section 7.4.2).

The LSN is a very important concept for recovery, as it accurately identifies states of transactions, pages, and indeed the whole system. In the remainder of this chapter, several special names for LSN-valued functions and variables are frequently used. Table 7.1 gives an overview of their names, descriptions, transient/persistent storage locations. For reasons of clarity[3], we sometimes use other names than the original ARIES paper [66]. Wherever there is an equivalent in ARIES, we give its name in the table.

Table 7.2 is an index referring to the sections which explain the LSN values in more detail.

[3]. . .which is always in the eye of the beholder. However, the name `UndoNxtLSN` is even used with two different meanings in ARIES.

Name	Function of	Description	Storage	ARIES name
`anchorLSN`	log	LSN of most recent checkpoint	LM/LA	—
`annihilatorLSN`	object	LSN of last complete before image	V/V	—
`beginLSN`	log	smallest LSN needed for restart	LM/LA	—
`deletionLSN`	record	LSN of record delete operation	RC/RC	—
`flushLSN`	log	Log is on disk up to this LSN	LM/–	—
`nextUndoLSN`	log rec.	Undo which log record next	LR/LR	`prevLSN`, `UndoNxtLSN`
`nextL1UndoLSN`	transact.	Undo which L1 operation next	TC/L	—
`oldestUndoLSN`	system	Smallest `transactionLSN`	TM/L	—
`pageLSN`	page	Last update operation on page	PC/PC	`pageLSN`
`redoLSN`	page	First update operation after page write	PI/L	`RecLSN`
`systemRedoLSN`	system	Smallest `redoLSN` in system	BM/L	`RedoLSN`
`transactionLSN`	transact.	First log record written by transaction	TC/L	—
`undoLSN`	transact.	Last written record with undo info During rollback: Next log record to undo	TC/L	`UndoNxtLSN`

Transient/Persistent LSN Storage Locations

BM	buffer manager	LR	log record	TC	transaction control block
L	log	PC	page contents	TM	transaction manager
LA	log anchor record	PI	page interpreter	V	varies by object type
LM	log manager	RC	record contents		

Table 7.1: Special LSNs

Name	Explained in Section(s)
`anchorLSN`	7.4.2
`annihilatorLSN`	7.7
`beginLSN`	7.4.2
`deletionLSN`	7.5.4
`flushLSN`	7.4.2
`nextUndoLSN`	7.2.2 (as `PrevLSN`,`UndoNxtLSN`),7.4.1, 7.4.2,7.4.7, 7.4.8
`nextL1UndoLSN`	7.4.7,7.4.8
`oldestUndoLSN`	7.4.7
`pageLSN`	7.2.2, 7.4.5
`redoLSN`	7.2.2 (as `RecLSN`), 7.4.5
`systemRedoLSN`	7.2.2 (as `RedoLSN`), 7.4.6
`transactionLSN`	7.4.7
`undoLSN`	7.2.2 (as `UndoNxtLSN`),7.4.7

Table 7.2: Special LSN Index

```
enum REC_LogRecordKind;
class REC_LogRecord {
  protected:

    enum LogRecordFlags ;

  public:
    TransID transactionID() const;
    void transactionID(TransID id);

    REC_LogRecordKind kind() const;
    void kind(REC_LogRecordKind lrk);

    uint8 operationKind() const;
    void operationKind(uint8 ok);

    void clearFlags();
    bool isUndo() const;
    void setUndo(bool undo);
    bool isRedo() const;
    void setRedo(bool redo);

    LSN nextUndoLSN() const;
    void nextUndoLSN(LSN u);
  };
```

Figure 7.5: Log record base class

Base Log Record

Natix log records consist of the following required fields at minimum.

`transactionID()` identifies the transaction that logged the update. To save space, this ID is just an index into the transaction table of all active transactions. The full, unique `transactionLSN` (refer to Section 7.4.7) is not necessary here, as the index together with the current log position is sufficient to uniquely identify a transaction during normal and restart processing.

`kind()` identifies the general kind of log record and determines the general layout of the information in the log record (see below). By *general*, we mean ”necessary for the *recovery manager* to process the log record”. As the recovery manager often forwards log records to other objects for further processing, we can hide some of the log record internals to increase loose coupling.

`operationKind()` specifies further log record type and layout information, which is not necessary for the recovery manager, but only for the ”end user” of the log record (see below).

Flags describe whether the record contains redo-only, redo-undo, or undo-only information.

```
class REC_LogRecordSegmentUpdate : public REC_LogRecord {
public:
  void segmentID(const SegmentID &id);
  const SegmentID &segmentID() const;
};
class REC_LogRecordPageUpdate : public REC_LogRecordSegmentUpdate {
public:
  PID targetPID() const;
  void targetPID(const PID &pid);
};
class REC_LogRecordFSIUpdate : public REC_LogRecord {
public:
  PageNo start() const;
  void start(PageNo p);
  PageNo size() const;
  void size(PageNo p);
  uint8 value() const;
  void value(uint8);
};
class REC_LogRecordBufferChange : public REC_LogRecord {
public:
  void pageID(const PID &id);
  const PID &pageID() const;
};
class REC_LogRecordActiveTransaction : public REC_LogRecord {
public:
  LSN transactionLSN() const;
  void transactionLSN(const LSN &);
  LSN undoLSN() const;
  void undoLSN(const LSN &);
};
```

Figure 7.6: Some log record classes

`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 of the same transaction which contains undo information.

Only in case of compensation log records (CLRs, Section 7.2.2), the `nextUndoLSN` points to the operation logged before the operation this log record is an undo log record for. In Section 7.7, we show another situation where the `nextUndoLSN` chain of log records is not just the reverse sequence of log records of one transaction.

Note that compared to other ARIES implementations [66, 36, 94], Natix log records are rather compact, as the LSN of log records is not stored in the records themselves, and only one log record backward chain (`nextUndoLSN()`) is maintained instead of two or more.

Log Record Kinds

From the base log record class, special log record classes are derived. They encapsulate the actual storage formats of the data. Some of them are shown in Figure 7.6, without the details of their layout. Mapping between the log record kinds and respective classes used to interpret the layout is illustrated in Table 7.3. This table also describes whether the log record occurs in ARIES or not.

Log Record Kind	Log Record Class (also depends on operation kind)	Explained in Section	Introduced in ARIES
`SEGMENT_UPDATE`	`REC_LogRecordSegmentUpdate` (or subclass)	7.4.4	–
`PAGE_UPDATE`	`REC_LogRecordPageUpdate` (or subclass)	7.4.5	√
`FSIPAGE_UPDATE`	`REC_LogRecordFSIUpdate`	7.5.5	√
`PAGE_FORMAT`	`REC_LogRecordPageUpdate`	7.5.6	√
`COMMIT_TRANSACTION`	`REC_LogRecord`	7.4.7	√
`ABORT_TRANSACTION`	`REC_LogRecord`	7.4.7	√
`BEGIN_CHECKPOINT`	`REC_LogRecord`	7.4.8	√
`BUFFERMGR`	`REC_LogRecordDirtyPages`	7.4.6	√
	or `REC_LogRecordBufferChange`		–
`TRANSACTIONMGR`	`REC_LogRecordActiveTransaction`	7.4.7	√
`PARTITIONMGR`	`REC_LogRecordMountedPartition`	7.5.1	(√)
`END_CHECKPOINT`	`REC_LogRecord`	7.4.8	√

Table 7.3: Log record types and the associated classes

The log record's `kind()` and `operationKind()` values (see above) determine which class is used, as we do not want to store C++ dynamic type information [88]. Only C++ `static_cast<>` operators are used to obtain correctly typed pointers to log records.

The most frequently created log records are page-level log records (`PAGE_UPDATE`). The individual page interpreters (Section 7.4.5) each have special log record classes to describe their updates, which are subclasses of `REC_LogRecordPageUpdate`. The appropriate class is a function of the segment class and the `operationKind()` of the log record.

In some cases, segments need to write log records, too, to describe logical inverses of highly concurrent L1 operations. Such log records of the kind `SEGMENT_UPDATE` also have segment-specific subclasses to access their contents, again depending on the log record's `operationKind()`.

The buffer manager generates two `operationKind()`s of log records. During a checkpoint, it writes the complete dirty page map to disk, and during forward processing, it logs when pages are removed from the dirty page map by either a flush or by dropping an invalid page from the buffer without writing it (Section 5.6.5). The dirty page map modification log records do not use the `REC_LogRecordPageUpdate` class as they do not need to store a `segmentID`.

The records generated by the buffer, transaction and partition managers during checkpoints are detailed in the sections specified in the table.

7.4.2 Log Manager

Nearly all components of the recovery subsystem are dependent on the log manager, either because they read log records, or write log records, or both.

Buffering the Log

Log records are accessed in main memory for reading and writing and have to be transferred to and from secondary memory in the form of partition objects. While the regular buffer manager appears to cover similar functionality, log access behaviour differs from that of regular data, and requirements for the regular buffer manager and the log buffer manager are very different (Subsection 7.4.3).

Hence, the log is buffered using a special log buffer manager, which is a part of the log manager. Further details on requirements, design and implementation of the log buffer can be found in Subsection 7.4.3.

Flushing the Log

Synchroneous writing of the log when replacing pages or when committing transactions is important to the correctness of ARIES. The log manager interface must support log flushing up to a specified LSN.

There is a certain degree of freedom concerning when such a synchroneous call may return to the caller. The last page that has to be flushed may not be completely filled with log records. If the page is flushed immediately, then either no new log records may be inserted on the same page, or it must be written again, causing several synchroneous I/Os where only one would be required.

As a remedy, the log manager may delay the log flush operation and block the thread requesting the log flush. Other transactions may concurrently add records to the partially filled page. When it is full, the waiting thread is unblocked and the page is flushed. Even longer delays can include more consecutive pages in a single write request, amortizing the latency cost for I/O processing.

Transaction commits are the main reason for log flushes. The technique of delaying log writes is therefore called *group commit* [31]. Log flushes due to Write-Ahead-Logging are less frequent. By the time a page is replaced and has to be written to disk, associated log records usually have been flushed as the updating transaction has already committed.

A group commit delay can be dynamically adapted to yield optimal throughput in an environment with varying workloads [41]. This is not yet supported by Natix. However, if only few transactions operate concurrently, or if a single transaction has special response time requirements, group commit delays may be inacceptable. Hence, the interface has to allow transaction-specific log flush behaviour.

Data Copy vs. Fix/Unfix

Creating a log record in Natix is done by two separate calls to the log manager, between which the caller is supposed to fill in the log record header and data directly in the buffer location. To read a log record, the caller must use two calls to announce begin and end of a read access, between which the caller may access the log record directly in the log buffer.

This minimizes the time spent in the log manager monitor (see Synchronization below), and allows for multiple transactions to create and read log records in parallel. It also allows to avoid data copy operations during read and write because the caller may access log records directly in their buffer location. A disadvantage of Natix's approach is that log records may not span several disk pages. However, updates are logged on the page level. Hence, no more than one page of data can be modified by a single operation, and update log records typically do not get larger than a log page if the log page size is larger or equal to the data page size. It is possible that before and after images require a combined log record larger than the page size, in which case it is possible to write undo and redo information into separate log records.

An alternative approach (described for example in Gray et al. [36]) is to require the log record contents to be provided as arguments to a single log record creation call, allowing log records to be larger than a page. This requires at least one additional copy operation to move the log record contents from the caller's buffer to the log manager's buffer. Probably, additional copy operations are required to assemble the log record in the caller's buffer. The authors argue that an interface which allows direct access to the log header is not modular and dangerous in the presence of untrusted callers.

Apart from encapsulation, Gray and Reuter consider the necessity for a dedicated protocol with fix and unfix operations for the log a disadvantage, but without explaining why.

Their arguments are not strong. Apart from the fact that we are not designing a general purpose transaction manager, but a specialized DBMS for XML which is unlikely to integrate untrusted callers, the fix/unfix interface used by Natix is in fact the more general approach. It hides the actual implementation, which could as well be one that internally isolates the caller from the log and copies or transforms the log records before transferring them to the actual log. The absence of a fix/unfix protocol would also require an extra copy operation for read operations, causing a severe performance hit for transaction and restart recovery.

Natix also makes it possible to assemble a log record in several increments, for example to write the undo information first, then performing an operation, and then writing the redo information. Gray and Reuter argue that such an approach will confuse restart processing because restart may see incomplete log records. We prohibit this by not writing any log page to disk before all log records on it are completed. Since we do not allow log records larger than a page, we do not run into log buffer overflows when doing this, as long as there is space for at least one log page per logging transaction in the buffer.

Log Manager Interface

Figure 7.7 shows part of the public interface of the log manager class. We will now explain the methods grouped by purpose.

```
class LogManager {
public:
  void startLogRecord(TaCB *tacb,
                      uint16 size,
                      bool withundo,
                      REC_LogRecord*& dest,
                      LogHandle &handle
                     );
  void startSubcommitLogRecord(TaCB *tacb,
                      uint16 size,
                      bool withundo,
      LSN nextundolsn,
                      REC_LogRecord*& dest,
                      LogHandle &handle
                     );
  LSN       log(const LogHandle &lh);

  void flushLog(TaCB *,LSN flushlsn);
  void flushLogComplete();
  LSN flushLSN() const;

  class iterator {
  public:
    iterator &operator ++ ();
    void skipToLSN(LSN undolsn);

    REC_LogRecord &operator *() const;
    uint16 logRecordSize() const;
    LSN lsn() const;
  };

  iterator find(LSN aLSN);
  iterator end() const;

  const PID& beginPID() const;
  LSN beginLSN() const;
  LSN anchorLSN() const;
  void anchorLSN(LSN anch);
  void writeAnchor();
  void readAnchor();

};
```

Figure 7.7: Log manager interface excerpt

Creating Log Records

`startLogRecord()` This method allocates space in the log for a new log record of the specified transaction. The log record's size, given as parameter, may not exceed the log page size. The `withundo` flag determines whether the log record will participate in the transaction's `nextUndoLSN` chain.

The log manager returns the log record's address in the log buffer (`dest`) and a handle that allows the caller to notify the log manager once writing the record has been completed.

The log record will not be initialized, except for its nextUndoLSN, which will be filled in with the transaction control block's `undoLSN` value. The `undoLSN` value is set to the new log record's LSN.

`startSubcommitLogRecord()` The method performs the same function as the previous method, but allows for the caller to provide a custom LSN to be used as `nextUndoLSN`. This is necessary for L1 subcommit log records, see Section 7.4.4.

`log()` By using this method the creator of a log record may notify the log manager that the log record has been completed and may be written to disk. It returns the LSN of the newly created log record.

Flushing the Log

`flushLog()` flushes the log up to a specified LSN. The call may be delayed, for example to wait for the last page to be filled completely to avoid multiple writing of the same page, or to wait until a larger amount of log pages can be written with one I/O request. The transaction control block is specified to control the delay behaviour on a per-transaction basis.

`flushLogComplete()` flushes all log pages currently in the buffer. This call is used when the system shuts down.

`flushLSN()` returns the highest LSN which is known to be on disk. The flushLSN is stored because the log buffer manager only knows up to which page the log has been written to disk, but not which individual log record was stored. When `flushLog()` is used and group commit delays are not performed, the log manager may have to write the same page several times, depending on which LSN is on disk already.

Reading Log Records

`class iterator` Read access to log records is performed using an iterator interface based on the STL iterator concept [47].

Iterators reference log records, and as long as an iterator for a log record exists, it may be accessed in the location returned by `operator*()`. Fixing and unfixing the log page in the log buffer is handled by the iterator. Additional access functions of iterators return the size and LSN of the current log record.

Backward scans through the log use the `skipToLSN()` functions, while forward scans use `operator++()`, which advances the iterator to the log record with the next highest LSN value.

`find()` returns an iterator for a given LSN.

Maintaining the Log Anchor

`beginPID()` is used to obtain the first log page containing relevant log record.

`beginLSN()` returns the LSN of the first record on the `beginPID` page. This value is necessary to map LSNs to PIDs and vice versa. For details, see Section 7.4.2.

`anchorLSN()` The `anchorLSN()` functions are used by the recovery manager to determine or announce the LSN of the last complete checkpoint taken by the system. This value is made persistent by a call to `writeAnchor()` (see below).

`writeAnchor()` is used to make the current log anchor information durable in the current log partitions master segment.

`readAnchor()` is called by the recovery manager to read the log anchor values from disk.

Synchronization

The log manager is implemented as a *monitor* [44], which means that only one thread can execute any of the log manager's methods at any given time[4].

The interface approach for accessing log records described above still allows for several transactions to create log records in parallel, with several other transactions reading log records concurrently, even from the same log pages.

Note that access to the logging-related fields (`undoLSN`, `transactionLSN`) in the transaction control block is synchronized since accesses always originate from inside the monitor.

Since log records are never modified, and each transaction is only interested in its own log records, there can be no conflicting accesses between transactions, apart from allocating space for new log records, which is done sequentially in the startLogRecord monitor methods.

During regular operation, the only reason why transactions need to read log records is rollback. The transaction only follows its own `nextUndoLSN` chain, there is only one thread executing on behalf of the transaction, and log records are never moved or modified. Hence, there are no conflicting accesses here either.

[4]group commit is the only exception. With active group commit, log flushes may be delayed and other threads may use the log manager until the log flush occurs.

The above is only true on a logical level. In the implementation world, the log records must be accessible in main memory. Since only part of the log is buffered in main memory, replacement of pages in the log buffer must be synchronized between threads. Again, the log buffer's methods are only called from inside the log manager monitor, and no conflicts can occur.

Physical Organization of the Log

Natix's log is stored using regular partitions (Section 5.2). However, metadata organization of log partitions differs from regular partitions (compare Section 5.6):

There is only one single-page master segment. The rest of the partition consists exclusively of log pages without further fragmentation into segments. The descriptor of the master segment contains type information indicating that this is a log partition which does not have free space or user data segments.

The master segment also contains a *log anchor record* with log metainformation such as the position of the last checkpoint (`anchorLSN`) and the current log begin in form of `beginPID` and `beginLSN`. How these values are used, and how the end of the log can be found is described below.

The log pages themselves are simple slotted pages, which are optimized for append-only access, so that the slot descriptor only contains position and length of the record, but no status information. There are also no free space data structures on the log pages.

The available log space is used in a circular fashion. Only a small part of the accessible log pages hold *active log records*, i.e. those that are relevant for redo or undo. The window of active log pages is bounded by the `beginPID` page and the last log page. New log records are always inserted in the last log page, which is advanced to the next page if it becomes full. Due to the fix/unfix interface to access log records, several log pages may concurrently be written to because the last log page may be advanced even if not all log record writes are completed.

The `beginPID` is advanced if the available log space is too low, and is then set to the log page with the smallest LSN that is necessary for undo or redo. This process is called *log truncation* (see below).

If the last available log page is used up, logging wraps around back to the first available log page. By performing regular log truncation, it is ensure that the log never ”bites its own tail” and there is always enough space to store potential compensation log records of all active transactions. If there is a danger of log overflow, the system is stopped by aborting all transactions, and more log space has to be allocated by the administrator.

To determine the end of the log (the last page which contains log records), on each page the LSN of the first log record on that page is stored. While scanning the log during analysis or redo, the log manager can then determine the end of the log by detecting the first page with an LSN that is not the successor of the previous page's LSN. To make the system robust against garbage in newly allocated files, all pages are formatted with a Null LSN when a log partition is created.

LSN Mapping

Natix's LSNs are closely related to the log record's physical position to allow for a quick mapping. A Natix LSN consists of a pair $(pageNo, slotNo)$ where pageNo describes a logical page number starting from 1 when the log is first initialized, and slotNo describes a slot on that page. The pageNo is increased every time a new log page is used, even if the log wraps around and reuses lower PIDs. Lexicographic ordering on the LSN tuple provides the required ordering of LSNs.

The log anchor contains a `beginPID` and a beginLSN, describing the first physical page ID of the current log partition, and the associated LSN. Mapping between PIDs and LSNs is then performed by using the beginLSN/PID as base value, and adding the difference of the current page/LSN and the `beginPID` and LSN values. Some extra arithmetic is required to take care of multiple log partitions and log wraparound, but mapping between LSNs and PIDs is still very fast, taking only constant time, no matter how large the log is.

In order to save space, Natix does not store the LSN inside the log records. Only the LSN of the first log record on each page is stored in the page header.

Log Truncation

The system must always be able to perform restart recovery. In the worst case of a system crash where all transactions were uncommitted at the instant of the crash, this means there must be enough log space to roll back all currently active transactions.

Natix uses a generous estimate of the required log space by reserving at least twice the amount of log written by all active transactions.

Calculation of the required amount is based on the difference between the current `beginPID` and the last log page.

The process of advancing the `beginPID` is called *Log truncation*. It is invoked regularly, at least every time a checkpoint is taken.

The `beginPID` must point to the first page that contains relevant log records. Due to committing transactions, flushed pages and recent checkpoints, the `beginPID` may point to a location too far back in the log.

The currently required `beginPID` value is the smallest LSN of a log record required by restart. If a mere recalculation of the currently required window of active log records does not sufficiently advance the `beginPID`, then stale pages are flushed until the window of active log records is again small enough. However, this only advances the `beginPID` if the `systemRedoLSN` is smaller than the `anchorLSN`. The `anchorLSN` is only advanced when a checkpoint takes place.

We therefore perform log truncation every time a checkpoint is taken (see `checkpoint()`, Section 7.4.8). If this does not sufficently advance the `beginPID`, we issue a warning message, indicating that log space is low. In this case the administrators need to increase the checkpoint frequency or the available log space.

If the available log space is still lower than twice the amount of log written by all active transactions, the system is shut down. If this is considered too harsh, it is possible to take a softer approach by introducing a sequence of thresholds. Passing each threshold would invoke progressively more effective measures to reclaim log space. Such measures include, in order of increasing impact, (1) disallow new transactions to begin (2) suspend running transactions (3) abort the n oldest transactions, (4) shut the system down. But such a proliferation of thresholds and effects make the systems behaviour more complex and difficult to understand, increasing administration effort. Natix only has one threshold and shuts down the system. We wanted to keep administration effort low, and the long-term solution for problems with log space is to reconfigure the system anyway, for example by adding more log space or allowing fewer concurrent transactions.

The following steps are performed to implement a log truncation:

1. Acquire log manager monitor mutex.

2. If already in log truncation mode, return immediately.

3. Go into log truncation mode by setting a flag. This avoids several threads to invoke log truncation simulataneously and avoids warning messages if log truncation itself tries to write log records.

4. Determine current `anchorLSN` (LSN of last checkpoint).

5. Determine current `beginLSN`.

6. Drop log manager monitor mutex because log truncation itself may take some time and requires writing of page flush log records.

7. Determine `oldestUndoLSN` by calling transaction manager.

8. Determine `systemRedoLSN` by calling buffer manager.

9. Determine minimum of `anchorLSN`, `oldestUndoLSN`, `systemRedoLSN`, and use as prospective new `beginLSN`.

10. If new `beginLSN` is sufficiently larger than old `beginLSN` or if it is not equal to `systemRedoLSN`, skip to 13.

11. Flush the oldest page in the buffer manager.

12. Repeat from 8.

13. Reacquire log manager monitor lock.

14. Determine `beginPID` for new `beginLSN`.

15. Store Log Anchor Record (This can be left out if called from checkpoint, as the checkpoint will write the log anchor anyway).

16. If the `beginPID` could not be advanced sufficiently, issue proper error messages and perform system shutdowm while still in log truncation mode.

    The log space will still be sufficient to write compensation log records for all active transactions because log truncation fails immediately once the minimum threshold is reached, and the threshold is sufficently generous.

17. Go back to normal mode.

18. Drop log manager monitor mutex.

Careful Writes

Sometimes, when log pages are flushed, the same log page may be written several times because the `flushLog()` function flushed a non-full page. The log anchor record is also written more than once.

The log manager must take care to guarantee that several writes of the same log-related pages do not corrupt the pages if a system crash occurs during write. Since this topic has been exhaustively treated in the literature (for example in Gray et al. [36]), we will not go into details here.

7.4.3 Log Buffer

The *log buffer* is a part of the log manager and performs the transfer of log records from memory to disk and vice versa.

Since our design choice is to implement fix/unfix interface for the log manager with direct access to the log buffer locations for the log records, we have to take the resulting requirements into account when designing the interface used to access the log buffer from the log manager.

Most database recovery literature (including for example [36, 66, 94]) does not describe a detailed protocol how to access the log buffer. The problem is considered trivial and often reduced to suggesting some kind of ring buffer to allow appending log records and sequentially writing them. This leaves open the question how the log buffer is employed when reading log records during transaction rollback. Even when the rollback procedures use private buffers for each transaction, some of the younger log records needed for rollback may still reside in the log buffer for writing and there needs to be a protocol to access them without conflicting with log writes that may happen simultaneously.

As a solution Gray et al. [36] propose to use an instance of the regular buffer manager as log buffer, since it already contains the necessary protocol to synchronize multiple readers and writers. We did not follow this approach in Natix, as the log buffer and the regular buffer manager, due to properties of the recovery method and our design goals for high performance, have very different access characteristics (Table 7.4).

These differences strongly suggest using an interface and an implementation for the log buffer that are different from the regular buffer manager. The Natix design and implementation is presented in the following subsections.

Regular Buffer Manager	**Log Buffer Manager**
needs to synchronize associative access to pages	only called from log manager, which is a monitor
random access for write	only sequential writes
arbitrary modifications of pages	append only
only one writing thread per page	multiple writing threads on the same page
has to ensure write-ahead-logging for data pages	log pages do not log updates

Table 7.4: Buffer manager vs. log buffer manager

Log Buffer Organization

The log buffer is similar to the regular buffer manager in that it contains a fixed set of pages, and an associative access structure to find pages based on their PID. In contrast to the regular buffer manager, these structures are not synchronized because the log buffer is only called from the log manager, which is a monitor and hence does not allow concurrent access.

The log buffer in Natix is organized as an array of buffer frames, each having a buffer frame control block with two *fi x counters*, one for the number of readers and one for the number of writers. A frame is called *fi xed* when at least one of the read or a write fix counters is larger than zero. Otherwise, the frame is *unfi xed*. Initially, all frames have both fix counters set to 0.

The array is used as a ring buffer, where the role of *current frame*, i.e. the frame that contains the current end of the log, cycles through the buffer.

Usually, a buffer ring contains consecutive pages, and the oldest page in the buffer is contained in the *next frame in the ring* after the current frame, i.e. the frame in the array that is reached by incrementing the current frame's index by one and doing a modulo division by the buffer capacity.

However, in our case, if a rollback is reading old log pages, some of the frames in the ring may contain older, out-of order pages. When cycling the current frame through the ring, those frames are skipped if there is still a thread that has a fix on them. The currently unfixed pages are not chained, akin the LRU chain in the regular buffer manager. Instead, the frame to hold the new current page is determined by advancing one frame at a time until the next unfixed frame is found.

Log Buffer Interface

`getNewPage()` allocates a new frame in the log buffer without loading a page from disk. The frame allocated is the next unfixed frame after the current frame. The allocated frame is the new current frame afterwards. The frame's write fix counter is incremented.

```
class LogBuffer {
public:
  class LogFrameCB {
  public:
    LogPage *page() const;
    ptr_t contents() ;
    bool isFixed() const;
    bool hasPendingWrites() const;
    bool isDirty() const;
  };

  LogFrameCB *getNewPage(const PID &pid);
  LogFrameCB *getCurrentPage();
  void getPageReadOnly(const PID &pid, LogFrameCB *&frame);

  void releaseRead(LogFrameCB *frame);
  void releaseWrite(LogFrameCB *frame);
  void releaseReadInvalidate(LogFrameCB *frame);

  void flushToPID(const PID &flushpid);
  void flushToCurrent();
};
```

Figure 7.8: Log buffer interface excerpt

`getCurrentPage()` returns the current frame and increases its write fix counter.

`getPageReadOnly()` returns a frame control block for the specified page.

If the page is not in memory, the frame given as argument is replaced if that frame is not fixed. Otherwise, the next unfixed frame in the ring after the current page is replaced.

This allows for the recovery manager to reuse the frame used in an undo chain traversal. Since undo frequently reaches far back into the log, not reusing the same frame would contaminate the log buffer with a lot of old pages. In effect, this results in each transaction rollback having a private log buffer page while still sharing log buffer space if possible.

`releaseRead()` decrements the current read fix counter for the page.

`releaseWrite()` decrements the current write fix counter for the page.

`releaseReadInvalidate()` decrements the current read fix counter for the page and invalidates the associated PID for that frame.

This call is used while searching for the end of the log during analysis. If a page with out-of-order LSN in the page header is read, then the end of log has been passed. The out-of-order page is the page behind the last log page and must be removed from the buffer without writing it, as it does not contain meaningful log data.

`flushToPID()` Writes all dirty pages from the chronologically first dirty page up to the page with the PID given as argument back to disk. If the given PID is not in the buffer or is not dirty, no operation is performed.

The log buffer remembers the last dirty page that was flushed to disk, to avoid a linear search every time flushToPID is invoked. Scanning the buffer ring fowards, it writes groups of dirty log pages that are consecutive in main memory to disk using the partition's `writePages()` call. Not all of the pages are consecutive in main memory, because the buffer ring may wrap around, and because `getPageReadOnly()` may have read some old pages into the buffer.

`flushToCurrent()` Writes all dirty pages back to disk.

Log Buffer Capacity Requirements

As a result of the design outlined above, there are some requirements for the capacity of the log buffer.

During forward processing, it is possible that every transaction is concurrently writing a log record. To allow this to happen, there must be one buffer frame for every transaction because it is possible that every transaction writes a log record that consumes a whole buffer frame. It would have been possible to employ multi-thread synchronization to suspend a thread until an unfixed log frame is available for it to write its log record, but this would have complicated the interaction between the log manager and the log buffer. We do not consider a number of log frames equal to or larger than the number of allowed concurrent transactions a serious drawback. Hence, we kept the interface of the log buffer simple, assuming a sufficient number of available buffer frames.

During rollback, a transaction scans backwards through the log, reading its log records and undoing them. Undoing the log records may result in compensation log records. Hence, a transaction may simultaneously read from one and write to another log frame. This adds the number of transactions rolling back concurrently to the number of required log frames. It is possible to put an upper bound on the number of transactions rolling back concurrently by putting some kind of semaphore into the recovery manager. In contrast to the synchronization of free log frames in the previous paragraph, this would only add internal synchronization to a single class.

Currently, Natix simply requires a log buffer capacity of at least twice as many log buffer frames as allowed concurrent transactions.

7.4.4 Segments

We now describe in more detail how the segment classes perform logging and recovery. We start with the mechanisms in the segment base class and then explain how a regular data segment class can be made recoverable. A description of the framework for recovery of L1 operations concludes the section.

```
class SEG_Segment {
[...]
   // persistent constrcutor
   SEG_Segment(const Recipe &);
   // persistent destructor
   static void destroy(SEG_Segment *);

   void undoLogRecord(StmtCB *stmtcb, BufferFrameCB *lastframe,
                      LogRecord *undorecord, uint16 logrecsize);
   LSN redoLogRecord(BufferFrameCB *lastframe,
                     LogRecord *redorecord, uint16 logrecsize);
[...]
};
```

Figure 7.9: Segment base class recovery functions

Segment Base Class

For regular recovery, the segment base class in Natix offers two virtual functions, one for redo and one for undo. They are called during processing of log records that belong to the segment.

The default implementation of these functions demands the appropriate pages from the buffer manager and then forwards the log records to the associated page interpreter.

The following is a description of the methods shown in Figure 7.9.

`undoLogRecord()` In addition to the log record, the `undoLogRecord()` function receives the frame control block for the last processed page and a statement control block for the undo operation. Only if the frame contains a different page than the target page of $undorecord$, the segment needs to unfix $lastframe$ and fetch $undorecord$'s target page. This minimizes the calls to the buffer manager if a series of subsequent log records apply to the same page, which is likely especially if our clustering XML storage format is used.

After establishing a main memory address for the page, the log record is forwarded to the page interpreter by calling its `undoLogRecord()` function.

The virtual function `undoLogRecord()` only needs to be refined by derived classes if they perform and log segment L1 operations, as described below.

`redoLogRecord()` To redo a log record, the segment instructs the buffer manager to fetch the page, performing the same check for last used frame as the `undoLogRecord()` function above. It then checks whether the page already contains the log record's update by inspecting the `pageLSN`. If the page does not contain the update, the log record is forwarded to the page interpreter by calling `redoLogRecord()` on the associated page interpreter (Section 7.4.5).

The current pageLSN is returned to the recovery manager to update the dirty page map. In case the current log record's update was already contained in the page, this allows for the recovery manager to avoid attempts to redo log records until the redo log scan reaches the proper `redoLSN` for this page.

The virtual function redoLogRecord only needs to be refined if the segment writes segment-level redo log records, which was not necessary for any encountered segment type in Natix. The redoLogRecord function is virtual only for reasons of symmetry and completeness.

Persistent Constructor, Persistent Destructor, grow() To enable metadata recovery for segments, the segment base class also writes log records when a segment is created on disk, destroyed, or when it grows. Treatment of these log records is described in Section 7.5. Derived segment types do not need to change or add the metadata recovery.

Adding Simple Recovery to a Segment Type

Segments have to make sure that the recovery versions of the page interpreters are used. Since the segments use a factory class to create the page interpreters, the only code particular to recovery is initialization of the proper page interpreter factory when a recoverable segment is opened. The actual code necessary to enable recovery for a segment type is limited to about 5 lines in the segment constructor.

Adding Recovery to a High-Concurrency Segment Type

The following paragraphs describe a framework provided by Natix's recovery subsystem, which allows for relatively simple integration of object-level recovery techniques as explained in Section 7.2 to enable recovery for segments in which the same object may be concurrently updated by several transactions. Operations on such objects, which cannot be undone by applying the inverses of the original page updates, are called *L1 operations*.

In Natix we distinguish between two classes of L1 operations, *segment L1 operations* and *metadata L1 operations*.

Segment L1 operations are necessary for regular segment types that support a high degree of concurrency (e.g. B-Trees). Although the inverse of the operation on the segment might not be correctly described by the page-level inverse operations, the inverse of the L1 operation only affects pages of the original segment. Below, we describe the framework which can be used to easily provide recovery for this kind of L1 operations.

Metadata L1 operations result from the fact that even operations that at first glance only affect single pages sometimes need metadata maintenance, thus affecting more than one page. Examples include free space inventory updates, where in addition to the data page, a free space inventory page is updated; or segment growth, where not only the segment metadata, but also the partition-level free extent table needs modification. Since we definitely want high concurrency on the metadata structures, operations on them are L1 operations.

As in the examples just mentioned, metadata maintenance and recovery often affect not only several pages, but several segments as well, and the interactions are quite delicate and performance-sensitive. Hence, we do not provide a general framework for metadata L1 recovery. While some of the recovery logic for metadata L1 operations can be implemented using segment L1 operations for metadata segment types, a certain amount of special code is still necessary. Further details on recovery of metadata L1 operations can be found in 7.5.

At this point, it is sufficient to say that metadata recovery is handled by the segment base class. Apart from the segments for metadata implementation, none of Natix's segment types needs special metadata handling code.

Forward Processing The segment executes the L1 operation using its regular update code, generating page-level log records in the process. The only difference is that the execution of the L1 operation is surrounded by calls to the transaction manager indicating begin and end of the L1 operation.

Figure 7.10 shows a code sample for a B-Tree insert call. For simplicity, we assume that key-value pairs are given as memory address and size arguments, and the BTree segment knows how to deal with the contents, including how to extract the key, and the value, and how to compare keys.

Before executing an L1 operation, the segment calls `initiateL1()` on the transaction manager to mark the begin of an L1 operation. The `initiateL1()` call returns an LSN that marks the state of the transaction before the L1 operation.

After performing the L1 operation using its regular code (in the example, this happens inside `regularInsertKV()`), the transaction manager is notified by calling `notifyCompleteL1()` (see Section 7.4.7). Then the segment writes a subcommit log record containing the undo information necessary for the segment to logically undo the operation. The segment instructs the log manager to use as `nextUndoLSN` for this record the LSN returned by the `initiateL1()` call.

The regular undo chaining performed by the log manager will not be suitable for this record, as the default behaviour is to set the `nextUndoLSN` field to the last log record written by the transaction, in this case one of the log records that describe the L1 operation.

The correct record to undo after the subcommit record was undone is the one that was written *before* the first log record of the L1 operation. This is the LSN returned from `intiateL1()`, and since the log manager allows custom undo LSN chaining, we can use it as `nextUndoLSN` when writing the L1 subcommit log record.

Undo Processing During undo processing, the recovery manager calls the segment `undoLogRecord()` virtual function for every log record with a matching segmentID.

The segment base class implementation of the undoLogRecord function will forward the log records on to the appropriate page interpreter. For a segment that logs L1 operations, we have to refine the undoLogRecord function to deal with L1 undo. The refined function processes regular page-level log records by calling the base class implementation function as usual.

```
void SEG_BTreeSegment::insertKV(StmtCB *cb,ptr_t keyvaluepair, uint16 keyvaluesize) {
  TRN_TransactionManager *trnmgr=cb->getTransactionManager();

  LSN beforelsn=trnmgr->initiateL1(cb->getTaCB());
  regularInsertKV(cb,keyvaluepair);
  trnmgr->notifyCompleteL1(cb->getTaCB(), beforelsn);

  logInsert(cb, keyvaluepair, beforelsn);
}

void SEG_BTreeSegment::logInsert(StmtCB *cb,
                                 ptr_t keyvaluepair,
                                 uint16 keyvaluesize,
                                 LSN beforelsn) {
    TaCB *tacb=cb->getTaCB();
    REC_LogManager *logmgr=stmtcb->getLogManager();
    REC_LogRecord  *lr;

    uint16 logrecsize=sizeof(BTreeInsertionLogRecord)+
                      keyvaluesize);
    bool undoable=!tacb->isL1UndoInProgress();

    // get log record location
    REC_LogManager::LogHandle logrec;
    logmgr->startSubcommitLogRecord( tacb,
                                     logrecsize,
                                     undoable,
                                     beforelsn,
                                     lr,
                                     logrec);

    BTreeInsertLogRecord  *blr=
           static_cast<BTreeInsertLogRecord*>(lr);
    // fill in log record header fields
    blr->kind(REC_LRK_SEGMENT_UPDATE);
    blr->segmentID(segment()->id());
    blr->operationKind(SEG_OPER_BTREE_INSERTRECORD);

    memcpy(blr->contents(),keyvaluepair,keyvaluesize);

    // finish log record creation and update LSN fields
    logmgr->log(logrec);

}
```

Figure 7.10: Performing L1 operations

```
void SEG_BTreeSegment::undoLogRecord(StmtCB *stmtcb,
                                     BufferFrameCB *lastframe,
                                     LogRecord *undorecord,
                                     uint16 logrecsize) {
  if(undorecord->kind() == REC_LRK_SEGMENT_UPDATE) {
    switch(undorecord->operationKind()) {
      case SEG_OPER_BTREE_DELETE: {
          BTreeDeleteLogRecord  *blr=
             static_cast<BTreeDeleteLogRecord*>(lr);

          insertKV(stmtcb, blr->contents(), blr->keyValueSize());

          return;
        }

[... other L1 operations for B-Trees...]

      default:
        break;
    }
  }

  // process non-L1 log records by invoking base class
  SEG_Segment::undoLogRecord(stmtcb,lastframe,undorecord,logrecsize);
}
```

Figure 7.11: Undoing L1 operations

On encountering L1 subcommit records, however, the function must call the appropriate segment-level inverse operation of the segment, using the undo information supplied in the subcommit log record.

The regular segment-level inverse operation usually is a regular function of the segment's data structure interface. For example, in a B-Tree segment, a subcommit log record generate by `deleteKV()` (delete key-value pair) will be undone by calling the segment-level inverse `insertKV()` operation on the segment (Figure 7.11).

Using the regular high-level functions for undo will automatically cause the logical undo operation to be an L1 operation itself, as required by the object-recovery protocol outlined in Section 7.2.

Execution of the inverse L1 operation is a subtransaction which again has special requirements with respect to `nextUndoLSN` chaining. However, the segment announces begin and end of the inverse L1 operations as during forward processing. In Figure 7.11, this occurs for example in the call to `insertKV()` (refer again to Figure 7.10). When `initiateL1()` and `notifyCompleteL1()` are called from `insertKV()`, the recovery and transaction managers know that high-level undo is taking place. They can ensure that the transaction control block is set up in a way that causes correct `nextUndoLSN` chaining for the inverse L1 operation (refer to sections 7.4.7 and 7.4.8 for details).

It is not necessary for the segment code to call special segment-level undo functions for L1 undo. The only difference is that when writing L1 subcommit records, the segments need not include logical undo information if L1 undo is in progress, which is indicated by the `isL1UndoInProgress` flag.

Redo Processing Nothing needs to be changed for redo processing in segments with L1 operations. The L1 subcommit log record may be ignored because the necessary steps to redo an L1 operation are described by the page-level log records written during the execution of the operation.

Analysis and Checkpointing There is nothing that a high-concurrency segment needs to change for redo processing, analysis or checkpointing.

More Concurrency

The framework described above allows for transactions to concurrently modify multi-page data structures in a recoverable way, while keeping the code necessary for recovery separated from the actual modification of the data structure.

However, the fact that every undo operation can no longer be performed on the page-level, but has to be undone on the logical level, imposes limits on concurrency during undo. For example, in a B-Tree, undoing an insert operation using the above framework will cause a new search for the key to be deleted, including locking for the upper levels of the tree. This may prove too expensive for some applications, especially since operations that modify the structure of the tree must hold exclusive locks also on higher tree levels, even if these locks are not held until the transaction commits.

Further concurrency can be achieved by implementing specialized methods like ARIES/IM [67] for B-Trees, or ARIES/LHS [61] for persistent hash tables. Their approach is to annotate the data structure itself with information about ongoing concurrent structural modification. This requires recovery-specific interfaces for page interpreters, and segment-level code where recovery logic and data structure modification are not separated. Undo processing does not use the regular forward processing code. The simple code structure outlined above, where data structure modifying code is simply parenthesized by recovery calls, is no longer possible.

Still, implementation of such methods is simplified by our framework and can make use of the provided infrastructure methods to maintain log records and transaction control blocks. Also, note that Natix's recovery manager is not affected by methods like ARIES/IM. To implement them, it is sufficient to define and refine methods of the segment and page interpreters, keeping the overhead and complexity localized.

7.4.5 Page Interpreters

The page interpreters form the largest component of the recovery subsystem, containing about half of the total code necessary for recovery.

The following subsections elaborate on the design and implementation issues related to recoverable page interpreters. We first explain the basic design pattern used to integrate the recoverable page interpreters. The required modifications to the page interpreter base class are discussed next, followed by a list of requirements for the interfaces of data types that are to be made recoverable. Finally, some implementation issues for concrete data types are reviewed.

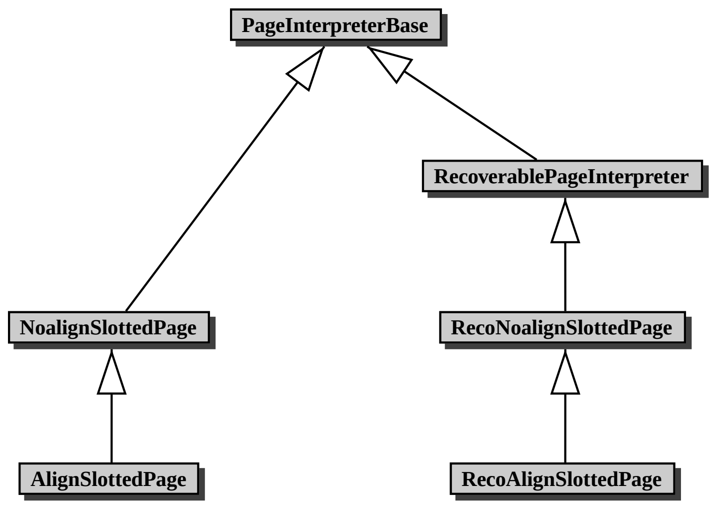

Figure 7.12: Two parallel page interpreter hierarchies

Page Interpreter Class Hierarchy Design

The design for integrating recoverable page interpreters takes an object-oriented approach. By defining an interface for a given data type and deriving different versions of page interpreters for that interface, it is possible to encapsulate recovery in the page interpreters. The page interpreters' callers, the segments, do not interact differently with recoverable and non-recoverable pages. The only difference is that, as a side effect, recoverable page interpreters write log records during forward processing.

Recoverable page interpreters must implement two interfaces, however. The first is the interface of the stored data type, derived from the common page interpreter base class, and the second is the interface for common page-level recovery functionality, such as maintenance of `pageLSN` and `redoLSN`, and write-ahead logging.

We have to select one of the interfaces as the root of our hierarchy. Multiple inheritance is no solution, not only because compiler incompatibilities and problems often surface with uncommonly used features as multiple inheritance. The main reason we do not want to use multiple inheritance is that we would need at least two page interpreter pointers in the frame control block, one to the virtual function table implementing the regular page interpreter base class, and one for the virtual function table implementing the recovery functionality.

The design alternatives for the page interpreter hierarchy are as follows:

two parallel hierarchies (Figure 7.12). We can have a common base class for the recoverable page interpreters, with an attached inheritance tree that mirrors the regular page interpreter tree.

abstract data type interface class (Figure 7.13). There is an explicit abstract base class for each data type, from which both the regular and the recoverable version are derived.

derive recoverable from regular versions (Figure 7.14). We use the regular classes as base classes for the recoverable versions.

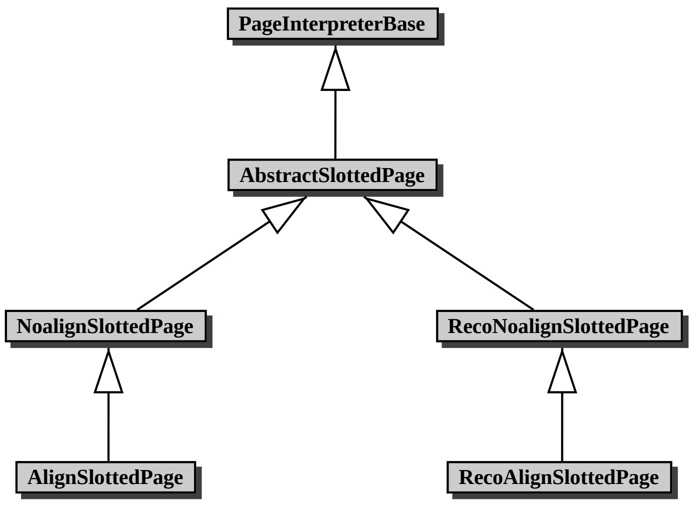

Figure 7.13: Abstract data type interface

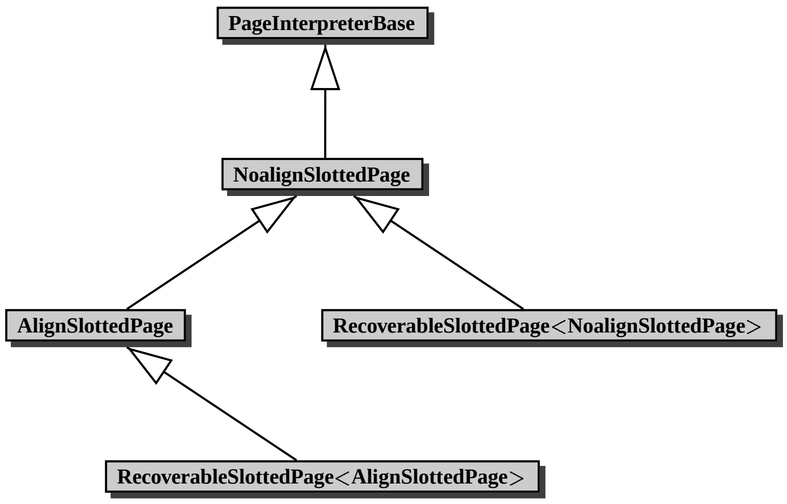
PageInterpreterBase
NoalignSlottedPage
AlignSlottedPage
RecoverableSlottedPage<NoalignSlottedPage>
RecoverableSlottedPage<AlignSlottedPage>

Figure 7.14: Derive recoverable from regular

The first alternative introduces a common base class for all recoverable pages. This factors functionality common for all recoverable pages (like `redoLSN` storage). While this is a nice property, this class hierarchy design is inappropriate because the recoverable and nonrecoverable versions cannot be used from the segments in a uniform way because the common base class for them does not contain the type-specific interface used to access the page's contents. As a consequence, only the last two approaches are viable.

In Natix the last alternative was chosen over the second because the recovery functionality is an addition to the regular functionality, and not an alternative – the original functionality, storing the data, is still required. The simplest way to integrate it is to inherit from the original class.

A way to implement a hierachy in the style proposed by the second alternative would be to factor the original functionality into its own class, for example called `StorageLayout`. This class would then be used as part of both the recoverable and nonrecoverable page interpreter variants. However, this aggegration-based approach would result in a lot of *delegation*, where the page interpreter classes have lots of redirecting one-line function calls to `StorageLayout`. In turn, `StorageLayout` would need a back pointer to the page interpreter to access the core page interpreter infrastructure. In short, breaking the tight coupling between the page interpreter object and the implemented data type would make the code unnecessarily large and harder to understand and maintain.

That is why the third design alternative, deriving the recoverable from the regular page interpreters, was chosen in Natix.

For data types that have several alternative page interpreter implementations even without recovery, this requires to derive a recovery version from each of the regular implementations, although the recovery functionality is identical. One example in Natix are the slotted pages, which have two implementations for the same interface. The first stores records directly next to each other, while the other aligns the records. The recovery code for both versions is identical. To avoid code duplication in such cases, the C++ template mechanism is used. The recoverable version is specified with the regular implementation as template parameter.

Page Interpreter Base Class

As a result of the design pattern for our page interpreter class hierarchy, functionality common to all recoverable classes has to be located in the page interpreter base class (Figure 7.15), in order to allow for the recovery algorithms to treat the recoverable pages in a uniform way, and allow for dynamic linking where needed.

`pageLSN()` These functions allow to access to the associated page's `pageLSN`. While every page interpreter class may choose to implement a different storage location somewehere in the page contents, the base page interpreter provides a standard handling which uses the first 8 bytes of the page content to hold the `pageLSN`.

redoLSN The `redoLSN` is an attribute of the page interpreter class and is not stored on the page itself, containing the LSN of the first operation that modified the page since

```
class PGE_Page {
[...]
   virtual void pageLSN(LSN);
   virtual LSN pageLSN() const;
   virtual void redoLSN(LSN);
   virtual LSN redoLSN() const;
   virtual void logLSN(LSN);
   virtual void redoLogRecord(StmtCB *, LSN, LogRecord*, uint16 recordsize);
   virtual void undoLogRecord(StmtCB *, LogRecord *, uint16 recordsize);
   virtual void prepareWrite(GloCB *);
[...]
};
```

Figure 7.15: Page interpreter recovery functions

it was last written to disk. It is null for unmodified pages. It is set to null after a page has been loaded, and it is modified by the `logLSN()` function below. Flushing a page to disk also resets the `redoLSN`.

`logLSN()` This call is used by recoverable page interpreters after a log record for the page has been created. It updates the `pageLSN` with the log record's LSN. If the `redoLSN` is null, it is also set to the new LSN.

`redoLogRecord()` This is a virtual function that is called by the segment (see 7.4.4) during redo for every log record for the page whose update is not yet contained in the page. It does nothing in the default implementation.

`undoLogRecord()` This is a virtual function that is called by the segment during undo for every log record which updated the page and requires undo. It does nothing in the default implementation.

`prepareWrite()` This function is called by the buffer manager before the page is written back to disk. The existing base definition of the function as explained in Section 5.4 does not need to be changed.

Page Interpreter Interface Requirements

To augment a data type with recovery, some basic requirements must be fulfilled by the page interpreter's interface. Most data type interfaces conform to the requirements without modification, and for the others the necessary changes are usually limited.

virtual update functions We want recoverability to be transparent from the segments' perspective. This is achieved through dynamic binding, as a common interface is used for recoverable and nonrecoverable page interpreters for the same data type. Therefore, all functions used to update the data structures on the page need to be virtual. The recoverable version of a page interpreter has to refine them to write log records.

existence of the inverse To allow for proper undo, every operation on the page interpreter must have its exact inverse operation available in the interface of the class. While this seems obvious, this is usually the point where existing page interpreters have to be modified to allow for recovery.

For example, for slotted pages the inverse operation for record deletion on a slotted page is record insertion. But since the slot number is used to identify the record, we must have a record insertion routine that inserts a record into a *specifi ed slot*, while the regular `insertRecord()` function performs a search for a free slot. To allow recovery for slotted pages, an insertRecordHere function is necessary to allow undo.

Sometimes it is not desirable to export such functions as `public`, because their usage may violate invariants of the data structure if called outside the undo process. Since undo is implemented in a derived class, such function can be made `protected`.

bounded in-place updates Avoidance of data copy operations is a popular guideline to achieve optimal performance. For page interpreter update interfaces, this means that instead of requiring the caller to provide new object contents as argument to update function calls, a buffer memory location is returned where the object's contents are going to be located. Then the caller can construct or modify the object in place. Constructing it in a separate buffer and then performing an expensive copy operation is not necessary.

Since the new contents were never announced to the page interpreter, it is impossible to log the operation, as a proper after image is essential for a complete description.

We therefore require that such in-place update interfaces always include some kind of completion call which announces that construction of the new object contents is complete.

Recoverable Page Interpreters

The implementation of a recoverable page interpreter is simple, once the requirements stated above are met.

The page interpreter has to log all updates and refine the functions to interpret the log records during redo and undo. In addition, the page interpreter has to trigger write-ahead logging before a page is flushed to disk.

In the following, we illustrate the design and implementation issues by adding samples from the implementation of the `deleteRecord()` operation on slotted pages.

Logging updates Every time an update method is called, the page interpreter executes the desired update operation using the respective method of the base class. In addition, it calls the log manager to create a log record that describes the operation.

The log record contains information about the physical location of the update (segment and page ID), an operation code, stating which operation was invoked, and the logical

```
void PGE_RAL_LogSlottedPage::deleteRecord(StmtCB *stmtcb, Slot*slot) {
      // log delete operation
      logDeleteRecord(stmtcb, slot);

      // call nonlogging base class to perform operation
      PGE_RAL_SlottedPage::deleteRecord(stmtcb, slot);
    }

inline void PGE_RAL_LogSlottedPage::logDeleteRecord(StmtCB *stmtcb, Slot *slot) {
    TaCB *tacb=stmtcb->getTaCB();
    REC_LogManager *logmgr=stmtcb->getLogManager();
    REC_LogRecord  *lr;

    uint16 logrecsize=sizeof(SlottedPageUpdateLogRecord);
    bool undoable=!tacb->isL0UndoInProgress();

    // determine whether before image needed
    if(undoable)
      logrecsize+=slot->size();

    // get log record location
    REC_LogManager::LogHandle logrec;
    logmgr->startLogRecord(tacb, logrecsize, undoable, lr, logrec);

    SlottedPageUpdateLogRecord  *plr=
           static_cast<SlottedPageUpdateLogRecord*>(lr);
    // fill in log record header fields
    plr->kind(REC_LRK_PAGE_UPDATE);
    plr->targetPID(pid());
    plr->segmentID(segment()->id());
    plr->operationKind(PGE_OPER_SLOTTEDPAGE_DELETERECORD);
    plr->slotNo(slotNo(slot));
    plr->setRedo(true);
    plr->setUndo(undoable);

    if(undoable)
    { // include before image from buffer
      memcpy(plr->contents(),slot->contents(),slot->size());
    }

    // finish log record creation and update LSN fields
    logLSN(logmgr->log(logrec));
}
```

Figure 7.16: Code for logging record deletions

location within the page (e.g. slot number). For undoable operations, a before image of the object is stored in the log record, and for redoable operations, an after image is included.

The log manager returns an LSN for the log record, which is stored in the page's contents as `pageLSN`.

A code sample for record deletion on slotted pages is shown in Figure 7.16. After initializing a new log record by calling the log manager, the appropriate log record fields are filled. Some transformation from main-memory to durable formats is necessary, in this case for the slot. For performance reasons, it is addressed using its main memory address in the page's slot table. In the log record, the slot number is used instead, since the main memory address may not be the same when the log record is interpreted later.

In the example, the after image is empty, but for undoable records a before image is needed. To create it directly from the record's contents in the buffer, the operation is logged before the actual execution of the operation. This is not a problem because the page is latched and is only accessible by other callers after the operation is both logged and executed. Undoable log records are those which are not created during undo, as explained below under undo processing.

The code sample in Figure 7.16 uses the asynchronous interface to the log manager, which is elaborated on in Section 7.4.2. The second call to the log manager on the last line returns the log records LSN, which is then written as `pageLSN` and possibly as `redoLSN` by the call to `logLSN()`.

The page interpreter may also decide to distribute redo and undo information for an operation onto two records. In that case, the undo information log record must precede the redo information log record. Otherwise, if the system crashes and only the redo log record was stored on disk, it is unable to undo the operation.

A typical case where the undo and redo information is distributed over two records is for bounded in-place updates as described in the previous section. The call that initiates the in-place update generates an undo-only log record, while the call that terminates the update writes a redo-only log record. In this way, the log record creation time is not extended over the complete in-place update time, nor is it necessary to buffer the before image somewhere until the operation is terminated and the log record can be written.

Redo Processing The recoverable page interpreters must refine the redoLogRecord to allow for the processing of log records during redo.

The implementation usually consists of a simple switch statement which extracts the redo information from the log record and calls the appropriate update method of the non-logging base class, as shown in Figure 7.17.

To correctly reflect the page state, the `pageLSN` and possibly `redoLSN` are set to the LSN of the redone log record, which is given to redoLogRecord as an argument.

Undo Processing Analogous to redo processing, the page interpreter must refine `undoLogRecord()` to process log records during undo (Sample Code in Figure 7.17).

The recoverable page interpreters use their own refined versions of the update function calls to perform the update. This causes log records to be written. Log records written

```
void PGE_RAL_LogSlottedPage::redoLogRecord(StmtCB *cb,
                                           LSN redolsn,
                                           REC_LogRecordPageUpdate *logr) {
  SlottedPageUpdateLogRecord *logrec=
    static_cast<SlottedPageUpdateLogRecord*>(logr);

  [...]
  switch(logrec->operationKind()) {
  [...]
    case PGE_OPER_SLOTTEDPAGE_DELETERECORD:
      PGE_RAL_SlottedPage::deleteRecord(cb,slot(logrec->slotNo()));
      break;
  [...]
  }
  logLSN(redolsn);
}
```

Figure 7.17: Redo processing in page interpreters

```
void PGE_RAL_LogSlottedPage::undoLogRecord(StmtCB *stmtcb,
                                           REC_LogRecordPageUpdate *logr,
                                           uint16 logrecsize) {
  SlottedPageUpdateLogRecord *logrec=
    static_cast<SlottedPageUpdateLogRecord*>(logr);

  switch(logrec->operationKind()) {
   [...]
   case PGE_OPER_SLOTTEDPAGE_DELETERECORD: {
      Slot *sl=slot(logrec->slotNo());
      cptr_t afterimage=logrec->contents();
      insertRecordHere(sl,afterimage, logrec->afterSize());
      break;
   }
   [...]
  }
}
```

Figure 7.18: Undo processing in page interpreters

during undo are compensation log records. They do not need to contain undo information, and have to be chained not to the last log record written by the transaction, but to the log record that has to be undone next,

The update function (refer again to the sample in Figure 7.16) has to check the transaction control block to see whether undo is in progress and compensation log records have to be written instead of regular log records.

Page flush notification To adhere to the write-ahead-logging protocol, the system must make sure that all log records created for a page have been safely flushed to disk before flushing the page itself. In Natix, this is the responsibility of the recoverable page interpreter who has created the log records. Before the associated page is written to disk, the buffer manager notifies the page interpreter by calling `prepareWrite()`.

The decision not to have the recovery manager flush the log itself is due to the desire for loose coupling. In the Natix architecture, the buffer manager does not need to know about the precise representation of the log and the `pageLSN` inside the page interpreter, and does not need to know whether a page is recoverable or not. The page interpreter may also perform some housekeeping before the page is flushed. See Section 7.6 for an example where this is used to optimize recovery performance for XML data.

In most cases, however, the refined `prepareWrite()` function of the page interpreter will just request a log flush up to the page's `pageLSN`, by calling the log manager's `flushLog()` function.

Logical Logging

Some data types are implemented by extending the interface of an existing data type.

For example, the XML page interpreter class implementation is based on the slotted page, and the XML page interpreter class is derived from the slotted page class, adding XML-specific functions to the interface. The regular page interpreter functions are used to implement most of the functionality of the new interface.

To create a recoverable XML page interpreter, it seems to be a clever idea to use the template method described in the page interpreter class hierarchy design above. The recoverable XML page interpreter class is created by instantiating the template recoverable slotted page interpreter class with the regular XML page interpreter class. This refines the slotted page functions to their logging versions. As a result, calling XML update functions will use the logging versions of the slotted page functions, creating log records in the process.

While the approach above is possible, it will create more log information than necessary, because all modifications to the record are logged on a ”physical” level. For example, to maintain the local pointer structure in the XML subtree record, more than one part of the record has to be modified if a new node is added, creating several log records.

If a special recoverable XML page interpreter is implemented instead, it can log the modifications on the ”logical” level of the XML specific interface. Just logging the insertion of a node will implicitly contain the necessary information to maintain the pointer structure, resulting in a more compact log representation.

It is also easier to devise advanced logging techniques if log records are used that know about the logical operations on the data type (see Section 7.6).

In fact, ARI**ES** itself was designed to **E**xploit **S**emantics by operation logging on pages instead of logging operations against the ultimate underlying representation of a page: A fixed size byte array.

7.4.6 Buffer Manager

Forward Processing

To support the correct operation of recovery, the buffer manager notifies each page interpreter by calling its `prepareWrite()` function before the associated page is written to disk.

The page interpreter uses this notification call to enforce write-ahead-logging and possibly other, data type specific recovery actions, for example reserved space collection (Section 7.5.4).

To support efficient recovery, the buffer manager writes log records after pages have been flushed to disk so that they do not need to be examined by restart redo. The log record kind for these log record is `REC_LRK_BUFFERMGR`, the operation kind is `REC_OPER_PAGEFLUSH`, and the class `REC_LogRecordBufferChange` is used. The log records simply contain the PID of the flushed page. If a page is dropped from the buffer manager without writing it because it was deallocated, this is also recorded in a log record because redo work on the page then becomes unneccesary.

The buffer manager maintains the `redoLSN` field of the buffer frame control block. If a page is newly installed in the buffer manager, and after a page has been written to disk, the `redoLSN` field is set to null. Synchronization for accessing the `redoLSN` is coupled to synchronization of the dirty flag for the page:

The `redoLSN` is set to an update operation's LSN *before* the dirty flag is set and while an exclusive latch on the page is held.

The `redoLSN` is cleared only *after* the dirty flag is cleared and while the hash bucket containing the frame control block is locked using its mutex (refer to Section 5.3 for synchronization of the dirty flag). The `redoLSN` may only be set from null to non-null and from non-null to null. It is never updated from one non-null LSN to a different non-null LSN. This protocol allows read access to `redoLSN` values without latches on the pages, as explained below under checkpointing.

To support log truncation, the buffer manager provides a function that returns the page in the buffer which has the smallest `redoLSN` value. This page's LSN is called `systemRedoLSN`. The log manager uses this function to make it possible to drop parts of the log by flushing stale pages, see Section 7.4.2.

Checkpointing

When taking a checkpoint, the recovery manager calls the buffer manager's `checkpoint()` method. If a heavyweight checkpoint is desired, the recovery manager will call the `flush()` method of the buffer manager first, to write all modified pages to disk (Section 7.4.8).

The buffer manager will then write log records describing the dirty pages in the buffer. For each dirty page, its PID and the `redoLSN` value are written to a log record.

The collection of dirty page information is performed in several increments to reduce contention on the buffer manager and log manager because regular system operation continues in parallel to checkpointing. Each incremental step collects information about a fixed number of hash buckets, locking the mutex for each hash bucket only once. Hence, locking of the buffer manager structures is minimized and mutexes are freed regularly. This may cause some pages that got dirty during the checkpoint not be included in the buffer manager's checkpoint log record. Since the updates to those pages are logged between the begin and end checkpoint log records, restart recovery will be able to deduct the dirty state for the affected pages anyway (Section 7.4.8, page 169).

While the PID access hash table of the buffer manager needs to be accessed with synchronization as explained in the previous paragraph, latching all of the individual pages during every checkpoint would be very resource-intensive. Fortunately, although the `redoLSN` values are modified by update operations on the page, it is not necessary to always latch the pages themselves. The protocol to modify `redoLSN`, as explained above under forward processing, allows to read `redoLSN` values in a conflict-free manner most of the time without latching.

To obtain the `redoLSN` for a page during a checkpoint, the page's dirty flag is checked first.

If it is set, the `redoLSN` is read while still holding the associated hash bucket mutex. It is not possible to have a conflict with a concurrent updater of the `redoLSN` because if some other thread wants to clear the `redoLSN` and dirty flag, it needs to acquire the mutex on the associated hash bucket, which the checkpoint code still holds. Apart from clearing it, there are no other update operations on `redoLSN`s of dirty pages. Hence, there never are conflicting updaters, and the obtained `redoLSN` value can be used without fear of conflict, without latching the data pages.

Every time information about a fixed number of dirty pages has been collected, a single log record describing them is completed. By having several dirty pages' information share a log record, logging overhead is reduced.

If the dirty flag is not set, then either the page is not dirty and the `redoLSN` value is irrelevant, or the page is still latched and fixed by an updater who has not yet set the dirty flag. The latter case can be detected by checking the fixcount and latch in the buffer frame control block. Only if the dirty flag is not set and the page is fixed and latched in exclusive mode, we give up the hash bucket mutex and request a shared latch for the page. If the page is still not dirty, it may be ignored, otherwise it is included in a dirty page log record as above. This limits the data page latching to a few unlikely conflict cases, without requiring extra synchronization of `redoLSN` values as proposed for example by Mohan et al. [66].

7.4.7 Transaction Manager

The main job of the transaction manager is to provide an interface allowing applications to initiate and terminate transactions and to mark transaction states as savepoints and to reestablish them.

Hence, the first responsibility of the transaction manager is to allow applications to obtain and release transaction control blocks, i.e. to provide transaction control block memory management.

To avoid multiple fixed size per-transaction memory areas, the transaction control blocks centrally store the per-transaction information of all system components in the recovery subsystem. As a result, the second responsibility of the transaction manager is to provide an interface for the other components to read and write their information. This includes notification of transaction state changes, for example if the application requests to terminate, savepoint or rollback the transaction.

We first briefly discuss requirements with regard to transaction control block management and the notification mechanism, and then describe the interfaces to the transaction manager for its different clients, namely the applications, the segments, the log manager and the recovery manager.

Transaction Control Block Management

For the application, the address of its transaction control block sufficiently identifies a transaction. To write log records, however, it must be possible to identify transactions in a persistent way because after a crash the transaction control blocks might be at a different memory location.

To support fast recovery, mapping the persistent identifier to the corresponding transaction control block must be a fast operation.

To ease the design and implementation, it seems reasonable to provide an upper bound to the number of concurrent transactions.

To allow internal system operations to be handled in a uniform way with application operations, the transaction manager has to maintain a system transaction that never aborts and is always active. This transaction control block is used to perform operations that require transaction data structures but are not associated with a specific transaction. Examples include modification of system metadata structures, and writing log records for checkpoints.

Pending Actions

It must be possible for other components of the system to be notified when certain transaction-related events occur.

For example, after a transaction commits, the partition manager must return partition files to the operating system if a transaction has dropped a partition. This is only allowed after the transaction commits because otherwise a rollback would need a complete before image of the file to undo the drop.

In addition to such a basic pending action mechanism, which is already described in ARIES [66], Natix needs to support more flexible notifications about transaction-related events. For example, let us consider main memory structures that duplicate on-disk structures for performance reasons. Apart from physical metadata (Sections 5.6 and 7.5) and the Physical Schema (Section 6), such structures are common with object-oriented database interfaces, where an object graph is represented in a more C++-friendly fashion by using C++ objects for nodes and main-memory pointers for edges. An XML example for such an interface is the Document Object Model DOM (Section 2.4.1), as implemented in the Xerces XML Parser [73].

If such an interface is used, it must be possible to perform recovery on it to keep it consistent with the data in secondary memory. Therefore, the components containing such data must be notified if the transaction aborts, commits or rolls back to a savepoint, in order to maintain the main memory structures accordingly. This could be done using regular log records, however, a main memory pending actions approach is more flexible and does not put unnecessary extra load on a bottleneck ressource, in this case the log manager.

To capture the requirements discussed above, we introduce the concept of *pending actions*, which describe actions that have to be performed on certain events. For our purposes, we need to be notified at the following occasions:

precommit Actions to perform before a transaction commits.

postcommit Actions to perform after a transaction commits.

subcommit Actions to perform before a transaction writes a subcommit log record, i.e. before an L1 operation is completed.

savepoint Actions to perform before a savepoint is established.

rollback Actions to perform when the transaction rolls back to a specified LSN.

Note that often the same object requires notification on several events. For example, if a transaction commits, a main memory object has to be written to disk, but if it aborts, the same object has to be deleted instead. The implementation should be able to exploit this to avoid excessive memory usage by the pending action mechanism.

Application Interface

`beginTransaction()` When a transaction is initiated, a transaction control block is initialized and returned. If the maximum of concurrently allowed transactions is already active, 0 is returned.

No transaction begin log record is written, as many transactions are read-only and do not need recovery, and as a consequence no log records are necessary. For update transactions, the first log record of a transaction is considered its begin transaction log record.

`commitTransaction()` When the application requests a transaction commit, the transaction manager performs all precommit operations in the transactions pending actions list. Then, as described in section 7.2.2, a commit log record is written and the

```
class TRN_TransactionManager {
public:
  typedef LSN SaveID;

  // called by application
  TaCB *beginTransaction(SeCB *);
  void commitTransaction(TaCB *);
  void abortTransaction(TaCB *tacb);
  SaveID saveTransaction(TaCB *tacb);
  void rollbackTransaction(TaCB *t, SaveID s);

  // called by segments with L1 operations
  LSN initiateL1(TaCB *tacb)
  void notifyCompleteL1(TaCB *tacb, LSN begin)

  // called by log manager
  LSN notifyNewLSN(TaCB *tcb, LSN newlsn)
  LSN notifyNewUndoLSN(TaCB *tcb, LSN newlsn)

  LSN oldestUndoLSN();
  // called by recovery manager
  void checkpoint(TaCB *tacb);
  void analyzeLogRecord(StmtCB *stmt, REC_LogRecord *, uint16);

  TaCB *provideTaCB(SeCB *, TransID, const LSN &first, const LSN &current);
  void releaseTaCB(TaCB *);

  void prepareL0Undo(TaCB *, LSN);
  void prepareL1Undo(TaCB *, LSN);
  void completeUndo(TaCB *);

  uint32 getConcurrentTransactions() const;
  TaCB *getTransaction(TransID id);
  TaCB *getSystemTransaction();
};
```

Figure 7.19: Transaction manager interface

log manager is instructed to flush the log up to that record. Afterwards, the postcommit pending actions are executed. Finally, the transaction control block is released.

`abortTransaction()` The transaction manager uses the undoTransaction call provided by the recovery manager to undo all the transaction's updates. This also causes all rollback actions in the pending actions list to be executed. A transaction abort log record is written and the transaction control block is released.

`saveTransaction()` This call establishes a transaction savepoint.

The transaction manager first executes the savepoint actions in the pending actions list. Then it consults the transaction control block to find out which operation lead to the current transaction state and returns its LSN as a marker for the current transaction state.

`rollbackTransaction()` This call allows to return to a previous transaction state by giving the `savepointLSN` the transaction is supposed to roll back to.

First, all rollback actions in the pending actions list are executed which were added after the specified savepoint was taken. Then, the recovery manager is called to undo all operations on data structures that were performed after the savepoint was taken.

Transaction Control Blocks

The transaction conrol block's interface is shown in Figure 7.20. The first part of the interface deals with fixed-size per-transaction data and is described below, and the second part deals with the pending actions mechanism, and is explained in the next subsection. We start with a brief description of transaction control block management.

Since we only need a fixed number of transaction control blocks, they are preallocated in one array when the transaction manager is constructed. We can use a small integer index into that array as `transactionID` to identify transactions, allowing very fast mapping from the `transactionID` onto transaction control blocks. This `transactionID` can also be used as an identifier in log records because at any given point in the log, the `transactionID` uniquely specifies a single transaction, although the same `transactionID` may be reused many times during a system run. The position in the log determines which transaction is referred to. We only have to make sure that transactions are given their original `transactionID` during redo, which is not difficult because the array of transaction control blocks and all modifications to it are recorded in the log. This log information can be used to maintain the transaction array in the same state as it was during Forward processing of any log record.

The currently unused control blocks are linked in a singly linked free list, using their `transactionID` fields as link. Natix uses a 2-Byte integer to identify transactions.

`getSeCB()` returns the associated session control block.

`getOutStream()` returns the output stream for this transaction.

```
class TaCB {
public:
  SeCB* getSeCB() const;
  std::ostream* getOutStream() const;
  std::ostream* getErrStream() const;

  TransID transactionID() const;
  void transactionID(TransID tid);

  LSN transactionLSN() const;
  void transactionLSN(LSN lsn);
  void currentLSN(LSN c);
  LSN currentLSN() const;
  void undoLSN(LSN c);
  LSN undoLSN() const;
  void nextL1UndoLSN(LSN c);
  LSN nextL1UndoLSN() const;

  void setL0UndoInProgress(bool f);
  bool isL0UndoInProgress() const;

  void setL1UndoInProgress(bool f);
  bool isL1UndoInProgress() const;

  class PendingAction {
  public:
    enum ProgressMode {
      UNDEFINED,
      DELETE_CONTINUE,
      DELETE_STOP,
      REMOVE_CONTINUE,
      REMOVE_STOP,
      KEEP_CONTINUE,
      KEEP_STOP
    };

    virtual bool isSavepointMarker() const;
    virtual ProgressMode precommit(TaCB*);
    virtual ProgressMode subcommit(TaCB*);
    virtual ProgressMode postcommit(TaCB*);
    virtual ProgressMode savepoint(TaCB*);
    virtual ProgressMode rollback(TaCB*,LSN targetLSN);
  };

  void lockPendingActions()
  void unlockPendingActions()

  void addPendingAction(PendingAction *);
  void removePendingAction(PendingAction *);
};
```

Figure 7.20: Transaction control block interface

`getErrStream()` returns the error stream for this transaction.

`transactionID()` returns the current `transactionID`.

`transactionLSN()` returns the LSN of the first log record written by the transaction and the Null LSN if the transaction didn't write any log records yet.

The LSN of its first log record is also considered a unique and persistent identifier for an update transaction.

`currentLSN()` returns the LSN of the last log record written by the transaction.

`undoLSN()` returns the LSN of the last undoable log record written by the transaction.

`nextL1UndoLSN()` During L1 undo returns the operation that has to be undone after the current L1 undo operation is completed.

`isL0UndoInProgess()` returns true while L0 undo is being performed.

`isL1UndoInProgess()` returns true while L1 undo is being performed.

Pending Actions Interface

Pending Actions are managed using a variant of the observer pattern [30]. The *pending actions list* is maintained in the transaction control block as a doubly linked list of objects belonging to a class which is derived from the `PendingAction` base class.

A subsystem may create pending actions for a transaction by adding an object of a class defined by the subsystem to the list. The derived class has to refine the methods for those events it wants to be notified about. If the specified event occurs, the transaction manager will invoke the appropriate methods on the objects in the list, beginning with the last object and traversing the list backwards.

For example, if the partition manager wants to be notified after the transaction has committed to drop partition files, it uses an object of the DropPartition class which is derived from PendingAction and has the `postcommit()` method refined.

Each such object may represent pending actions for more than one event. This reflects the fact that often several pending actions for the same object but different events are necessary. In such cases, there is no need to allocate several small pending action objects. Instead, one object is sufficient which belongs to a class that has several refined event methods.

`addPendingAction()` appends a pending action to the end of the list.

`removePendingAction()` removes a pending action from anywhere in the list.

`precommit(),postcommit(),subcommit(),savepoint()` these methods on items in the list are called on each object on the pending action list before and after a commit, and before a savepoint, respectively.

They may return a `ProgressMode` value which determines how processing of the pending action list proceeds. If the `ProgressMode` value contains CONTINUE, the pending actions list is traversed further. If the `ProgressMode` value contains STOP, the pending action list traversal stops.

The KEEP, REMOVE and DELETE values determine memory management. Items returning KEEP stay in the list after they have been processed. items retuning REMOVE are removed from the list, and DELETE items are removed and destroyed using the C++ delete operator. This allows a flexible memory management for pending actions.

`rollback()` The rollback event method is called with a parameter specifying the savepointLSN. This can be used by derived classes to make their action dependent on how far back in the log the rollback is going to go.

Segment Interface

High-concurrency segments need to store information about ongoing L1 operations for each transaction (Section 7.4.4). The following calls represent the interface for this.

`initiateL1()` The segments use this call to indicate the begin of an L1 operation. During forward processing, this will take a savepoint and return the savepoint LSN as begin L1 marker.

During L1 undo processing (see Section 7.4.8 for more), this call returns the `nextL1UndoLSN` field of the transaction control block.

`notifyCompleteL1()` This call is used by the segments to indicate the end of an L1 operation. It will cause the subcommit pending actions to be executed. During L1 undo, it resets it the L1 undo flag in the transaction control block to false and sets the L0 undo flag back to true.

Log Manager Interface

The log manager calls the transaction manager to update the various LSN fields in the transaction control block when new log records are written.

`notifyNewLSN()` Every time a transaction writes a redo only log record, the log manager notifies the transaction manager of the new LSN. The LSN is stored in the transaction control block as last LSN written by that transaction. If the transaction has not had any log records yet, the `transactionLSN` value in the transaction control block is set to the log record's LSN.

The current `undoLSN` of the transaction control block is returned and used by the log manager to maintain the transactions `nextUndoLSN` chain in the log.

`notifyNewUndoLSN()` If a transaction writes an undoable log record, the log manager calls this function. It is similar to notifyNewLSN, but in addition also updates the `undoLSN` of the transaction to the record's LSN.

`oldestUndoLSN()` The oldestUndoLSN is determined by computing the lowest `transactionLSN` value of all transactions.

Callers of this function take advantage of the fact that no operation with a smaller LSN is in danger of being undone. For example, when truncating the log, the log manager needs to know how far back in the log potential undo processing runs may reach. The log may not be truncated beyond this point, otherwise some transactions may not be able to roll back. Section 7.5.4 will give another application, and yet another appears in a paper by Mohan [63].

Recovery Manager Interface

The recovery manager needs to obtain and release transaction control blocks during restart analysis and needs to maintain the LSN fields in transaction control blocks during undo:

`provideTaCB()` This call creates a new transaction control block with the given transaction ID, `transactionLSN` and undoLSN. It is used during analysis to recreate transaction control blocks that have not been logged in a checkpoint.

`releaseTaCB()` is used to drop transaction control blocks during analysis if a terminating log record (commit or abort) has been found. It is also used internally by the transaction manager after `abortTransaction()` or `commitTransaction()` are called.

`checkpoint()` is called during checkpoints and writes log records describing all currently active transactions.

For each transaction control block that is used, the transaction manager writes one log record of `kind() TRANSACTIONMGR` using the class `REC_LogRecordActiveTransaction` (Figure 7.6).

`analyzeLogRecord()` is called during analysis to recreate transaction control blocks for all transactions that were active during the checkpoint.

`prepareL0Undo()`,`prepareL1Undo()` are called before an L0 or L1 log record is undone, respectively. The methods set the undo flags correctly and update the `undoLSN`/`nextL1UndoLSN` fields in the control block.

`completeUndo()` terminates undo processing by resetting the undo flags.

7.4.8 Recovery Manager

The recovery manager is implemented as a stateless class whose interface allows to invoke the core recovery processes. These are transaction recovery, restart recovery, and checkpointing. We describe the recovery manager's interface (Figure 7.21) first and then elaborate on some details of its implementation.

```
enum REC_OperationKindCheckpoint {
  REC_OPER_RMGR_CHECKPOINT_UNDEFINED  =0,
  REC_OPER_RMGR_CHECKPOINT_INITIAL    ,
  REC_OPER_RMGR_CHECKPOINT_RESTARTREDO,
  REC_OPER_RMGR_CHECKPOINT_RESTART    ,
  REC_OPER_RMGR_CHECKPOINT_REGULAR    ,
  REC_OPER_RMGR_CHECKPOINT_SHUTDOWN
};

enum REC_CheckpointEffort {
  REC_CHECKPOINT_UNDEFINED=0,
  REC_CHECKPOINT_HEAVYWEIGHT=1,
  REC_CHECKPOINT_LIGHTWEIGHT=2
};

typedef uint8 REC_CheckpointType;

class REC_RecoveryManager {
public:
  static void undoTransaction(TaCB *tacb, LSN targetlsn);
  static void abortAllActiveTransactions(TaCB *);
  static TaCB * restart(GloCB *glocb);
  static void checkpoint(TaCB *tacb,
                         REC_CheckpointType oper,
                         REC_CheckpointEffort effort);
};
```

Figure 7.21: Recovery manager public interface

`undoTransaction()` The undoTransaction function rolls back all changes made by the specified transaction up to, but not including the operation specified by targetLSN.

`abortAllActiveTransactions()` The `abortAllActiveTransactions()` function rolls back all updates of all currently active transactions and removes the transaction control blocks from the transaction manager. It is used during restart undo and before system shutdown. The system transaction is given as parameter.

`restart()` The restart function initializes the recovery systems services. Necessary restart recovery is performed by invoking the analysis, redo and undo phases. The control block of the system transaction is returned.

`checkpoint()` Performs a system checkpoint. The system transaction is given as parameter, because log records written during the checkpoint will be associated with the system transaction.

The CheckpointType specifies the reason for the checkpoint. Apart from regular checkpoints during system operation, the system takes checkpoints after redo and undo recovery during restart, and before system shutdown.

CheckpointEffort specifies whether a lightweight or a heavyweight checkpoint is required. A lightweight checkpoint only causes metadata written to disk that will make restart recovery faster. A heavyweight checkpoint also flushes all dirty pages in the buffer manager to disk.

Transaction Undo (R1 Recovery)

Leaving aside L1 operations, transaction undo is relatively simple (Figure 7.22). We describe the Natix implementation of transaction undo without L1 operations first, and then explain how to support L1 operations.

A general design goal was to have as few recovery specific code as possible, which in the undo context means we do not want to write special undo code for every segment type. We rather strive to use the regular forward processing code to perform updates during undo as well.

Regular L0 undo Starting from the current `undoLSN` in the transaction control block, the undo chain of log records for the current transaction is traversed backwards using the log manager. The next record of the chain is always pointed to by a log record's `nextUndoLSN` pointer.

For every L0 log record, undo is initiated by setting a flag in the transaction control block indicating that L0 undo is in progress. This is done in the transaction manager's `prepareL0Undo()` function, and causes all page-level log records to be created as redo-only compensation log records, as shown in the example in Section 7.4.5.

In the `prepareL0Undo()` method, the `undoLSN` field of the transaction control block is maintained to contain the LSN of the log record that has to be undone next. The

```
void REC_RecoveryManager::undoTransaction(TaCB *tacb,
                                          LSN targetlsn) {
  GloCB *glocb=tacb->getSeCB()->getGloCB();
  REC_LogManager *logmgr=glocb->getLogManager();
  SMR_SegmentManager *segmgr=glocb->getSegmentManager();
  TRN_TransactionManager *trnmgr=glocb->getTransactionManager();

  REC_LogManager::iterator logscan;
  BufferFrameCB *undoframe=0;
  SEG_Segment *undosegment=0;

  if(!tacb->undoLSN().isNull()) {
    logscan = logmgr->find(tacb->undoLSN());

    while(targetlsn<tacb->undoLSN()) {
      REC_LogRecord *logrec=logscan.logRecord();

      if(logrec->isUndo()) {
        switch(logrec->kind()) {
          case REC_LRK_PAGE_UPDATE:
          case REC_LRK_SEGMENT_UPDATE: {
              REC_LogRecordSegmentUpdate *seglogrec;
              seglogrec=static_cast<REC_LogRecordSegmentUpdate*>(logrec);

              [... if different undosegment than last time,
                     unfix undoframe if fixed, and
                     open correct segment ...]

              if(logrec->kind() == REC_LRK_SEGMENT_UPDATE)
                trnmgr->prepareL1Undo(tacb,logrec->transUndoLSN());
              else
                trnmgr->prepareL0Undo(tacb,logrec->transUndoLSN());

              undosegment->undoLogRecord(tacb, undoframe,
                                         logscan.lsn(),
                                         seglogrec,
                                         logscan.logRecordSize());
            }
            break;
          default: // ignore all others (e.g. CLRs)
            break;
        }
      }

      logscan.skipToLSN(tacb->undoLSN());
    }

    [... unfix the last modified data page and close the segment...]

    trnmgr->completeUndo(tacb);
  }
}
```

Figure 7.22: Recovery manager undo loop

log manager initializes the `nextUndoLSN` of undoable log records with the transaction control block's `undoLSN` by default. Hence, the segments and pages may use regular logging operations to undo an operation, and proper compensation log record chaining is carried out automatically.

For every log record, we extract the associated segment and open it. Then, the log record is forwarded to the segment's undoLogRecord function for processing.

We make sure that we do not unnecessarily open and close segments and fetch and latch pages by caching main memory segment objects and frame control block pointers between calls. In this way, if several updates operate in sequence on the same segment and/or the same page, the segment is not closed and reopened, and the page is not unfixed and refetched. This is hidden by comments in Figure 7.22 to avoid excessive source code display.

L1 operation undo When encountering an L1 subcommit log record, as above the record is forwarded to the segment for undo processing. The segment may then use its regular segment-level operations to perform the logical inverse of the original action.

However, the performance of the segment-level inverse operation itself is an L1 operation. There are some special requirements with respect to inverse L1 operation's log records (as explained in Section 7.2.3), and we need to set up the transaction control block correctly to meet these requirements. The prepareL1Undo method of the transaction manager performs maintenance of the transaction control block as follows.

undo partial inverse L1 operation We may not write compensation log records for the L0 operations that are executed as part of the inverse L1 operation, but need to write regular redo-undo records. For this reason, before the L1 log record is forwarded to the segment, the L0 undo flag in the transaction control block is set to false.

As a result, the execution of the L1 operation logs undoable log records, and if an L1 operation is not completed before a crash, it is undone by regular restart L0 undo.

After the log record is undone, the L0 undo flag is set back to true.

redo of incomplete inverse L1 operation The first L0 log record that is written as part of the execution of the inverse L1 operation has to be chained to the subcommit record of the L1 operation that is being undone.

This is achieved by leaving the `undoLSN` in the transaction control block unchanged before forwarding an L1 log record to a segment. That way, the `undoLSN` still points to the L1 operation that is undone, and the first L0 log record written on behalf of the inverse L1 operation will use that `undoLSN` as `nextUndoLSN`, as desired.

L1 restart idempotence To achieve restart idempotence for L1 operations, we store the `nextUndoLSN` of the forward L1 operation in the `nextL1UndoLSN` attribute of the transaction control block and set a flag indicating that L1 undo is in effect.

When the segment invokes the inverse L1 operation using a regular update method, the initiateL1 call on the transaction manager that announces the begin of the inverse

L1 operation will return the `nextL1UndoLSN`. As a result, the segment properly sets the `nextUndoLSN` value of the inverse L1 operation's subcommit record.

After the segment returns from the undoLogRecord call, we reset the L1 undo flag back to false.

Abort All Transactions

While it is possible to abort all active transactions by calling the transaction managers abortTransaction method for each transaction one after another, there is a more efficient approach.

Transaction undo causes each transaction to scan its `nextUndoLSN` chain backwards. The naive algorithm for aborting all transactions scans the log backwards multiple times, resulting in repeated reads of the same log pages.

It is more efficient to only scan the log once and perform all transaction undo processing in parallel. In each step of this algorithm, the largest `undoLSN` of all transactions still active is determined and that log record is processed next, resulting in a backwards sequential scan of the log.

To allow for system shutdown and restart undo to employ such a more efficient implementation, the recovery manager provides the entry point `abortAllActiveTransactions()`.

System Restart

To initialize the recovery subsystem, the application calls the `restart()` method of the recovery manager.

The recovery manager first reads the position of the last successful checkpoint from the log partitions master segment by calling `readAnchor()` on the log manager. A log analysis follows to see whether the system was properly shut down or a crash occured, and whether there were any dirty pages or incomplete transactions when the system stopped.

If restart analysis finds dirty pages, restart redo is called to restore their contents as of the time of the system crash. If any work is performed during restart redo, a lightweight checkpoint is taken. This checkpoint will include information on the precise on-disk state of the presumed dirty pages, speeding up following restarts. Since it is a lightweight checkpoint, it does not significantly slow down restart.

If there were any incomplete transaction, restart undo processing is performed, rolling back those transactions and logging the necessary operations. A further lightweight checkpoint is taken after restart undo to provide future restarts with a system state that already includes the necessary undo operations.

Analysis

During analysis, the recovery manager scans the log starting from the last begin checkpoint record, which is obtained as `anchorLSN` from the log manager, to the end.

During the log scan, if operations of a transaction are encountered for a transaction which is not yet known to the transaction manager, the recovery manager notifies it using `provideTaCB()` and returns a transaction control block. Both `transactionLSN` and `undoLSN` of this new transaction control block are set to the current log record's LSN. Afterwards, depending on the kind of log record, one of the following actions is performed:

page modifications The respective pages are inserted into the dirty page set. The dirty page set is implemented as hash table mapping PIDs to their respective `redoLSN`s.

If the log record is undoable, the associated transaction control block's `undoLSN` is set to the log record's LSN. Otherwise, the log record's `nextUndoLSN` value is stored as new `undoLSN`.

buffer manager checkpoint record The contained PIDs and their `redoLSN`s are also inserted into the dirty page set. If a page flush or evict records are encountered, the PID is dropped from the dirty page set.

commit or abort records The respective transaction control block is released by calling `releaseTaCB()` on the transaction manager.

active transaction log records were written by the transaction manager (Section 7.4.7) using the `REC_LogRecordActiveTransaction` log record class (see Figure 7.6). The recovery manager forwards to the transaction manager using its `analyzeLogRecord()` call, which will install transaction control blocks for the contained transactions.

At the end of the analysis phase, the dirty page set will contain all pages that may have been dirty when the system crashed, and the transaction manager will have control blocks for all loser transactions, whose `undoLSN` and `transactionLSN` fields are properly initialized.

Restart Redo Processing

Redo processing determines the page from the dirty page set with the smallest `redoLSN`, called `systemRedoLSN`. It then scans the log in forward direction starting from that LSN.

It determines the affected segment for each record and opens it if it is not already open yet because it was also target of the previous log record.

In case of page-level log records, it is verified if the affected page is contained in the dirty page table, and if so, the log record is forwarded to the segment. From the redoLogRecord call to the segment, the current pageLSN for that page is returned. This `pageLSN` is maintained in the dirty page set to skip redo of log records whose updates are already contained on the page.

In case of segment-level log records, they are immediately forwarded to the segment.

Restart Undo Processing

Undo processing consists of a simple call to `abortAllActiveTransactions()`.

Checkpointing

When the recovery manager is instructed to take a checkpoint, it performs the following actions in the order specified:

1. write a begin checkpoint log record
2. call the checkpoint method of the partition manager
3. call the checkpoint method of the transaction manager
4. if the checkpoint is supposed to be heavyweight, call the buffer manager's flush method
5. call the checkpoint method of the buffer manager
6. write an end checkpoint log record
7. update the `transactionLSN` value of the system transaction's control block
8. truncate the log (`truncateLog()`, see Section 7.4.2)
9. flush the log up to the end checkpoint log record
10. notify the log manager of the new anchorLSN, which is the LSN of the begin checkpoint log record, and instruct the log manager to write the anchorLSN to disk

Step 7 is necessary because the system transaction (Section 7.3.8) participates in the determination of the `oldestUndoLSN`. To allow the log to be truncated, it is necessary that this LSN is regularly moved forward.

7.5 Metadata Recovery

Physical metadata is the persistent metainformation required to access the storage engine's objects, such as partitions and segments. Metadata handling in Natix is explained in Section 5.6.

A proper recovery subsystem has to incorporate support for metadata recovery. Since Natix's metadata is materialized using regular storage engine structures, basic metadata recovery is mostly automatic.

However, access to the metadata structures needs to be highly concurrent. Locking them until transaction completion can effectively seralize all transactions because metadata access is fundamental to storage engine access.

High concurrency on data structures needs support by the recovery mechanism. We have seen this in the context of L1 operations (see Sections 7.2.3, 7.3.3, and 7.4.4). This section explains the issues raised by metadata recovery and discusses the solutions provided in Natix.

7.5.1 Instance Metadata: Partitions

The set of partitions that makes up the current instance is defined in a configuration file (Section 5.6.2).

The configuration file cannot be stored in any partition of the instance, as it is needed to access the partitions in the first place. As a consequence, the recovery subsystem cannot protect the configuration file against crashes.

Include Partition Metadata in the Log

To prevent the use of incorrect version of the configuration file, Natix records all partition information in the log when taking a checkpoint. In addition, every time a new partition is mounted or a partition is dropped, a log record with kind `PARTITIONMGR` is created.

The only information that must be correct in the configuration file is the partition information for the log partition. Usually, even very old backup versions of the configuration file contain the correct log partition. Even if no backup is available, the reconstruction of a simple configuration file pointing to the log partition is rather simple.

The configuration file can still be used to add partitions to the system, as the system uses the superset of all partitions found in config and log entries. A conflict exists if there are partition numbers which have different physical partitions assigned to them in the config file and the log. If there is a conflict between the information in the log and the config file, the system prints an error message explaining the problem, and refuses to mount the partitions in the configuration file.

Undo Processing for Dropped Partitions

Destroying partitions is a special situation, as we do not want to record before images for all pages in all dropped partitions. However, the transaction dropping the partitions might abort, making undo of the partition destroy operations necessary.

ARIES [21, 66] uses a pending actions concept, to defer the actual operation of dropping a partition until after the transaction commits. We use our pending action mechanism's `postcommit()` method (Section 7.4.7) in the same way[5].

Implementation

Including the partition metadata in the log is achieved using the same observer mechanism as explained in Section 5.2.

The recovery subsystem instantiates its own partition observer object using a derived class, and registers it with the partition manager. Every time after a partition is created or destroyed, the observer object is called by the partition manager, a log record is written describing the partition and the log is flushed up to this log record. During a checkpoint,

[5]For ease of exposition, we do not discuss how to guarantee the durability of a partition drop. The partition drop operation is written to the log after the commit record, and thus could be lost. See Dey et al. [21] for details and a solution.

log records are generated for all mounted partitions. Before a partition is dropped from the system, all its pages currently in the buffer are flushed.

During startup, the coniguration file is only read until a log partitition is found. Only this log partition is mounted, and system restart proceeds. During restart analysis, the partition log records found in the log are processed, mounting or removing partitions from the current instance.

After restart is completed, the remaining entries in the configuration file are read, and potential conflicts are reported. New partitions in the configuration file are mounted using regular calls, causing log records to be written, preventing information about the new partitions to be lost if the configuration file is lost.

Note that if a partition was dropped, and the same partition number was assigned to a different partition afterwards, this does not cause problems for the recovery mechanism in the event of a crash. No log records from the first lifetime of the partition number are redone on the second partition, as all the old partition's pages were flushed before it was unmounted, and because format log records for the pages in the new partition are used (see Section 7.5.6). Since information about the new partition is contained in the stable log before it is used, there is also no conflict if the configuration file does not yet include the new partition.

7.5.2 Partition Metadata: Master Segment

Every partition holds a master segment which contains information about the different segments stored on the partition (Section 5.6.3).

Master Segment Recovery

The Master Segment must always be a recoverable segment. This is necessary to have correct metadata for the segments when recovery is performed on them. It is also necessary to have recovery on the metadata of nonrecoverable segments, because such segments, which are typically used for temporary results in query processing, use up free space of the partition, which must be returned to the system in case of a crash.

The master segment is implemented as a simple slotted page segment, where each record describes metadata of one segment. The most important record types in the master segment are the segment descriptors and the extent tables (Section 5.6.3). Recovery for segment descriptors is straightforward. We now describe how extent tables are treated by recovery.

Extent Table Recovery

The master segment contains records which describe the extents belonging to each segment, the so-called extent table of the segment. New extents are added to a segment by modifying its extent table record.

Access to the extent table needs to be concurrent: If a segment grows due to updates of one transaction, other transactions need to be able to use the new pages, even if the transaction which caused the segment to grow has not yet committed.

Should this transaction roll back, other transactions may already have used the new extent(s). Hence, an extent once added to a segment may not be removed, except if concurrent updates are somehow suspended, for example by a segment-level exclusive lock.

As a consequence, the `grow()`-operation on segments becomes an L1 operation (refer to Section 7.3.3) of the master segment:

Growing a segment consists of removing the free space from the partitions free space pool and adding it to the extent table. By combining all these operations into one L1 operation that has the null operation as its logical inverse, we achieve the desired result: If the system crashes before the grow operation is finished, already performed modifications are rolled back, but once completed, a segment growth is not undone by transaction undo. This is the same as a so-called nested top action, as described by Mohan et al. [66].

But if segment growth was never undone, a problem would occur if the transaction that grew a segment was also its creator. If such a transaction rolled back, the segment would not exist any more, but the associated extents would. This inconsistent state has to be prevented by the recovery system.

Natix solves this problem by making segment creation also an L1 operation. As the logical inverse of segment creation we define segment destruction. By segment desctruction, we mean not only deletion of the master segment records of the segment, which would be the physical inverse operation. Instead, segment destruction includes release of all extents allocated by the segment. This is implemented in the master segment using the L1 operation framework (Section 7.4.4).

As a result, during undo of the segment-creating transaction, not only those extents are removed with which the segment was initially created, but all extents of the segment, including those that were added later in the transaction.

7.5.3 Partition Metadata: Free Extents

On every non-log partition, there exists a Free Extent Segment that manages the extents on the partition that currently do not belong to any segment.

Updates to the free extent segment need to be highly concurrent, as the free extent pool of a partition is used for all segments on that partition. This raises a couple of issues related to recovery that are discussed in the following.

Free Extent Merging

The Free Extent segment optimizes storage by merging adjacent extents. For example, suppose a transaction T_1 adds an extent of 1000 pages starting at page 123 to the free extent segment, which already contains a free extent of 23 pages starting at page 100. After the add operation, the two extents are merged into one free extent of 1023 pages starting at page 100.

Due to this merging, if a certain free extent was modified by one transaction, it must be possible for another transaction to modify it before the first transaction commits. As an extreme example, at the beginning there is only one free extent (the whole partition), and it is not practicable to defer concurrent transaction execution until the free space is fragmented into several extents. Hence, in Natix, only short duration exclusive locks are used to synchronize consumption of free extents.

The lock on the free extent table is not held until transaction commit. As a result, if the data structure employed to implement the free extent segment is larger than a page, then plain page-level physiological undo is a problem. The reasion is that inverse actions on the free extent segment are not necessarily the inverse of the original actions. As an example, assume transaction T_0 consumed 50 pages starting at page 50, changing an existing free extent of length 73 starting at page 50 into a free extent of 23 pages starting at page 100. Now T_1 consumes 3 pages starting at page 100, leaving a free extent of 20 pages starting at page 103. If T_0 were to rollback, it could not simply apply the inverse operation of its original update, as T_1 has meanwhile changed the free extent data structure.

The solution is to use L1 operations for updates to the free extent table (for treatment of segment L1 operations refer to Section 7.3.3). By defining logical undo that can determine how to merge free extents properly, correct recovery can be provided.

Deferred Release of Extents

Another problem with free extents occurs when a transaction T_0 adds an extent to a partition's free extent pool and then rolls back. If another transaction T_1 has consumed some of this free extent, T_0 cannot reclaim its pages without aborting T_1.

To avoid this situation, extents are not immediately freed, but only during commit processing:

When during forward processing an extent is supposed to be returned to the free extent pool, it is remembered in the transaction control block, using a `precommit()` pending action (Section 7.4.7).

Upon transaction commit, all extents stored in the transaction control block are added to the free extent segment, and the free extent segment and its page interpreters write log records for the operations. To prohibit other transactions from using the space before the transaction commits, the free extent segment is locked in exclusive mode during the add operations. This prohibits concurrent access to the free space segment, but only for a short time, as the free space modifications are performed right before commit processing is complete and the transaction releases its locks anyway.

After adding the free extents, the commit record is written and the log is flushed as usual.

Should the transaction abort, the list of free extents in the transaction control block is discarded, and if the abort occured during commit processing, free extents that already have been added to the free extent segment are removed as a result of normal transaction undo.

If the transaction aborts after the free extent has been added to the free extent table, the free extent table lock must be held at least until the free extent update has been undone.

This procedure is robust against crashes at any point:

Crashes during forward processing will make the transaction a loser transaction during restart. Its free space updates will not be executed, as they were buffered in the transaction control block and therefore did not survive the crash. If some of the updates had already been performed before the crash, and already made it to stable storage, they are undone as part of the normal restart undo processing.

If the transaction commits, the free extent segment changes are recorded in the stable log and thus are reexecuted during redo processing if they were not stored on disk.

7.5.4 Page Metadata: Reserved Space

In the previous section, the problem of freeing space from a transaction that may abort and thus need to reclaim that space was discussed on the partition level. The same problem also occurs at the page level:

Fine-grained locking on physical records within a page allows a transaction T_2 to use up space that resulted from a record delete operation of another transaction T_1. If T_1 aborts, it needs to use the previously freed space to undo the delete, but that space is already used by T_2, and the undo cannot take place.

Always Employing L1 Operations

L1 operations provide a solution for this. They allow to define undo operations that are not the page-level inverse operations of the original actions (Section 7.2.3). One could define every page-level operation an L1 operation whose logical undo operation included relocation of the record should the undo fail due to a full page. The problem seems to be solved. But there are two serious drawbacks:

First, although space acquisition for the undo operation is delegated to the segment level, it may still fail, making the undo impossible. Second, since the record is moved, all references to the record have to be updated as well, which in the presence of large indices may be very costly. Using a TID-like concept does not work here, as undo would still require some space on the page to store the new record location and its slot information.

Space Reservation

The problem can be solved without prevention of concurrent page updates by not immediately freeing space, but merely marking it *reserved* [53, 65]. Disallowing space-consuming operations of other transactions to reuse this reserved space until the reserving transaction commits guarantee that undo operations can always succeed.

This is what free extent management, as explained in the previous section, does when it records the extents that have to be freed as pending actions in the transaction control block, as explained in the previous section. The reserved space here is maintained in a main memory structure. Making the reserved space available to the system after commit is not costly in terms of resources because the number of extents freed by a transaction is usually limited, as an extent refers not to individual data items, but to a large set of pages. In addition, freeing extents is not a frequent operation (databases usually grow).

Classification of Page Space Reservation Techniques

Collecting reserved space on individual pages, however, requires a more thorough analysis and design, as record deletion and shrink operations are very frequent and the number of objects (pages) affected may be very large.

We now discuss the different general approaches to collect reserved space and then describe the method for reserved space on pages used in Natix.

We classify methods for reserved space collection into three categories:

Eager collection Upon commit, all pages on which the transaction reserved space for undo are visited and the space is freed.

Lazy collection Space is freed at some point after the transaction has committed.

Different strategies on when exactly space collection is performed include:

Explicit An explicit reorganization transaction regularly passes over the database and collects reserved space.

On-the-fly Regular transactions collect space during their normal activities. Transactions that update a page anyway can, as additional activity, mark space free that is no longer reserved.

Discussion of the techniques Eager collection is easy to implement. As an example, it is used by free extent management as described in the previous section, where all free extents are added to the free extent segment when a transaction commits.

The number of deleted records and affected pages per transaction can be very large. In general, it is not feasible to keep a main memory structure remembering all reserved space on all updated data pages and to revisit them on commit. Not only would the main memory structure be large and expensive to maintain, but transactions that have working sets larger than the buffer could actually double their I/O time, as they have to revisit every data page. In addition, successful transactions require extra CPU time to perform the space collection at commit.

Similarly, explicit collection is easy to implement, but expensive. The reorganiziation transaction must make a pass over the whole database, or alternatively use the recovery log to analyze which pages possibly contain reserved space. The pages have to be reloaded and rewritten, putting additional load on the system. The decision when to invoke the reorganization is difficult to reach, and increases administration effort for the system.

On-the-fly Collection in Natix

The drawbacks of the two approaches considered above make them undesirable for usage in a scalable DBMS.

Hence, Natix uses the on-the-fly approach to collecting reserved space on pages. Our method allows for amortizing the costs of space collection in a way that greatly reduces impact on system performance. On the other hand, it is more difficult to implement and needs additional fields on each page, at least one reserved space counter and an LSN field. To improve performance, it also can be configured to use additional space in the page interpreter objects as follows.

To do accounting for reserved space, a `reservedSpaceTable` is kept per page. The `reservedSpaceTable` comprises a configurable, fixed number n of pairs $(transactionLSN, reservedSpace)$, indicating how much space was reserved by transactions that started with or after the given LSN. The `transactionLSN` is the first LSN written by the transaction, which uniquely identifies each update transaction (Section 7.4.7). Some m of the n pairs (with $1 \leq m \leq n$) are stored in the page's contents, while the rest is stored only transiently in the page interpreter. The table is ordered from the youngest to the oldest `transactionLSN`, and the m entries with the youngest `transactionLSN` are stored on the page.

Reserved space for a transaction is added by adding it to all entries with the same or a greater `transactionLSN`. If less than n entries exist in the table, in addition to updating existing entries, a new entry with the current `transactionLSN` is created. Before trying to add a new entry, if n entries already exist and the updating transaction's `transactionLSN` is larger than all the existing LSNs in the `reservedSpaceTable`, the oldest entry in the table is dropped.

When a transaction consumes reserved space, that amount of space is subtracted from all entries with the same or a larger `transactionLSN`. Afterwards, all entries with a reservedSpace value of 0 are dropped.

In addition, the page's contents contains a new counter `markedSpace` akin to `free` and `unused` (see Section 5.4).

After a brief review of existing on-the-fly space collecting techniques, we explain how Natix maintains the `reservedSpaceTable` and `markedSpace` in the different modes of operation.

Related Work

A similar on-the-fly space-reserving scheme is presented in Lindsay et al. [53]. A configuration akin to the special case $n = 1, m = 1$ of Natix's method is described. Only the youngest transaction can reclaim reserved space already during forward processing.

In Mohan et al. [65], instead of transaction identifiers on the page, the concurrency controller is used to store information about which transactions have reserved space on a page. By requesting special space reservation locks in non-blocking mode, a transaction can find out whether some other transaction still has reserved space on a page, or whether the space can be regarded as free. Flags on the page mark whether memory areas are reserved or not.

Detailed Description

Forward Processing The actions that affect reserved space on a page are space-reserving operations (deleting records or part of a record), space-consuming operations (inserting or extending records) and reserved space collection. We now describe those actions in detail:

Space reservation If a transaction causes a record to **shrink**, first a shrink log record containing the before image of the removed section of the record is written. Afterwards, the shrink operation is performed, and the size of the removed part of the record is added to the `reservedSpaceTable`,

as described above.

Record **delete**s are not actually performed, instead records are only marked deleted by setting a slot flag and storing the `transactionLSN` of the deleting transaction (called *deletionLSN*) at the beginning of the record. This prevents other transactions from consuming the space necessary to undo the deletion, since the slot is still marked occupied and the space it refers to may not be collected by garbage collection.

Only the part of the record that is overwritten by the `deletionLSN` is stored as before image in the delete log record. Hence, a side effect of this method is that it keeps record deletion log records small. The size of the record is also added to the `reservedSpaceTable` as described above. The sum of the sizes of all records that are marked deleted is maintained in `markedSpace`.

Reserved Space Collection The page interpreters have an additional method to initiate reserved space collection. Reserved space collection is performed in two steps:

1. For records that are marked deleted, it is checked whether their `deletionLSN` is older than the `oldestUndoLSN` (see Section 7.4.7), meaning that the deleting transaction has committed. If yes, the record can actually be deleted without writing a log record. $markedSpace$ is decreased accordingly.
2. All space from entries in the `reservedSpaceTable` that have a `transactionLSN` earlier than the `oldestUndoLSN` can be removed from the table, adjusting the remaining entries by subtracting from them the reservedSpace amount of the youngest entry that was dropped.

   Since entries from the `reservedSpaceTable` can be dropped to keep the table small for efficiency reasons, some space remains reserved although it could be collected already. Reserved space by transactions that are older than the smallest `transactionLSN` in the `reservedSpaceTable` cannot be exactly associated with a single transaction. This space is kept reserved until the reserving transaction aborts or until `oldestUndoLSN` is larger than the smallest `transactionLSN` in the table. We tolerate this because this waste of space is bounded (since there can not be more reserved space than the page size. At some point either the shrink operations have reserved all the space, and no further updates on the page occur, or some space is reclaimed).

   Also note that, although the oldest entries are dropped from the table, which loses some of the information about which transactions reserved space, no *amount* of reserved space gets lost, since reserved amounts are cumulated in all younger entries as well. This also applies to entries that are lost because they have not been stored persistently (if $n > m$, not the whole reservedSpaceTable is stored on stable storage when the page gets evicted from the buffer).

Space-consumption When a space consuming operation fails because not enough free space is available, reserved space collection is performed and the operation is retried. Only if the operation still fails, regular garbage collection is attempted before failure is returned to the caller.

For space-consuming operations, it is desirable that transactions can reuse space they have reserved themselves. In order to safely reuse space, it is necessary to know how much space was reserved by the current transaction. This information can be obtained by looking at the reservedSpaceTable: Subtracting from the entry with the consuming transaction's `transactionLSN` the amount stored in the adjacent older entry, and also subtracting $markedSpace$ (as records that are marked deleted may not be reused, see below), yields the amount of reserved space that can be reused by the current transaction.

If there is no entry for the current transaction, or it is the oldest entry in the reservedSpaceTable, this information is lost, and reserved space cannot be reclaimed by the reserving transaction. Only operations that take place after the reserving transaction commits can reclaim the reserved space. So, only the youngest $n - 1$ transactions with updates on the page can reclaim space this way. This is not a severe limitation, as this mechanism is mainly intended to efficiently support a short sequence of grow/shrink updates on one record by the same transaction, as they occur as a result of the XML operations described in Section 5.7.5.

A transaction cannot reuse space it reserved by deleting a whole record, although using the `deletionLSN`, it would be simple to verify that a record was deleted by the same transaction that now tries to consume space on the page. But because small log records for deletions are used, after completely removing the record, there is no before image of the deleted record, which is required to undo the deletion.

If reusing space of completely deleted records is a serious performance advantage for an application, this can be enabled either

1. during record deletion, by writing deletion records with a complete before image, or
2. during the space consuming operation, by writing a special reclaim log record that really deletes a record that has only been marked deleted, including a full before image in the record.

Reusing deleted record space will reduce extra data page I/O only if the pages with deleted records are not reused before they get removed from main memory. On the other hand, reusing deleted record space will increase CPU usage and I/O in any case, as record contents have to be copied to log records and written to the log. Natix therefore does not reuse space of deleted records.

Space-consuming operations include in their log records the amount of consumed reserved space to allow correct undo of reserved space information (see below under undo processing).

After describing the page-level methods for dealing with reserved space, we also have to pay attention to how reserved space on the pages affects metadata of the segment. For efficiency reasons, the segments keep redundant information about page free space information in the Free Space Inventory, see Section 5.6.5. The FSI takes reserved space into account as follows:

The FSI value for a page is determined as if the reserved space was unused space. This results in those pages being chosen as targets for record insertion, leading to regular reserved space collections during normal operation.

If too many record insertions fail because of reserved space, it is possible to delay attempts to insert new records: While the information in the Free Space Inventory segment is calculated to include reserved space as free space, the Free Space Inventory Cache (Section 5.6.5) is updated in a way that counts reserved space as occupied (this is hinted at in McAuliffe et al. [55]). As a result, pages with reserved space are only targets for record insertion when there is enough unreserved space, or when they have dropped out of the FSI cache and the cache cannot satisfy a space request. This decreases the frequency of repeated unsuccessful insertion attempts while retaining on-the-fly reserved space collection.

Analysis and redo Analysis does not require special handling of intra-page space management. Neither does redo, which treats reserved space in the same way as forward processing.

Undo Processing During undo of a record deletion, the before image in the log record is used to restore the record contents that was overwritten by the `deletionLSN`, and the delete flag is cleared. The record size is removed from the reservedSpaceTable and subtracted from markedSpace.

Other space-reserving operations are undone using their space-consuming counterparts as during forward processing. The only difference is that these space-consuming undo operations always consume reserved space instead of regular free space, even if there is no entry in the reservedSpaceTable for the transaction any more. This is correct because the transaction *must* have reserved the space it is now reclaiming.

Undo of space-consuming operations has to take into account that a rollback is never undone, so undo of a space consumption is not always a space reservation:

Record insertions are undone by deleting them without reserving space. A redo-only deletion compensation log record without before image is written.

Other space-consuming operations during undo are treated in the same way as their space-reserving counterparts in forward processing, with the exception that not the full consumed amount is reserved during undo, but only the amount of reserved space they consumed, as recorded in the log record (refer to Forward Processing for space-consuming operations above). If the space-consuming operation did not use any reserved space, then the reserved space table is not modified at all.

7.5.5 Segment Metadata: Free Space Inventory

The segments maintain a persistent Free Space Inventory (FSI) for their pages, see Section 5.6.5. FSI information needs to be precise not only for performance reasons, but also for correctness, as it is the base for decisions about overwriting or ignoring the contents of certain pages (see Section 7.5.6). Hence, proper recovery is necessary.

We will first present possible approaches to FSI recovery, and discuss their pros and cons. Then we will describe how FSI recovery is done in Natix.

Existing Approaches to FSI Recovery

We will first outline how FSI is dealt with in the classical ARIES line of papers. Then we describe FSI recovery as an application of object-level recovery techniques as presented in 7.2.3, treating data page and FSI update pairs as subtransactions.

FSI recovery in classic ARIES The approach followed by Mohan et al. [65, 66] is to treat the FSI as regular data, by logging updates on FSI pages and performing analysis, redo and undo during restart and transaction recovery. Their motivation is “to avoid special handling of the recovery of FSIPs during redo and undo, and also to provide recovery independence” [66].

Actually, due to fine-grained locking on data pages, and due to its derived nature, FSI data is subtly different from ordinary data or indices in terms of recovery.

Fine-grained locking allows for several transactions to modify records on the same data page, and some of them may roll back while others do not. In this case, undo log records with before images are not a correct source of undo information for FSI pages.

Consider the following example: A transaction T_1 consumes 15% of a page’s space, so that the free space on the page changes from 55% free to 40% free. This causes the FSI information for that page to change from “more than 50% free” to “more than 25% free”. Another transaction T_2 consumes another 10%, but does not need to change the FSI data. Now T_1 rolls back. If it were to change the FSI information back to its original value of “more than 50% free”, this would be incorrect, as the current page state is 45% free.

The correct behaviour for forward processing is achieved by logging FSI updates as redo-only log records *after* logging the data page update.

During undo the FSI value for the data page is recalculated to see whether an FSI update is necessary, and if yes, an FSI compensation log record is written *before* the data page update. This delicate log ordering dependency is needed to ensure that the update to the FSI and the data page update are treated as an atomic action also during restart recovery [65]. This means that undo processing requires special code for FSI processing. Although the authors of the method themselves consider special recovery code for free space management to be undesirable, they can only avoid it during redo, introducing special FSI undo code.

FSI recovery using L1 operations We will now propose a straightforward way to also avoid special undo code for the method above by applying object-level recovery techniques.

We will then discuss the shortcomings of this approach.

By treating the combination of a data page and an FSI update as a single L1 operation with logical undo, the desired result is obtained as well: The data and metadata updates are atomic, and undo does not need to be the inverse operation of the original action.

This could be implemented by always (during forward and undo processing) writing an undo-redo FSI log record with before image first (instead of just a redo-only FSI record), followed by an L1 subcommit log record describing the data page update.

In case restart recovery sees the FSI update, but does not see the subcommit record, the old FSI value is restored. In contrast to the example for forward processing in the previous section, the before image in the FSI log record is always correct: If a crash occurs, there can be no later committed updates on the page, otherwise the log would have been flushed and both the FSI *and* the data page log record would be visible to restart recovery. No other log records can be written for the same data page between the FSI log record and the data page log record, because the data page latch is held until both log records are written (refer to Section 5.6.5). Hence, the FSI before image is always correct for restart undo when the crash occurs before the L1 subcommit log record is written.

If the subcommit log record is encountered by restart undo, it triggers logical undo which undoes the data page update and determines the correct FSI value even in the presence of other committed updates on the page.

Discussion The problem with this L1 operation approach to FSI recovery, and also with the special undo code from Mohan et al. [65] as described above, lies in the fact that the FSI log record has to be written before the data page log record, which forces the order of performing and logging the data page operation, as follows. To determine the FSI value that should be logged, the operation is performed and the new FSI value calculated. Then the FSI log record is written, and afterwards the data page log record is written. This disallows certain optimizations, and introduces additional dependencies between the page interpreter and the segment modules.

For example, we sometimes would like to write the log record before actually performing an operation. In that case we can write the before image using the current contents of the record, avoiding at least one copy operation. Using the asynchroneous interface of the log manager (Section 7.4.2), we can even surround the operation execution with the log record's creation, writing the before image before performing the operation and the after image after performing the operation to the same log record. Avoiding data copying like this is not possible if we need to write FSI log records before the operation, unless the page interpreters provide methods to predict the free space value change caused by an operation. Then, it would be possible to first determine whether to log an FSI change and logging it before performing the operation. But, depending on implementation, it is very much possible that predicting an exact change of free space is as performance intensive as actually performing the operation, e.g. when data compression is involved. CPU usage would double in this case.

We now dicuss the architectural consequences of the approaches above and see that in addition to lowering performance, the approaches above would make the system more complex.

Natix's storage system architecture (Section 5.1) is designed in a way that keeps the modules for intra-page data structures, the page interpreters, separate from the modules for inter-page data structures, the segments. In addition, segments in general do not need to know if, and how, intra-page data structures are different for recoverable and non-recoverable segments.

FSI management belongs to the segment module, as segments use the FSI to find a page that can hold a certain amount of data. To implement the approach above, we either have to add some FSI code to the pages, distributing the knowledge about FSI management over two modules, or we need to add code to all the segments, calling the page interpreters differently in the case of undo.

This is even more undesirable from a software engineering point of view than adding special recovery code for free space management, as the special undo code has to be added to *every type of segment and/or page* that is supported by the system. This results in two very similar, but subtly different pieces of code in all classes that implement persistent multi-page data structures, whereas special FSI recovery code would only need to be added to the segment base class.

Another argument against treating FSI updates as simple data updates originates in the fact that FSI information is critical to system performance (that's why it exists in the first place). As a consequence, the FSI information is cached (see Section 5.6.5, and McAuliffe et al. [55]). If maintenance of this cache is desired during recovery, we need to treat the FSI updates in a special way to update the main-memory segment data structures.

In the last paragraphs, we have argued that special FSI recovery code is necessary, and why the approaches using a special order for FSI log record are not desirable for our system. We will now elaborate on Natix's approach.

Natix FSI Recovery

It turns out that, given Natix's way of handling the Free Space Inventory, recovery handling fits in quite naturally. Recall from the storage system architecture (Section 5.1), that the segments control FSI management. When unfixing a page, i.e. after an update operation, the segment checks whether the FSI value has changed, and if so, records the change in the segments FSI cache and in the FSI segment. This leads to the key point of Natix's FSI recovery: All accesses to pages follow the fetch/do/unfix protocol, not only during forward processing, but also during redo and undo processing.

As a result, FSI maintenance is automatic during undo. Since all operations are undone for loser transactions, both during transaction recovery and restart undo, and pages are unfixed just as during forward processing, the necessary FSI changes are automatically performed.

Unfortunately this does not mean we can do away with FSI log records altogether: The remaining problem is redo, as not all operations of a transaction are redone. We need to write FSI log records for those cases where the correct page version was already on disk, so the data page update is not redone. We need FSI update log records to perform redo for the FSI pages. We also want to write FSI log records for R2 recovery, when only the FSI page was damaged on disk and needs recovery, but the associated data pages do not.

We now describe for the different modes of operation the additional steps which must be taken to ensure proper FSI recovery in Natix.

Forward Processing During forward processing, all modifications to FSI pages are logged as redo-only log records. This is done by just using logging FSI page interpreters, so neither segments for concrete data types nor FSI segments need special FSI recovery code.

The normal fetch/do/unfix protocol of the segments is used, which automatically updates the FSI during unfix. As a result, the data update log record always *precedes* the FSI update for that page.

To facilitate redo recovery (see below), the segment stores FSI log records' LSNs also as the data pages' `pageLSN`s ($*$). This guarantees that if a page is flushed to disk, its associated free space information is available in the stable log. The data page update and FSI log records are created during the same latch on the data page, so it is not possible that the data page is stored on disk, and the FSI log record is not.

Undo Processing Undo processing also uses the fetch/do/unfix protocol, thus automatically updating the FSI and writing log records. Note that in contrast to Mohan et al. [65], the order in which data and FSI log record are created is the same during forward and undo processing: The data log record always precedes the FSI log record.

The recovery manager updates the transaction control block during undo, recording which LSN is currently undone and which LSN needs undo next. This information is used by the log manager to automatically treat the log records generated during undo as CLRs, and maintaining a proper `nextUndoLSN` chain (Section 7.4.2).

If an FSI log record is generated, its `nextUndoLSN` field must point to the same LSN as the `nextUndoLSN` field of the data page CLR. As usual, the proper LSN value is taken from the `undoLSN` field of the transaction control block. The FSI log record in our implementation is written when the page is `unfix()`ed. The recovery manager only calls `unfix()` when the current log record affects a different page from the last log record (refer to Figure 7.22). Special care must be taken to avoid using a wrong `nextUndoLSN` value because the transaction control block's `undoLSN` value is modified by both the recovery manager's undo loop and the FSI log record creation.

Modification of `undoLSN` is done in the recovery manager using the `prepareLxUndo()` calls. We make sure the proper value is used by performing the prepareUndo calls which modify the `undoLSN` *after* pages are unfixed in the undo loop (again, refer to Figure 7.22). This also causes an FSI log record during undo to be written only once if a sequence of undos modifies the FSI value of the same page several times.

In our protocol, the segment is used in the same way as during forward processing. This has the beneficial side-effect that any main-memory cache structures for FSI management used by the segment are automatically updated during undo.

Checkpointing Nothing is changed to regular FSI management during checkpointing.

Analysis When encountering FSI update log records, the affected FSI pages are included in the dirty page map as usual.

Redo Processing The FSI update log records are used to redo FSI information if the affected FSI page is dirty, as in case of regular data pages.

For all pages for which no redo is necessary because the data updates have already been flushed to disk, proper FSI values are restored because their FSI values are contained in a log record that was write-ahead-logged prior to flushing that page (see ($*$) above in Forward Processing).

For all pages where redo is necessary, we may or may not have proper FSI log records. There may have been a system crash after logging the data page update, but before logging the FSI update. The FSI update seems lost at first glance, but we will now explain why the FSI page is updated properlyanyway:

For those pages redo is performed again using the regular `fetch()`/do/`unfix()` protocol. When the page is unfixed, the regular `unfix()` code automatically calculates the correct FSI value and, if it changed due to the operation, the FSI segment is updated accordingly. So the FSI page is updated properly, even if no FSI log record was found. As an additional benefit, `unfix()` processing also updates the main memory FSI cache for the segment. If a data page operation is followed by a proper FSI log record, no harm is done, as the FSI redo is now just an identity operation.

To avoid having `unfix()` logging all the FSI operations a second time and to avoid writing any log records during redo, we suspend FSI log record generation during the redo phase.

We still need FSI log records, even if it seems that their contents can be derived from the data page updates. In case the data page was flushed to disk, but the FSI page was not, we need a log record causing the recovery manager to redo the FSI updates. This points us to the only remaining problem with the approach described so far. What happens if the system crashes again, after the FSI update was deduced from the data page update, and the data page was flushed, but the FSI page was not? If there never was a log record for the FSI update, it may be lost because during the next restart, the data page update is not redone, and so the unfix operation cannot perform FSI maintenance.

We can fix this by writing the missing FSI log records after the redo phase: We augment the dirty page table with a field $lastFSI$ for each dirty page. During redo, we update this field every time we find a FSI update log record for that page. After the redo phase, we scan the dirty page table and check whether the current FSI value of the page, as stored in the FSI segment, was encountered as FSI log record. If not, the FSI log record was missing and is now appended to the log. After completing the scan, the system takes a checkpoint and flushes the log, and the previously missing FSI log records are now durable.

As a last caveat, we may not flush any pages that have missing FSI log records before redo is complete, because then we have not yet written the missing FSI updates. To require page writes during redo, the buffer must be smaller than it was before the system crash. This means that in addition to crashing the system, its configuration was changed before restarting it, which is not a good idea anyway.

7.5.6 Segment Metadata: Allocation and Deallocation of Pages

New, unused pages need to be formatted (Section 5.4). Newly allocated pages are not loaded from disk, as they are overwritten during format anyway (Avoid read). Symmetrically, pages that are empty are deallocated (marked as unformatted) and removed from the buffer manager to avoid writing them back to disk (avoid write). These optimizations are significant, as they can reduce I/O for update intensive operations by a factor of two[6]!

Recovery issues when avoiding I/O

1. The free space management info also reflects uncommitted record deletions. This means that a page can be marked empty because all records on it have been deleted, but since the transaction that deleted the records may roll back, it is possible that there is information about reserved space stored on the page which may not be discarded.

2. On the other hand, if the page was deallocated and not written back to disk, and is reallocated later, then the on-disk version of the page *must not* be used, as it still may contain some deleted records.

3. Further issues with recovery while avoiding I/O for newly allocated pages are related to redo. When a page in a partition was never loaded, the `pageLSN` value on the disk is meaningless and may contain garbage values that prevent log records from being redone. While this is avoidable by initializing new partitions' pages with Null `pageLSN`s, this simply substitutes I/Os when initializing the partition against I/Os when using it later.

4. In addition, initializing partitions does not remedy another issue: PIDs may have several ”lifetimes”. This happens, for example, if a page is deallocated and dropped from the buffer, and later the same page number is reallocated, possibly even to another segment. The partition number component of a PID is also affected, when a whole partition is dropped from the system, and later the same partition number is reused for another partition.

   Thus, a *lifetime* of a PID are those operations in the log starting from and including a format operation and ending at the last operation on the page either before the log ends or before (not including) the next format operation.

   In both cases, during redo there is the danger of applying log records with matching PIDs from several different lifetimes of a page to a page state which is not compatible with the operation. In particular, no log records from earlier lifetimes of the same PID may be applied to the most recent version on disk, because the effect of the operations is undefined and may even cause the system to crash because, for example, when the page was not initialized at all on the new partition.

[6]Without avoiding I/O, in the worst case during a bulkload, each page is read once and written once. When avoiding I/O, reading can be omitted, thus avoiding half of all I/O operations.

The above issues related to redo and uninitialized `pageLSN` values also apply to other state-identifying information on the pages. Examples include the *check bits* as employed by Mohan et al. [62], which are bits on each physical sector of a page used to detect if writes of multi-sector pages were completed. They are toggled before each write operation, and need well-defined initial values as well.

Avoiding I/O in ARIES

Two papers from the ARIES series [62, 65] provide techniques to address the issues raised above.

Space Management using Concurrency Controller Whether or not a page is truly empty, or may still have some uncommitted deleted records, is detected in Mohan et al. [65] using a complicated method that uses the lock manager to record information about reserved space. This solves issues 1 and 2 above.

The price of this technique are additional assumptions and call dependencies between intra-page data structures and the system-wide concurreny controller. In particular, now the Redo processing phase needs access to locking information to know whether it needs to load a page or not.

Explicit Formatting In Mohan et al. [62], the authors propose to explicitly log format actions. If a page is not loaded from disk, it must be formatted, and a format log record must be written. During redo, format log records always cause a page to be formatted without regarding its prior contents. This solves issue 3. It also simplifies issue 2 during Redo processing, as now no locking information is required any more to ignore the on-disk contents of a page.

Relevant Partition Lifetime Mohan et al. [62] propose to deal with issue 4 for the partition number[7] case by storing in the first page of each partition the `creationLSN`. This is the LSN of the operation with which the new partition was introduced. Redo is only attempted for log records whose LSN is larger than `creationLSN`. To speed up the comparison, the `creationLSN` is buffered in main memory for each partition. As an alternative, the authors suggest using the analysis pass to determine the `creationLSN` and filter out undesired log records.

Use LSNs to Avoid Corruption Mohan et al. [62] address issue 4 for reallocated page numbers by exploiting the `pageLSN` as during regular recovery. The `pageLSN` is monotonic also across lifetimes of PIDs, and already guarantees that log records are only applied if they are not yet contained in the page. Mohan et al. [62] propose to simply redo all operations on a page that have not been stored on disk, *including* all operations from previous lifetimes, and all format operations.

Note that this wastes resources by scanning the log far back into previous lifetimes of a page, and by redoing operations that are going to be overwritten by a later format operation.

[7]In ARIES, partitions are called *fi les*, and partition numbers are *fi le names*.

The last item above contains the assumption that page writes are atomic. If they are not, the sector containing the `pageLSN` may still be one from a previous write, while the remaining sectors already have the values according to a more recent write. Hence, the `pageLSN` indicates that a log record may be redone, while in fact the page is in a corrupt state. Mohan et al. [62] describe a mechanism to detect such partial writes, which we do not detail here. However, this mechanism depends on certain knowledge about the previous on-disk state, every time a page is written, except before the very first write. For the last item discussed above, this means that not only regular redo is performed for old lifetimes of a page if a partial write occured. In addition, even R2 recovery from a backup copy may be required, only to recover some check bits, while the redone changes will afterwards be obliterated with a format operation!

Avoiding I/O in Natix

Goals Our goals for avoidance of I/O during page allocation/deallocation in Natix are (1) to avoid additional assumptions and architectural dependencies between intra-page data structures and the system-wide concurreny controller, (2) to prevent redo of changes for old lifetimes of pages, (3) to avoid performing R2 recovery for old lifetimes, (4) to have the same mechanism for the very first allocation of a page and for subsequent reallocations. The approaches cited above have different mechanisms, making the code unnecessarily complex.

Differences to ARIES This is achieved by the following modifications to the techniques explained above:

1. The FSI values for *formatted, but empty* pages, for *allocated, being unformatted*, and *deallocated, free* pages are pairwise different (Section 5.6.5).
2. Pages are not deallocated unless all transactions with pending delete operations on a page have committed.
3. Deallocations are also logged.
4. The LSN of format operations on pages is used not only as `pageLSN` for the page, but also as `redoLSN`.

We now elaborate on Natix's method, before concluding the subsection with an explanation why this method reaches our goals while addressing the raised issues.

Detailed Description As a result of record-level locking, undo of a page allocation is not necessarily a page deallocation because another transaction might have added records to the newly allocated page, in which case the page needs to stay allocated.

This points to the need for page allocations to be implemented as L1 operations, since undo is not necessarily the inverse of the original action, and since a page allocation needs to be atomic in that a page may not be marked formatted when it is not.

The avoidance of unnecessary I/O is not always possible with this scheme. Consider the situation in which a page contains only records that are marked deleted, but the page is not deallocated because its contents may be needed for undo processing. Now further assume that the page is flushed to disk, and all transactions that deleted records on it commit. If the page is reused later, it will be loaded into memory although it does not contain meaningful data.

However, it is likely that the page will be reused before it is flushed to disk, in which case no unnecessary I/O is performed. As an additional measure, when the page interpreter is notified that the page is going to be written, it can perform reserved space collection. If the page is completely empty afterwards, the page interpreter sets its `isInvalid()` flag, and notifies the segment, which marks the page deallocated in the FSI. The buffer manager detects, after calling `prepareWrite()`, that the page is now invalid and needs not to be written.

Forward Processing

Allocation As mentioned above, page allocation is implemented as an L1 operation:

> After the free space management routines of a segment have found a page which is marked free in the FSI, that page is marked allocated and a FSI log record describing the allocation is written. This log record is called the *allocation log record*, and is a regular FSI log record with kind `FSIPAGE_UPDATE`. The FSI value 1 used to mark the page as *allocated, being formatted* prohibits other transactions to try to get a latch on the page (as explained in Section 5.6).
>
> A buffer frame is assigned to the PID, but the page is not loaded from disk. The buffer frame is formatted by calling the page interpreter's format routine. Then a *format log record* is written by the segment (log record kind `PAGE_FORMAT`), which is also the L1 subcommit log record for the L1 allocation operation. Its `nextUndoLSN` points to the transaction's log record preceding the allocation log record.
>
> The format log record does not cause the FSI to be updated to "formatted" state. Instead, the `recordedFreeSpaceInfo()` field in the page interpreter is set to an invalid value, thus forcing an FSI update when the page is unfixed. Usually, this FSI update will already include the first record insertion on the new page, which occurs directly after the format operation, while the page is still fixed.
>
> The format log record's LSN is recorded as `pageLSN` of the newly formatted page, and the `redoLSN` for that page is also set to the format log record's LSN.

Deallocation If no more regular records and records that are marked deleted are present on a page, it is evicted from the buffer manager, and this is also logged using an *evict log record*. An evict log record is a log record of kind `BUFFERMGR` with operation kind `REC_OPER_BUFFERMGR_EVICT`.

> Finally, the page is marked deallocated by setting the page's FSI state, which is logged using a log record of kind `FSIPAGE_UPDATE` which we call *deallocation log record*.

Neither log record contains the page's contents as before image, as we only allow to unformat truly empty pages. The deallocation log record also does not update the `pageLSN`, as the page is evicted from the buffer anyway.

Analysis

Allocation A format record causes the data page to be entered into the dirty pages map if it is not already present, just as a regular page-level update log record. The format log record's LSN is used as `redoLSN` in the dirty pages map. The allocation log record causes the FSI page to be included in the dirty pages map.

Deallocation If an evict log record is encountered during analysis, the page can safely be dropped from the dirty pages table, as any redone operations would be thrown away during redo anyway (See below).

Redo Processing

Allocation As during forward processing, we do not load a page before formatting it to avoid unnecessary I/O. This means we do not have a `pageLSN` to compare and check whether the update is already present during redo. This check is performed in the segment's `redoLogRecord()` function. For regular page-level update log records, it loads a page and verifies that the given log record's `LSN` is larger than the `pageLSN`.

To avoid I/O during redo when encountering a format log record, the segment does not use the `fetch()` call to acquire a main memory address for the page. Instead, it uses the `getFrame()` call, which avoids I/O. After receiving the main memory address, it does not compare the `pageLSN` — which may have any arbitrary value because an arbitrary frame of the buffer manager is used — to the format log record's LSN.

If the `redoLSN` in the diry page map says the format can be redone, redo is performed, without regarding the on-disk version of the page. Therefore, the algorithm is robust against garbage on on-disk pages that look like a valid LSN and would lead to incorrect redo, no matter if the garbage stems from an uninitialized page or a partial page write.

Since the `redoLSN` is reset during `format()` in forward processing, any log entries that occur before the format log record are not redone; those updates would be overwritten by the format operation anyway.

This method sometimes redoes updates that may have been flushed to disk already: Consider a page that first has been formatted, then updated by some operations and then flushed to disk. Now a crash occurs and restart recovery takes place. If the analysis phase results in the format log record's LSN as `redoLSN`, then the page's on-disk version is not accessed during redo. Instead, `getFrame()` is used to get a new buffer frame, which is formatted and all the update operations following the format are redone although they were already on disk.

Occurences of this undesirable, though not incorrect, effect are reduced by logging page flushes (see 7.4.6). On encountering a logged page flush during analysis, the page is dropped from the dirty page map. During redo, the format and update operations are not repeated, the loaded version is used instead. Already stored updates may still be repeated, but only if a crash occurs between the flush operation of the page and the writing of the page's flush log record. Even then, this is only hurting performance if redo of the updates takes longer than loading the page (which can take as long as several thousand machine instructions).

Redo of the allocation record is unnecessary, since the subsequent redo of the format also sets the FSI value of the page to its correct state and would overwrite the redone allocation record's update anyway.

Deallocation When redoing the last record deletion on a page, updating the FSI value and possibly evicting the page from the buffer are automatically performed because the page interpreter marks itself invalid when it is completely empty. This is only necessary when the last delete operation was logged, but the subsequent evict and deallocation were not. Otherwise, the page is not in the dirty page map in the first place.

Undo Processing

Allocation The allocation operation is an L1 operation, whose logical undo operation is the empty operation: As explained above, other transactions may have added records to the page, so deallocating the page may be incorrect.

Only if the L1 allocation operation was not completed before a crash, i.e. the allocated state was recorded in the FSI on stable storage, but the format operation was not, then the allocation is undone.

Regular deallocation of pages proceeds as part of normal undo processing of record insertions: The undo operation for a record insertion is a record deletion. Since the segments use their normal forward processing protocols during undo, we automatically use the regular deallocation routine: If a page is completely empty after a record deletion, the page is deallocated using the deallocation as described under forward processing during unfix.

Deallocation When encountering a deallocation log record during undo, the segment performs the same steps as for allocation during forward processing. This is necessary because the following undo operations (which repopulate the page) assume that the page is formatted and empty to allow undo of deletions on the page.

Note however, that undo of deallocations is rare, as pages are only deallocated by regular transactions when page-level locking is in effect. Otherwise, the pages will always still contain records that are marked deleted.

The only case where deallocations are logged is when a page interpreter detects during `prepareWrite()` that it is completely empty after it performed reserved space collection. In this case, the page is deallocated instead of being written. However, this deallocation is logged by the system transaction, which is never undone (Section 7.3.8).

Evaluation

We reconsider the issues with avoiding I/O and recovery, brought up in the beginning of this subsection, and show how they are addressed by our method.

1. Since Natix never marks a page as reallocatable in the FSI when there is any information on it that may be required later, we know before allocating a page that it is safe to ignore previous contents.

2. Since we explicitly mark pages as deallocated, we know when we may not use the on-disk contents of a page.

3. Since we explicitly log format operations and unconditionally redo them, ignoring previous page contents, there are no ”dangerous” values on pages that might compromise recovery.

4. Since we use the format operation’s LSN as `redoLSN`, we do not need to redo or worry about previous lifetimes of a PID.

Our method also meets the goals we have set in addition to correct operation:

1. Recovery that avoids I/O does not require assumptions or dependencies between modules in addition to those required by recovery that does not avoid I/O.

2. Redo only needs to scan and process log records from the last lifetime of a PID.

3. R2 recovery for prior lifetimes is unnecessary, and there is no need to store a partition’s `creationLSN`.

4. Allocation code is identical for the first and subsequent allocations of a page.

7.6 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 Natix, we relax both of these constraints. This boosts overall performance by reducing log size and increasing concurrency.

Suppose a given record is updated multiple times by the same transaction, which frequently occurs when using the storage layout for XML documents as described in Section 5.7, for example when a subtree is added node-by-node. In many cases, it is desirable to log this composite update operation as one big operation, for example by creating only one log record for the complete subtree insertion. 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 explain how Natix's recovery architecture supports such logging optimizations. We will also elaborate on the concrete implementation for the case of XML data.

7.6.1 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.

A 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*, i.e. regularly written as log records. The remaining architecture and interfaces of the system do not need to be changed.

Subsidiary Logs as Part of the Buffered Log

To retain recoverability, some rules have to be followed. Adhering to them will make the subsidiary logs become part of the log buffer as far as correctness of the recovery process is concerned. Although part of the log buffer is now stored in a different representation, its effects for undo and redo processing are the same. Basically, the rules 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.

Write-Ahead-Logging 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.

Force-At-Commit Likewise, all subsidiary log entries must be published to the regular log manager before the transaction commits, to follow the force-at-commit rule.

Publish-before-Savepoints When the application requests an identifier in the form of a savepointLSN for the system's state to allow partial rollbacks, this savepointLSN must include the operations in the subsidiary logs. For ease of implementation, we publish all the subsidiary logs to the log manager before returning a savepointLSN. Otherwise, we would have to introduce a means to identify the system state which would have to include information about which subsidiary log records belong to the savepoint and which do not. This would require to introduce an interface to specify savepoints to page interpreters, which would complicate the architecture.

Publish-Before-Owner-Change Before a transaction modifies a data page other than the one currently owning the subsidiary log, we require that the the subsidiary log is published to the log manager.

This is necessary to maintain a meaningful `pageLSN` value. In this way, modifications by other transactions show up as an increased `pageLSN` value. Otherwise, there might be modifications to the page that are not reflected in an increased `pageLSN` value. Techniques that depend on proper `pageLSN` values, such as CommitLSN [63], would not be applicable.

Subsidiary-Undo Every time a rollback is performed, all operations in subsidiary logs of the transaction have to be undone. It does not matter to which savepoint the transaction rolls back because the actions in the subsidiary logs have always happened after the last savepoint, due to the Publish-Before-Savepoint rule.

Note that a **Publish-Before-Checkpoint** rule is not necessary. A checkpoint is used to avoid a complete scan of the log during analysis. It records all active transactions at the time of the checkpoint, and it records which dirty pages have log records in the log before the checkpoint. The latter allows to determine how far back in the log redo processing has to start.

However, the entries in subsidiary logs do not need to be found by redo processing. If they are required at any point for redo processing, the subsidiary log will have been flushed anyway, either by a Force-At-Commit or by a Write-Ahead-Logging action.

Implementation Framework

Implementation of the rules is rather straightforward in Natix's architecture. The required modifications are limited to the page interpreter class which is supposed to perform subsidiary logging. It is possible to limit subsidiary logging to specific data types, or even to operate simultaneously with and without subsidiary logging on the same data structure. Apart from the page interpreter, no components in the system are affected, with the exception of additional synchronization for the transaction pending actions list, as explained in the following subsection about synchronization.

Write-Ahead-Logging 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. It first publishes the subsidiary log to the log manager, creating new log records and increasing the `pageLSN`, and then flushes the log up to the new `pageLSN`.

Publish-Before-Commit, Publish-Before-Savepoint To publish the subsidiary logs to the log manager before a commit occurs, all page interpreters that maintain a subsidiary log add a pending action to the transaction control block (see Section 7.4.7). This pending action is executed before the transaction commits, and causes the associated subsidiary log to be published to the log manager. Likewise, publish-before-savepoint is enforced by pending actions.

To avoid the creation of excessively many pending actions objects, just one object is used for every page interpreter that maintains a subsidiary log. These objects are created if a page interpreter begins to use a subsidiary log, and they belong to a class derived from PendingAction. It has refined methods for precommit and savepoint, which publish the entries in the subsidiary log and then remove the object from the pending action list.

Subsidiary-Undo To undo the entries in the subsidiary log in case of a rollback, the `rollback()` method of the derived class for our pending action objects is also refined and performs undo for all subsidiary log entries. There are no objects in the pending actions list that do not require subsidiary undo, no matter to which savepoint we are rolling back, since all subsidiary pending actions are removed from the pending actions list when a savepoint is established.

No compensation log records have to be written for undo of subsidiary log entries. As long as the entries are not published, there are no stable log entries that could survive restart. Entries that do not survive restart cannot be redone more than once — they are not even redone once — and thus do not need compensation log records.

Publish-Before-Owner-Change To detect if a different transaction is accessing the page, the page interpreter must maintain a `loggingTransaction` field, containing the transaction control block of the transaction that currently owns the subsidiary log.

If, before writing a log record, the current transaction detects that it does not own the subsidiary log, it flushes it before logging its own update.

Synchronization

As a result of the rules and implementation framework above, transactions write log records for other transactions when a page is flushed or before a transaction tries to create log records for a page which already has a subsidiary log owned by another transaction.

There are several areas where this causes data structures that were previously accessed by one transaction only to be modified by several transactions concurrently. We will list them together with an appropriate synchronization strategy.

Transaction control blocks Writing log records for other transactions does not cause multithreaded access conflicts for the transaction control block, because access to the critical LSN fields in the transaction control block is serialized by the log manager monitor lock.

Subsidiary log Access to the subsidiary logs itself can be synchronized by the data page latch in the buffer manager and does not cause conflicts either.

Pending action list If a transaction publishes a subsidiary log due to a page flush or an owner change, the transaction must modify the pending actions list of the owner transaction. Since other transactions including the owner transaction itself may be concurrently modifying the pending actions list, we need to introduce additional synchronization. An extra mutex in each transaction control block, serializing access to the pending actions list, is employed.

The last item introduces a new problem, as now deadlocks are possible. When removing a subsidiary log from the pending actions list, two synchronization objects are involved: The data page latch protects the page contents and the subsidiary log, while the pending action mutex protects the pending action list.

Unfortunately, in a naive implementation, these synchronization objects are not always accessed in the same order.

- When a page is flushed or the owner of the subsidiary log changes, a transaction holds the data page latch and tries to obtain the pending actions mutex to remove the page from the list.
- When a transaction is committing, establishing a savepoint, or rolling back, it holds the pending actions mutex while traversing the pending actions list and tries to obtain the data page latches on the pages with subsidiary logs.

This may cause a deadlock, for example if a transaction T_1 is committing while another transaction T_2 tries to publish one of T_1's subsidiary logs.

In such situations, where we need to hold both synchronization objects at once, we avoid deadlocks by forcing the order of requesting them. We never hold the pending actions mutex while requesting a data page latch. Before requesting a data page latch, the pending actions mutex is released. After successfully obtaining the data page latch, the pending action mutex is again locked. While the transaction which is processing the PendingActions list was waiting on the data page latch, some other transaction may have flushed the subsidiary log. Therefore, after reacquiring the pending action mutex, it is checked whether the subsidiary log on the data page is still present. If it is gone, nothing needs to be done, otherwise the log is published as desired.

7.6.2 Application: XML-Page Subsidiary Logging

We 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, either during initial document import or later while a document evolves. The Natix storage format (Section 5.7) usually causes 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, which is in conflict with our goal to do as few representation changes as possible, 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.

Now we show how a subsidiary log inside the XML page interpreters allows to amortize logging costs for subtree insertions, without requiring such a tight coupling of the application to the storage subsystem.

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 a representation for log records before publishing them to the global log.

Using a flag called *fresh* in the node headers on the data page, new nodes/subtrees 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 on 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 the subtree is not traversed when 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 nonsubsidiary 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 7.6.1, but also when a node is modified that has its *fresh* flag not set.

While we have the after-image information for our subsidiary log entries, which is stored in the page contents, the `transactionID` to create a complete log record for the subtrees is still missing. Note that we do not need any before images, as the net effect of subsidiary logged updates is always an insert operation.

The transaction IDs are not stored in the data page's contents. But as explained in the previous section, in page interpreters which do subsidiary logging we need a `loggingTransaction` field anyway. We can use that field to obtain the `transactionID` to be used in the log records.

Subsidiary Logging and L1 Operations

L1 operations, as explained in Section 7.4.4, are logged as a sequence of page-level log records which is terminated by a L1 subcommit log record. There are two apparent problems with the combination of L1 operations and page-level subsidiary logging, as explained in the next two paragraphs. These explanations are followed by the necessary steps to avoid them.

Missing L0 updates With subsidiary logging, it is possible that some of the L0 updates belonging to the L1 operation are stored inside a subsidiary log. In this case, if a subcommit record was written, those updates would enter the global log at some point later in time. Should the transaction abort, the L0 updates which were published after the subcommit record are not skipped by the L1 subcommit record `nextUndoLSN` pointer, and hence may be undone by regular L0 undo and not by logical L1 undo as desired, possibly causing a corrupted database state.

Additional L0 updates It seems that subsidiary logging may also cause some L0 operations that do not belong to an L1 operation to become part of one.

If some L0 operations were only stored in a subsidiary log, and they are published to the global log during the L1 operation due to a page flush operation or an owner change, then they cannot be distinguished from those L0 operations that constitute the L1 operation. During undo, those L0 operations would not be undone because they would be skipped after undo of the L1 operation's subcommit record.

Synchronizing subsidiary and global logs Both of the stated problems point to the need to synchronize the subsidiary logs with the global log if L1 operations are involved. Since subsidiary logging may reorder some log records of a transaction, and recovery of L1 operations strongly depends on log ordering of related L0 records, subsidiary logs must be published before an L1 operation commences and before it is completed. That way, all subsidiary log entries that do not belong to the L1 operation are logged before its begin marker LSN (returned from initiateL1 on the transaction manager), and all L0 operations that are performed on behalf of the L1 operation are logged before its subcommit log record.

Fortunately, `initiateL1()` already takes a savepoint before returning the L1 operation's begin LSN value (refer to Section 7.4.7). During a savepoint, all subsidiary log

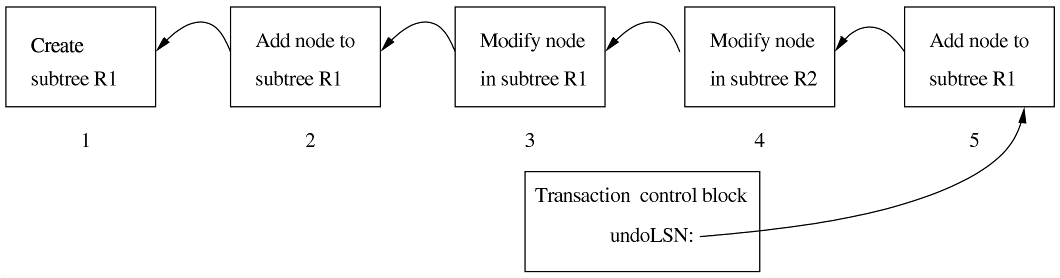
Create subtree R1
Add node to subtree R1
Modify node in subtree R1
Modify node in subtree R2
Add node to subtree R1
1
2
3
4
5
Transaction control block
undoLSN:

Figure 7.23: Log records for an XML update transaction

records are published, so the log of our L1 operation will not be contaminated with earlier L0 operations.

To publish subsidiary logs before an L1 operation is completed, we refine not only the `precommit()` and `savepoint()` methods in our pending action objects, but also refine the `subcommit()` method in the same way. Since the subcommit methods of all pending actions are invoked upon calling `notifyCompleteL1()` on the transaction manager, there are no subsidiary logs when the subcommit log record is written.

Subsidiary log effectiveness It should be pointed out that the efficiency of subsidiary logging is severely hampered if the data structure which employs subsidiary logging is modified exclusively by L1 operations. In such situations, when L1 subcommits happen often, the subsidiary logs are published frequently, reducing the number of updates that accumulate and can be treated more efficiently in the subsidiary log.

If L1 operations are used heavily on a data structure, it has to be carefully decided or measured whether subsidiary logging is increasing performance or not. On the other hand, it should be noted that L1 operations are not necessary if a multi-page data structure is locked in its entirety, as it is frequently the case with XML documents.

In addition, subsidiary logging does not modify the on-disk data structures in any way. Hence, it is possible to switch between regular and subsidiary logging during run-time, even for pages of the same segment, depending on which lock granularity is used.

7.7 Annihilator Undo

Transaction undo often wastes CPU resources because more operations than necessary are executed 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 do not need to be undone when the transaction is aborted, as the record is going to be deleted as a result of transaction rollback anyway. Refer to Figure 7.23 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.

7.7.1 Annihilators

We call undo operations that imply undo of other operations following them in the log *annihilators*. For example, the undo of a record creation like log record 1 in the example above is an annihilator, as it implies undo of all update operations 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 7.4.1 that the `nextUndoLSN` pointer of every log record points to the previous operation of that transaction requiring 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 above for the special case of XML data.

7.7.2 XML Subtree Annihilators

The XML page interpreters augment the stored information for the subtree as follows: In every XML subtree record header, an annihilator LSN and a `transactionID` are stored, containing the LSN and `transactionID` 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 of the current transaction, and if the annihilator was performed by the current transaction using the stored `transactionID`. If this test succeeds, there will

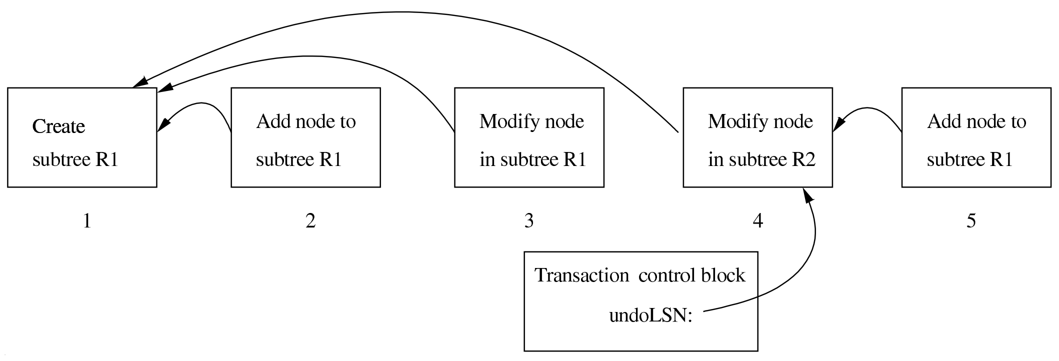
Create subtree R1
Add node to subtree R1
Modify node in subtree R1
Modify node in subtree R2
Add node to subtree R1
1
2
3
4
5
Transaction control block
undoLSN:

Figure 7.24: Undo chaining with check for annihilators

never be a rollback that does not include the annihilator operation. Hence, the update operation can be logged redo-only and will be skipped during undo.

Figure 7.24 shows the resulting undo chain after log records 1–5 from the example in Figure 7.23 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 explicitly announce major impending modifications to a subtree (the shopping cart), causing a before image to be written and the `annihilatorLSN` to be set. The state of the shopping cart before the session can easily be recreated by just one log record undo containing a complete before image, no matter how many single operations were executed in the meantime.

There are alternatives for the storage location of the annihilatorLSN, as reserving space for a whole LSN might be considered too high a cost for the benefits of annihilator undo.

It is possible to store the `annihilatorLSN` in main-memory only, in the state of the page interpreter object. This would disallow annihilator undo for updates that happen after the page has been kicked out of the buffer and was fetched again by the same transaction, which should be an unlikely event.

Another possibily is to store the `annihilatorLSN` for subtrees in the lock control blocks 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.

7.7.3 Page Annihilators

Another possible annihilator is the deallocation of a page. If a page is deallocated during undo, then it is not necessary to undo all updates to that page first.

Page-Level Locking

If page-level locking is employed, the inverse operation of a page allocation is a deallocation. If a transaction allocates a page, formats it and rolls back some time later, the page will be deallocated, and it is unnecessary to undo record inserts and undos on the page first.

Implementation of page annihilators is simple in this case: Here, the `annihilatorLSN` in this case is the LSN of the format log record. The format log record's LSN is already recorded for the page because it is always the `redoLSN` for that page, which is stored in the buffer frame control block. We also store a flag in the page interpreter that is set if the page was freshly formatted and not fetched into the buffer from disk. Analogous to XML subtree annihilators, before logging an update on a freshly formatted page, the page interpreter only needs to check whether the `redoLSN` is greater or equal to the last savepointLSN, and if so, may log updates as redo-only.

As a result, undoing a large-volume bulkload transaction, for example one that loaded a large document, would only be a matter of marking the new allocated pages unused. The FSI pages that record allocation state are usually in the buffer. This means that undo of importing documents or document collections which are much larger than the buffer would not require any I/O at all.

Unfortunately, this only works if page-level locking is in effect. Otherwise, a page that was formatted by some transaction may also contain records of other transactions, even before the formatting transaction terminates. That is why the inverse operation of allocation is not deallocation in Natix (refer to Section 7.5.6), but the null operation. Pages are only deallocated if the last record is deleted.

Record-Level Locking

When record-level locking is employed, we can still use the `redoLSN` as `annihilatorLSN` for the records created by the transaction that formatted the page. We only have to define the logical inverse of page allocation as ”delete all records created by the current transaction”. This would cause the undo of record creation for those records for which no undo log records were generated because of the annihilator LSN. In addition, if the current transaction was the only one that created records on the page, this will cause the page to be deallocated as desired.

The only problem is to determine which records on a page were created by the current transaction. The transaction that created a record is usually not stored with the record, and doing so would be quite a waste of space.

We solve the problem by maintaining a field in the page interpreter that allows to determine whether only one transaction has updated the page since it was brought into the buffer or since it was formatted. Similar fields are also maintained by subsidiary logging (see Section 7.6), and selective restart (see Section 7.8).

Using this information during record creation, we can mark records that were created by the same transaction that also formatted the page. We use a flag in the slot information to do this. If the `redoLSN` is greater or equal to the savepointLSN, we can write redo-only log records after creating a marked record.

When a transaction encounters a deallocation log record, it then may delete all records that have the flag described above.

Unfortunately, this makes it necessary to bring those pages into memory to check whether there are marked records or not. While this technique is advantageous in comparison with undo processing without annihilators because it needs to process fewer log records, it does not meet our goal to avoid data page I/O by just marking the pages deallocated in the FSI.

Avoiding I/O

There are three avenues of approach to use page annihilators and to avoid I/O of annihilated pages:

Exclusively locking new pages When using a multi-granularity locking method that includes pages as one level of granularity, it would be possible to exclusively lock newly allocated pages, so that no other transaction would be able to use the page until the allocating transaction has terminated. This would allow the inverse of page allocation to be deallocation, and allow us to use page annihilators as with page-level locking.

Note that this does not mean to use page-level locking all the time, but only for those pages that were allocated by a transaction.

Defer FSI update Instead of deferring access to a freshly formatted data page by locking it, it is also possible to defer FSI updates for newly allocated pages until transaction termination.

Page allocation consists of several steps in Natix (refer to Section 7.5.6). After writing an FSI value marking the page ”allocated but unformatted”, the page is formatted, and when the page is first unfixed after it is formatted, the FSI is updated to the final value.

After the first FSI update which marks the page ”allocated but unformatted” no other transaction will use the page until the second FSI update. This is because any transaction attempt to access such a page would be blocked by the data page latch which is held by the allocating transaction. Hence, the FSI search ignores pages which are marked ”allocated but unformatted” (Section 5.6.5).

Instead of updating the FSI after the first unfix, Natix can add a `precommit()` pending action to the transaction control block, causing the FSI update to be written before commit. This will have the same effect as a page-level exclusive lock on the page, i.e. prohibit updates on new pages by other transactions, but without blocking the other transactions.

Again, inverse of page allocation is deallocation in such an environment, and the page-level locking page annihilator technique can be used.

Store conflict pages in transaction control block The third method does not prohibit other transactions from accessing the newly allocated page until the allocating transaction terminates.

The page interpreters prepare for page level annihilators as described above for record-level locking, marking records that were created by the allocating transaction and writing redo-only log records for them.

By default, it is assumed that a newly allocated page was not accessed by another transaction. Hence, pages are deallocated by marking them deallocated in the FSI.

Only if a different transaction also adds records to the page, the affected PID is stored in a conflicting page set in the transaction control block. The inverse operation for page allocation checks whether the page is a conflicting page, and if so, fetches the page and removes only the marked records. If not, the default action of deallocating the page without fetching it is invoked.

AnnihilatorLSN Storage Location

As above with XML subtree annihilatorLSNs in the page interpreter, since the `redoLSN` is only remembered until the page is dropped from the buffer, annihilator undo can only be performed for records that were created until the page is written back to disk. Otherwise, the formatLSN would have to be recorded on the page's contents. This may or may not be acceptable for certain applications. By default, Natix only uses the `redoLSN` because in that case we can reuse information that is maintained anyway by the system and thus incurs lower cost for exploiting the annihilator technique.

7.8 Selective Processing During Restart

The ARIES protocol is designed around the redo-history paradigm, meaning that the complete state of the buffered 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.

7.8.1 Selective Redo in ARIES/RRH

There exists an extension to ARIES, called ARIES-RRH [69], that addresses this problem. Here, for pages that are updated with coarse-granularity locking, a special page-level-locking flag is 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 R2 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 each time. Although Mohan et al. [69] also describe a relaxed version of ARIES/RRH that in some situations allows selective redo and undo for fine-granulartiy locking. However, it 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.

7.8.2 Constraint for Correctness of Selective Redo

For selective redo in an ARIES-based recovery environment to work, it is not *necessary* that page-level locking is in effect for the affected pages during forward processing. Instead, it is *suffi cient* that uncommitted updates of at most one transaction are present on affected dirty pages.

Uncommitted updates of only one transaction per page are *enforced* by page-level locking, but *may* also happen with finer lock granularities. Especially when only a few large records are on each page, as is the case with Natix XML storage, it is very likely even with record-level locking that only one transaction has uncommitted updates on a dirty page.

Also note that *transaction* in this context includes Natix L1 operations, which are atomic but non-durable subtransactions of the regular transactions. Since such transac-

	Previous dirty flag	Old `updaterLSN`	New updaterLSN
1	false	any	$transactionLSN(T)$
2	true	$transactionLSN(T)$	$transactionLSN(T)$
3	true	$< oldestUndoLSN$	$transactionLSN(T)$
4	true	$>= oldestUndoLSN$	null
5	true	null	null

Table 7.5: Maintenance of updaterLSN

tions are not undone using the page-level physical inverses, undoing them by not redoing their page-level actions is not correct.

Since we do not treat L1 operations as explicit transactions with a `transactionID`, the remainder of this section can only be applied to segments without L1 operations. The XML segment is one of them, as are regular slotted page segments.

7.8.3 Selective Processing of Regular Log Records

If there are only updates of one loser transaction on a page, we can skip redo of those operations on the page that are going to be undone anyway. For log records that are not compensation log records, the condition that only one loser transaction has updates on the page is sufficient to avoid redo. How redo of such log records is avoided in Natix is described below. Later, in Section 7.8.4 we will show how compensation log records are treated.

To efficiently determine whether one or more than one uncomitted transactions have updated a dirty page, we add an `updaterLSN` field to the main-memory page interpreter. If only one uncommitted transaction has updated the page, the `updaterLSN` field contains that transaction's transactionLSN. Otherwise, the `updaterLSN` field is null.

We will first describe how the field is maintained during forward operation and analysis, and then describe its usage during restart recovery.

Maintenance of the `updaterLSN` field

In case of record-level locking, when transaction T updates a page, its `updaterLSN` is maintained as shown in Table 7.5. The first case describes what happens when a clean page is updated for the first time: Only T has updates on the page. If T updates the page again, nothing is changed (case 2). If the page only contains updates of one transaction that already terminated, then T is the only transaction with uncommitted updates on the page (case 3). If uncommitted updates by one other transaction may be present (case 4), the `updaterLSN` is reset. If there are already several transactions with uncommitted updates (case 5), we leave the `updaterLSN` field alone. Of course, case 2 takes precedence over case 4.

In case of page level-locking, we can always set the `updaterLSN` to $transactionLSN(T)$, since all other transactions that accessed the same page must have terminated.

During checkpoints, the `updaterLSN` for every page is included in the dirty page checkpoint log records.

The dirty pages table that is built during the analysis phase contains the `updaterLSN` in addition to the `redoLSN`. UpdaterLSN values of updates before the checkpoint are taken from the dirty page log records. Updates following the checkpoint cause the `updaterLSN` in the dirty page table to be maintained as shown in Table 7.5.

The dirty page table can now be used during restart redo to avoid redo on pages with only updates of one loser transaction.

Redo Based on updaterLSN

When a page update by a loser transaction T is encountered during redo, the recovery manager operates as usual. It first checks the dirty page table to see if the update is already contained in the stable version of the page, based on the `redoLSN`. If the `redoLSN` indicates that redo may be necessary, the decision whether to actually redo the update is also based on the page's `updaterLSN` from the dirty page table.

- If the `updaterLSN` is *not* equal to $transactionLSN(T)$, the update is redone as usual.

  This guarantees necessary redo on pages with updates of several transactions. It has to be noted that ”redo as usual” implies checking of the actual `pageLSN` whether the update is already present or not.

- If the `updaterLSN` is equal to $transactionLSN(T)$, the update is *not* redone.

  We do not redo the update because the update by loser transaction T would later be undone anyway.

Selective Processing During Undo

Regular undo processing in ARIES relies on the assumption that history has been repeated and all updates visible in the log are present in the buffered database state.

With selective processing during redo, this assumption no longer holds. Some records may not have been redone, and as a result their respective undo operations must not be performed.

Hence, the undo pass has to be modified to check whether the update described by a certain log record is present on the page by comparing its LSN to the `pageLSN`. Only if the update is present, i.e. the `pageLSN` is greater or equal to the log record's LSN, undo has to be performed. Otherwise, the log record's operation does not need undo, as it was not performed during restart redo.

In any case, a compensation log record is written. Although a CLR for an undo operation that has not been performed is not needed during restart recovery, R2 recovery is simpler when all redo log records for a given page can be redone unconditionally. If R2 recovery redoes all updates, then it also needs to redo all undos. Hence, CLRs must also be written if the undo was not performed.

7.8.4 Selective Processing of Compensation Log Records

Compensation Log Records need a special treatment during restart redo, as they may need to be redone even if the page contains only updates of exactly one loser transaction. If the update undone by the CLR was already written to disk before the crash, then we may not ignore the CLR, even if it belongs to the transaction with the page's updaterLSN. On the other hand, if the original update was not written, and ignored by the redo phase as described above, then the CLR may not be redone, otherwise the page would get corrupted.

ARIES/RRH uses an `undoneLSN` field in the log record to determine during redo whether a CLR's original update is present or not. We avoid increasing the log record header by such a field because this would mean to generate significantly more log only to benefit in a special case of restart recovery.

In the following, we will describe how to use a flag that specifies whether a CLR is only necessary for R2 recovery or also for restart recovery.

CLRs with Synchroneous Writes

Suppose that during forward processing, we always know for every LSN whether the result of the associated operation was already written to disk or not. Such information could be derived from the `redoLSN` of the affected page because all updates with an LSN smaller than the `redoLSN` have been written to disk. We will also suppose that no updates occur on a page while it is being written to disk, so that all operations with an LSN higher or equal to the `redoLSN` definitely have *not* been written to disk.

In this case, we can already determine *during forward processing* if a CLR will be required for R2 recovery only or if it may also have to be redone during restart. During undo, when writing compensation log records for a log record with LSN u, we distinguish two cases:

- If the page was written after the original operation and before the inverse operation was applied (`redoLSN`$> u$), then the CLR may be necessary during restart redo to undo the operation.

  In contrast to ARIES/RRH, we do not need to determine during redo whether the original operation was written to disk or not, as we know already during forward processing that it was written. By setting a flag in the CLR header, we instruct the redo phase to perform ordinary redo processing of this record, even if the `updaterLSN` of the page equals the `transactionLSN` of the CLR. The only thing we have to check during redo is whether the inverse operation was also already written to disk before the crash. But this is already part of the ordinary ARIES redo procedure: If the `pageLSN` is greater than or equal to the CLR's LSN, the inverse operation is already present and nothing has to be done. Otherwise, we redo the CLR.

- If the page was *not* written between the execution of the original operation and the undo operation (`redoLSN`$<= u$), then the CLR is only necessary for R2 recovery.

  There will never be an on-disk version of the page that only contains the original operation and not its inverse. The disk version either contains the effects of both operations, if the page was written between the undo operation and the crash, or it contains none of them, if the page was never written after the original update. In both cases, the CLR will never be needed for restart redo, since it is only necessary when the original is present on disk, but the inverse operation is not. We can set a flag in the log record header accordingly, saying that this record only needs to be redone during R2 recovery. This flag consumes much less space than a full-blown `undoneLSN`.

The flag may be used to avoid redo of a CLR only if the `updaterLSN` of the page is not null. Otherwise, more than one transaction may have CLRs for that page, and the regular ARIES redo procedure is employed, i.e. the CLR is redone as usual.

A drawback of this method is that certain knowledge about the on-disk state of a page is required during forward processing. This disallows modification of a page while there is a flush operation pending for the page. In many cases, this is not a problem, as pages are latched in shared mode while they are flushed to disk. Hence, updates to pages with pending flushes are prohibited anyway.

CLRs on Hot Pages

For *hot pages* that are updated very frequently, however, it is sometimes desirable to hold the shared latch only for the time necessary to make a memory copy of the page, and then doing the disk write operation asynchroneously from the memory copy. Normal update processing can resume while the page is being flushed. Since the flush operation is asynchroneous, it is possible that the `redoLSN` field does *not yet* contain the current on-disk `pageLSN`, but a previously written version of the page. In this case, it is not possible to use the `redoLSN` to determine whether CLRs can be marked media-recovery only.

To avoid having only very stale versions of hot pages on disk, which slows down restart recovery, they have to be written to disk every now and then to advance their `redoLSN` value. The pages' higher availability for new updates compensates for the cost of making an extra memory copy instead of writing them directly from the buffer.

One way of retaining selective redo also for hot pages would be to synchronize access to the `redoLSN` field, and possibly delaying writing of CLRs until a page write is complete. This would be complicated to implement and would introduce a dependency from the page interpreter writing the log records to the buffer manager that synchronizes access to the buffer. We therefore consider this approach unsuitable.

Natix employs the selective redo technique without `undoneLSN` field in log records also on hot pages while avoiding stale disk versions of those pages, as follows.

Instead of copying the pages to another memory location and writing from there, a redo-only log record with a full page image is created and the page's `redoLSN` is set to that log record's LSN. This means that the page is ”flushed to the log”, and not to its original location. Redo only needs to start at the full image `redoLSN`, as the full image contains all previous updates.

With this method, the treatment of CLRs remains as described above. Should a full image log record precede a CLR, the full image's LSN is used as `redoLSN` to determine during forward processing whether the CLR may be necessary for restart redo or only for

			Recovery Method		
Collection	Format		No Recovery	Regular	Subsidiary
Bioml	Clustered	Elapsed Time	150.98	186.39	165.7
		CPU Time	136.33	148.19	140.63
		CPU%	90%	80%	85%
		Log Size	—	83484	29696
	Single	Elapsed Time	253.41	675.75	408.38
		CPU Time	175.97	545.95	191.71
		CPU%	69%	81%	47%
		Log Size	—	366354	118197
Shakespeare	Clustered	Elapsed Time	29.12	39.91	33.58
		CPU Time	20.31	24.5	21.03
		CPU%	70%	61%	63%
		Log Size	—	28322	9997
	Single	Elapsed Time	50.26	120.03	66.83
		CPU Time	36.14	80.26	41.28
		CPU%	72%	67%	62%
		Log Size	—	125592	40593

Table 7.6: Import Performance With Logging (Times in seconds)

R2 recovery. The ordering of the log records ensures that if the CLR is in the stable log, the full image is also in the log.

The cost of this method is comparably low, since creating the log record and flushing it costs a memory copy and a disk write, like the write-from-memory-copy technique described above. In the log record case, the disk write may even be faster because log writes are sequential. The remaining extra cost, namely storing the page at its original location, only appears when the page is dropped from the buffer, which is by definition rare for a hot page.

The disadvantage of this solution is that it makes the log a bottleneck for writing of hot pages – hot pages of different partitions can not be written in parallel. Depending on the number of hot pages and partitions, this may not always be acceptable.

7.9 Evaluation

We compare the performance of Natix's recovery subsystem with and without Subsidiary Logging when importing documents using different storage formats.

7.9.1 Environment

The environment and data used to measure Natix's performance are identical to those used to gather results about the storage engine (Section 5.8).

Both the Shakespeare and Bioml document collections were imported. The import of each document represented a separate transaction. The experiments were performed using the storage formats introduced in Section 5.8. One storage format clusters nodes into physical records, while the other stores every node in a single record.

Three configurations of the recovery subsystem were examined. The first is a recaptitulation of the results without recovery from Section 5.8.2. The second uses regular logging while importing the documents. The third configuration enabled subsidiary logging during import. The Log Buffer was configured to a capacity of 20 pages.

7.9.2 Results

Table 7.6 presents the required time to perform the import, and the amount of log produced.

We make the following observations. First, the time overhead for logging varies greatly, between 10% and 150% compared to import with no logging. Second, subsidiary logging significantly reduces the recovery overhead. It cuts the time spent for recovery by at least 50%. The CPU usage for recovery is 60%-80% lower with subsidiary logging, and log generation is often reduced by a factor of three. Third, the clustered storage format causes less recovery overhead, which is easily explained because the single node format needs to write two log records for every node, one to add the node's record and one to include it in the children list of its parent node.

Finally, document import with both of our XML-specific optimizations enabled ourperforms document import with regular techniques by a factor of four with respect to elapsed time, while producing an order of magnitude less log.

Chapter 8

Conclusion

Too many pieces of music finish too long after the end.

–Igor Stravinsky

This work investigates core techniques for XML Base Management Systems (XBMS). XBMS are database management systems (DBMS) that are designated to process XML document collections.

After presenting the XML language itself, query languages for XML, and application programming interfaces to process it in Chapter 2, we analyze the requirements for XBMS (Chapter 3).

We found that in principle, these requirements do not differ from those for DBMSs for other kinds of data. However, due to the different data model, new query languages and the incurred novel access profiles, regular DBMSs are not equipped to play the role of XBMS. We have seen that simple XML add-on modules are no remedy. The need for special XML support permeates the complete DBMS architecture.

In Chapter 4, we introduced Natix, a *Native* XBMS designed from the ground up for XML processing. We gave an overview of its architecture, and presented some scenarios and examples for its usage. Since a thorough discussion of all of Natix techniques and modules is beyond the scope of this work, in the remaining chapters we focussed on the fundamental modules, the storage engine, schema management, and recovery.

The storage engine is the single most important module in the system (Chapter 5). In Natix, it provides a general state-of-the-art subsystem for transferring data from and to persistent storage. It was engineered for extendibility, as the emergent area of XBMS research will continue to produce novel storage techniques and optimizations, which have to be incorporated.

On the basis of the extendible general framework, we developed a novel XML storage format. We store connected subtrees of XML document trees in physical records smaller than a page and use a very compact representation for subtrees in physical records. Bulk-load and update algorithms were presented. Performance measurements demonstrated the superiority of our approach compared with conventional storage organizations for trees, during updates as well as during query processing.

Chapter 6 was devoted to schema management. Natix allows to group documents into logical hierarchies, and to specify constraints on the structure of documents within a collection. Distribution of collections onto physical media is controlled by a physical schema specification. We explain how the logical and physical schemas are implemented using the storage primitives from Chapter 5.

As a final example how core technologies for XBMS are different from regular DBMS, we reviewed Natix's recovery subsystem in Chapter 7. It provides a general recovery framework, which is as easily extendible as the storage engine, requires very little representation changes and copying of data, and incorporates sophisticated recovery mechanisms for physical metadata. The physical records used for XML storage can become larger than records in typical relational or object-oriented DBMS applications, and recovery needs to take this into account. To this end, we introduced novel techniques. First, there is *Subsidiary Logging*, which amortizes logging costs for many small node updates by using larger log records. *Annihilator Undo* reduces undo overhead when our XML storage format is used. *Selective Restart* increases system availability by speeding up restart. Performance measurements show again that our techniques dramatically improve performance for XML processing tasks.

Much remains to be done. There is still a gap between the functionality of a DBMS for other kinds of data and an XBMS. Although our exposition was limited to the run-time system, we have left many areas of research for XBMS untouched. We did not elaborate on index structures and their applications and maintenance. Techniques for object managers for DOM trees can be optimized, and it is unclear how efficient support of online revalidation after small updates can be implemented. Query execution was also omitted. These areas of the run-time system leave much research issues related to XML, even without examining query compilation and optimization, which are other large areas.

Bibliography

[1] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Very large data bases, VLDB '93: proceedings of the 19th International Conference on Very Large Data Bases*, pages 73–84, Dublin, Ireland, August 1993.

[2] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: A relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976. Also published in/as: IBM, San Jose, Research Report. No. RJ-1738, Feb. 1976. Reprinted in [86].

[3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. Technical Report RFC2396, The Internet Engineering Task Force, August 1999. URI `http://www.ietf.org/rfc/rfc2396.txt`.

[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] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes. Technical report, World Wide Web Consortium (W3C), May 2001. URI `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.

[6] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, 2002. W3C Working Draft 30 April 2002.

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

[8] Jon Bosak. The plays of shakespeare in XML. URI `http://www.oasis-open.org/cover/bosakShakespeare200.html`. `xml-dev` message, 1999.

[9] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. Technical report, World Wide Web Consortium (W3C), 1999. URI `http://www.w3.org/TR/1999/REC-xml-names-19990114`.

[10] 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. URI `http://www.w3.org/TR/2000/REC-xml-20001006`.

[11] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[12] Peter Buneman. Semistructured data. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 117–121, New York, NY 10036, USA, 1997. ACM Press.

[13] Peter Buneman, Susan B. Davidson, and Dan Suciu. Programming constructs for unstructured data. In Paolo Atzeni and Val Tannen, editors, *Database Programming Languages (DBPL-5), Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995*, Electronic Workshops in Computing, page 12. Springer, 1995.

[14] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and fi le management in the EXODUS extensible database system. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100, Los Altos, California, USA, 1986.

[15] Paolo Casarini and Luca Padovani. The gnome DOM engine. In *Aggregated Proceedings for the Extreme Markup Languages Conferences (2001-2002)*, 2001.

[16] Don Chamberlin. *A Complete guide to DB2*. Morgan Kaufmann, San Francisco, 1998.

[17] James Clark. XSL transformations (XSLT) version 1.0. Technical report, World Wide Web Consortium (W3C), November 1999. URI `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[18] James Clark and Steve DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C) Recommendation, 1999. URI `http://www.w3.org/TR/xpath`.

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

[20] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadelphia, Pennsylvania, USA, June 1999.

[21] R. Dey, M. Shan, and I. Traiger. Method for dropping data sets. *IBM Technical Disclosure Bulletin*, 25(11A):5453–5455, April 1983.

[22] J¨urgen Dufner. Design and implementation of a Java-API for Natix. Master's thesis, University of Mannheim, Mannheim, Germany, January 2001. (in German).

[23] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[24] André Eickler, Carsten Andreas Gerlhof, and Donald Kossmann. A performance evaluation of OID mapping techniques. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB '95: proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Sept. 11–15, 1995*, pages 18–29, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[25] T. Fiebig and G. Moerkotte. Algebraic XML construction in Natix. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE'01)*, pages 212–221, Kyoto, Japan, December 2001. IEEE Computer Society.

[26] T. Fiebig and G. Moerkotte. Evaluating queries on structure with extended access support relations. In *The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA, Maaay 18-19, 2000, Selected Papers*, volume 1997 of *Lecture Notes in Computer Science*. Springer, 2001.

[27] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a Native XML base management system. *VLDB Journal*, page (to appear), 2003.

[28] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical Report RFC2616, The Internet Engineering Task Force, June 1999. URI `http://www.ietf.org/rfc/rfc2616.txt`.

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

[30] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, USA, 1995. See book review [71].

[31] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fastpath. *IEEE Data Engineering Bulletin*, 8(2):3–10, 1985.

[32] 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. URI `http://www.ietf.org/rfc/rfc2518.txt`.

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

[34] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks in a large shared data base. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases, Framingham, MA, USA, September 22–24, 1975*, pages 428–451, New York, NY 10036, USA, 1975. ACM Press. ACM SIGMOD v. 1, no. 1, September 1975.

[35] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. Recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

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

[37] Theo Haerder and Andreas Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. Also published in/as: Res. R No. 50-82, April 1982. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan Kaufmann, San Mateo, CA, 1988.

[38] T. H¨arder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001.

[39] Elliotte Rusty Harold. Xom. URI `http://www.cafeconleche.org/XOM/`. Project Web Site, 2002.

[40] Stefan Haustein and Aleksander Slominski. Common api for xml pull parsing. URI `http://www.xmlpull.org`. Project Web Site, 2002.

[41] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In D. Gawlick, M. Haynie, and A. Reuter, editors, *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, volume 359 of *LNCS*, pages 301–328, Berlin, September 1989. Springer.

[42] S. Helmer, C.-C. Kanne, and G. Moerkotte. Isolation in XML bases. Technical Report Nr. 15, Lehrstuhl f¨ur Praktische Informatik III, Universit¨at Mannheim, 2001.

[43] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proceedings of the 3nd International Conference on Web Information Systems Engineering (WISE'02)*, page (to appear). IEEE Computer Society, 2002.

[44] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. Erratum in *Communications of the ACM*, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the fi rst solutions to the Dining Philosophers problem.

[45] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (DOM) level 2 core specifi cation. Technical report, World Wide Web Consortium (W3C), 2000. URI `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`.

[46] Jason Hunter and Brett McLaughlin. The jdom project. URI `http://www.jdom.org`. Project Web Site, 2002.

[47] Nicolai M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, Reading, MA, USA, 1999.

[48] Carl-Christian Kanne and Guido Moerkotte. Effi cient storage of XML data. Technical Report Nr. 8, Lehrstuhl f¨ur praktische Informatik III, Universit¨at Mannheim, June 1999.

[49] 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.

[50] Henry F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[51] Sukhamay Kundu and Jayadev Misra. A linear tree partitioning algorithm. *SIAM J. Comput.*, 6(1):151–154, March 1977.

[52] Tobin J. Lehman and Bruce G. Lindsay. The Starburst long field manager. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 375–383, Amsterdam, The Netherlands, August 1989.

[53] B. G. Lindsay, C. Mohan, and M. H. Pirahesh. Method for reserving space needed for rollback actions. *IBM Technical Disclosure Bulletin*, 29(6):2743–2746, November 1986.

[54] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, May 1974.

[55] Mark L. McAuliffe, Michael J. Carey, and Marvin H. Solomon. Towards effective and efficient free space management. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 389–400, Montreal, Canada, June 1996.

[56] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[57] David Megginson. SAX: A simple API for XML. Technical report, Megginson Technologies, 2001. URI `http://www.saxproject.org/`.

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

[59] Guido Moerkotte. Incorporating XSL processing into database engines. In *Proceedings of the 28th VLDB*, 2002.

[60] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In Dennis McLeod, Ron Sacks-Davis, and H.-J Schek, editors, *Very large data bases: 16th International Conference on Very Large Data Bases; August 13–16, 1990, Brisbane, Australia*, pages 392–405, Los Altos, CA 94022, USA, 1990. Morgan Kaufmann Publishers.

[61] C. Mohan. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. In *Proceedings/Ninth International Conference on Data Engineering, April 19–23, 1993, Vienna, Austria*, page 243, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, April 1993. IEEE Computer Society Press.

[62] 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.

[63] C. Mohan. Commit_lsn: A novel and simple method for reducing locking and latching in transaction processing systems. In Vijay Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 307–335. Prentice-Hall, 1996.

[64] C. Mohan. Repeating history beyond ARIES. 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 1–17, Los Altos, CA 94022, USA, 1999. Morgan Kaufmann Publishers. Also known as VLDB'99.

[65] 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.

[66] 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.

[67] 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.

[68] C. Mohan and I. Narang. ARIES/CSA: A method for database recovery in client-server architectures. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):55–66, June 1994.

[69] 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.

[70] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. 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 251–260. IEEE Computer Society, 1995.

[71] George Patapis. *Design Patterns, Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *C/C++ Users Journal*, 13(10):78–??, October 1995. See [30].

[72] Apache XML Project. Xalan C++ version 1.4. URI `http://xml.apache.org/xalan-c/index.html/`. Project Web Site, 2002.

[73] Apache XML Project. Xerces C++ parser. URI `http://xml.apache.org/xerces-c/index.html`. Project Web Site, 2002.

[74] Apache XML Project. Xerces2 java parser. URI `http://xml.apache.org/xerces2-j/index.html`. Project Web Site, 2002.

[75] The Apache HTTP Server Project. The apache http server project. URI `http://httpd.apache.org/`. Project Web Site, 2002.

[76] Kurt Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. 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 337–346, Los Altos, CA 94022, USA, 1989. Morgan Kaufmann Publishers.

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

[78] Mario Schkolnick. A clustering algorithm for hierarchical structures. *ACM Transactions on Database Systems*, 2(1):27–44, May 1977.

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

[80] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[81] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[82] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, and J.F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, UK, 1999.

[83] Dennis Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, March 1988.

[84] Kimbro Staken. XML database API draft. Technical report, The XML:DB Initiative, 2001. URI `http://www.xmldb.org/xapi/xapi-draft.html`.

[85] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[86] Michael Stonebraker. *Readings in Database Systems*. Morgan Kaufmann Publishers, San Francisco, CA 94104-3205, USA, 1988.

[87] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976. Reprinted in [86]. Also published in/as: UCB, Elec. Res. Lab, Memo No. ERL-M577, Jan. 1976.

[88] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

[89] 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.

[90] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures. Technical report, World Wide Web Consortium (W3C), May 2001. URI `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`.

[91] R. van Zwol, P.M.G. Apers, and A.N. Wilschut. Modeling and querying semistructured data with MOA. In *ICDT'99 Workshop on Query Processing for semistructured data*, 1999.

[92] Daniel Veillard. The XML C library for gnome. URI `http://xmlsoft.org/index.html`. Project Web Site, 2002.

[93] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[94] 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.

[95] 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 artifi cial intelligence; McGraw-Hill series in artifi cial intelligence. McGraw-Hill, New York, NY, USA, 1987.

[96] Tak W. Yan and Jurgen Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *Very large data bases, VLDB '94: proceedings of the 20th International Conference on Very Large Data Bases*, pages 740–749, Santiago, Chile, 1994.

Index

Statler: *I guess all’s well that ends well.*
Waldorf: *I don’t care, as long as it ends.*

–