NaXDB – Realizing Pipelined XQuery Processing in a Native XML Database System

Jens Hündling, Jan Sievers and Mathias Weske
Hasso-Plattner-Institute for IT-Systems-Engineering
at the University of Potsdam, Germany
{jens.huendling|jan.sievers|mathias.weske}@hpi.uni-potsdam.de

ABSTRACT

Supporting queries and modifications on XML documents is a challenging task, and several related approaches exist. When implementing query and modification languages efficiently, the actual persistent storage of the XML data is of particular importance. Generally speaking, the structure of XML data significantly differs from the well-known relational data-model. This paper presents the prototypical implementation of NaXDB, a native XML database management system (DBMS). A native approach means the XML data is stored as a hierarchical tree of linked objects. NaXDB is implemented as a MaxDB by MySQL kernel module and thus inherits several DBMS functionality. Furthermore, NaXDB uses object-oriented extensions of of MaxDB by MySQL to store the tree of linked objects. NaXDB implements a large subset of the query language specification XQuery 1.0 as well as XUpdate for modification. The design and architecture of the prototypical implementation is presented, and concepts for query processing within the database server are discussed.

1. INTRODUCTION

The significance of semi-structured data in the context of the eXtensible Markup Language (XML) is growing rapidly, with applications in advanced text processing, Web services technology, information exchange across organizational boundaries and Enterprise Application Integration, to name a few. XML documents often contain both semi-structured text data (e.g., sections, paragraphs, formatting) as well as business related well structured data, like order data. XML documents are adequately represented in a tree-structured way with an optionally defined schema. Managing XML data is a challenging task and several related approaches exist [6]. This paper reports on NaXDB, a recent development that aims at providing native storage of XML documents as well as XQuery [14, 4] and XUpdate [17] functionality. NaXDB is based on the open-source DBMS MaxDB by MySQL [20] and is a joint effort between the University of Potsdam and SAP Labs Berlin, who develop MaxDB. This paper presents design and implementation concepts of NaXDB.

Since an XML database system must handle different document schemas and accept documents with unknown structure, efficient automatic adoption of evolving schemas from incoming information is vital. In addition, it is necessary to efficiently adapt the evolving schema of incoming information with minimal manual intervention. This significantly differs from traditional relational DBMS, where database schema evolution is rare and typically managed by database administrators. Due to these reasons, NaXDB uses a native approach [6], i.e. the XML information is stored as a hierarchical tree of nodes. NaXDB implements an huge subset of the query language specification XQuery 1.0 [3] including XPath 2.0 [2] for navigation as well as XUpdate [17] for manipulation. Remarkably, several implementations of XQuery exist by now, but only some are real database management systems offering typical functionality like multi-user access, transaction handling, recovery, etc. Among those real XML DBMS only a few use a native approach, but are rather transforming XML data into relational tables. NaXDB is implemented specifically for a native XML storage by using an object-oriented persistent storage extension of MaxDB called Object Management System (OMS) [1]. Thus, it is possible to directly store the XML tree and still remain tightly integrated in the MaxDB server as a kernel module [1].

The rest of this paper is organized as follows: Section 2 introduces the system architecture and addresses the responsibilities of the components as well as their interaction. The architecture of the current prototypical implementation is presented in Section 3. Pipelined query processing, problems and solutions based on NaXDB's native object-oriented storage are discussed in Section 4 and Section 5 depicts an example and thereby accentuating the internal tree-like concepts for handling XQuery statements. The paper concludes with a discussion and an outlook on future work.

2. NAXDB SYSTEM OVERVIEW

The system architecture of NaXDB is presented in Figure 1. The system has three main subsystems: 1. A *Database Client*: A graphical user interface that enables users to write queries and receive results. 2. The *Database Server* is the core subsystem of NaXDB; it includes a number of components that will be discussed in the remainder of this paper. 3. The *Persistent Object Manager* is responsible for persistently storing XML data.

At a high level of abstraction, the interaction between these subsystems are as follows: The user accesses the graphical user interface and formulates a query. The query is then transferred to the database server, which in turn processes the query. Finally, the database server generates code and uses the persistent storage subsystem, which returns the results that are transferred back to the client. As already mentioned, the storage subsystem is based on an object oriented extension of MaxDB and thus benefits from the advantages of native XML storage [6]. Furthermore, NaXDB in-

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

herits comfortable database features from MaxDB. For instance, MaxDB handles the persistent storage on hard-disk pages and uses Multi Version Concurrency Control (MVCC) for concurrent multi-user access and transaction processing. Following MaxDB's build strategy, NaXDB can be deployed to Windows platforms as well as numerous UNIX flavors, including new 64 bit architectures.

NaXDB offers two communication options: A WebDAV interface and JDBC stored procedures. Using WebDAV [22], the user can navigate in the collections like in a file system. By indicating files to be imported "as XML" the NaXDB server stores them in the XML repository if they are well-formed. Furthermore, applications can send queries to NaXDB via JDBC stored procedures.

3. IMPLEMENTATION OF NAXDB

The prototypical implementation of NaXDB is explained in this section. This will be done by explaining the key components of the architecture depicted in Figure 1 and their query processing interaction. Thus, the following addresses the design and implementation of the Request Dispatcher, the query parsing and translating components, the Query Execution Engine, and the persistence management. The section also points to interesting features and concepts of NaXDB.

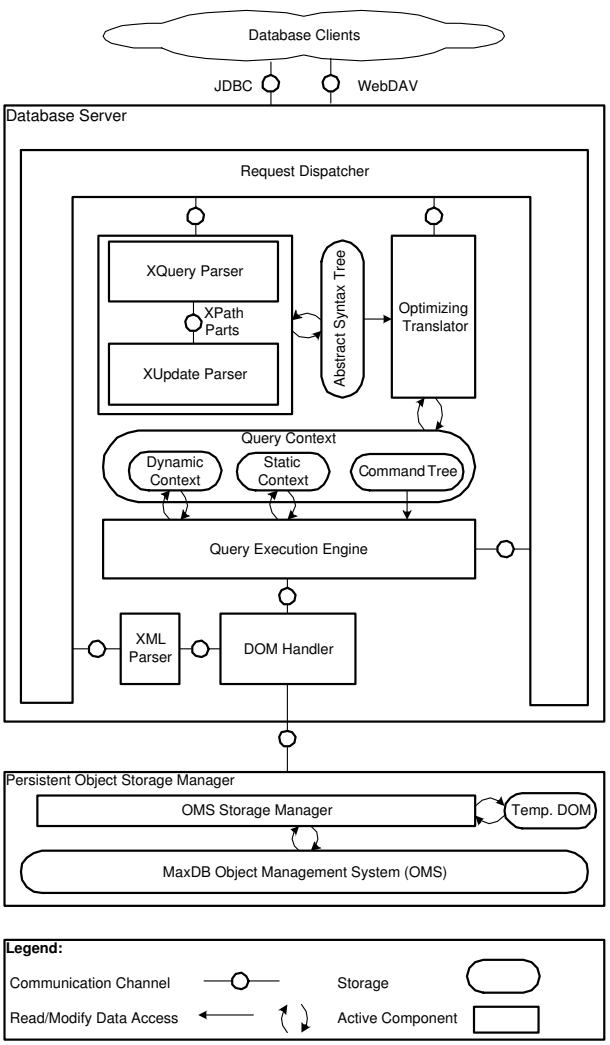

Figure 1: System architecture of NaXDB

3.1 Request Dispatcher

The *Request Dispatcher* component provides stored procedures for XQuery and XUpdate functionality and handles WebDAV requests. Additionally, the transaction handling is done by the Request Dispatcher by using MaxDB's core functionality. It also offers enhanced methods for query handling, e.g. resolution of document identifiers.

3.2 Parser Components

NaXDB's parser has two components: An *XQuery Parser* and an *XUpdate parser*. Since XUpdate [17] is an XML dialect, its XML structure is parsed using the open-source XML parser expat [7] and the XPath parts of XUpdate are parsed using the XQuery parser. The XUpdate statements are mapped to the same (tree-structured) internal representation, i.e. the internal representation is capable of modifications as well as XQuery operations. From now on we will not further distinguish between XUpdate and XQuery statements.

To handle the syntactical complexity of XQuery, the XQuery parser was built using the ANTLR Parser Generator [19], which turned out to be a quite helpful tool. For using ANTLR, we have developed an XQuery grammar with relatively little effort and produced a fast and fully featured XQuery parser. One interesting feature of the NaXDB parser is its ability to handle the ambiguity of expressions, emerging with non-reserved keywords. For example, the keyword `for` is not a reserved word in XQuery. Thus, element names as well as user defined functions may be named `for` as well; this can be recognized by the NaXDB parser correctly.

As a first step to generate an internal representation, the parsers transform the query strings to an abstract syntax tree (AST). The AST is inspired by the XQuery Core language assumed in the formal semantics of XQuery [8], but still closer to XQuery. Some equivalence rewritings are already made by the XQuery parser to reduce complexity of the query. Additionally, this rewriting decreases the possible number of operators of the internal algebra, which is produced in the next processing step, e.g. `where`-statements are transformed to `if`-statements, but predicates are left as they are. The parser output is an AST, which is a equivalent representation of the original query. Thus, XUpdate and XQuery statements are both generating trees based on the NaXDB internal language.

3.3 Optimizing Translator

The *Optimizing Translator* takes the AST as input and recursively traverses it building the *Command Tree* at the same time. While translating the AST into a Command Tree, some optimizations are accomplished. For instance, the Optimizing Translator is able to detect whether and when sorting or duplicate removal is absolutely necessary, thereby following the ideas presented by Jan Hidders and Philipp Michiels in [12]. This can save a lot of time, because otherwise the intermediate results have to be sorted after every axis step, following the XQuery specification [3]. Moreover, static XQuery errors can already be detected by this component.

In difference from many other solution, NaXDB uses an tree-structured internal query representation. This offers a key advantage: code execution through traversal of the Command Tree's objects. For each operator of the language a specific class containing the processing logic exists. The Optimizing Translator instantiates these classes corresponding to the query and the resulting objects build the Command Tree. The Query Execution Engine starts the traversal by calling the evaluation method of the root object (i.e. root node) of the Command Tree. Each tree node knows how to proceed and when to evaluate its children. While XQuery as well as the AST generated from XQuery is of functional, declarative na-

ture, the Command Tree has imperative character; a sample Command Tree and processing will be described in Section 5.

3.4 Query Execution Engine

The *Query Execution Engine* processes the Command Tree. Each node leads to a single operation, e.g. get the parent of the actual node in the document or calculate the sum of the results of the next two Command Tree nodes. While doing so, the engine uses the Query Context data structure, which holds information related to the current query, including some proposed by the XQuery specification [3]. As defined in the specification, static information, like the ordering mode, has already been set in the static analysis phase by the Optimizing Translator. Dynamic information, like the context position, is set by the Query Execution Engine itself. Besides the information mentioned in the specification, the Query Execution Engine uses the Query Context to store internal information needed for more than one processing step.

NaXDB implements a pipelining XQuery processor [15, 11], i.e. it immediately processes the *next* subexpression for each item resulting from the *current* subexpression. In contrast, a non-pipelining processor first collects all items of the first subexpression in memory and then iterates them for the next subexpression. This can lead to enormous intermediate results in the memory. Especially using axes like `descendant-or-self` it would quickly result in holding the document as a whole – may be more than once – in memory. As we experienced, the object-oriented execution of the tree-structured query algebra as explained above fits very well into the pipelined query processing; the advantages and also some problems are discussed in Section 4.

Additionally, NaXDB supports *early termination*, which enables the Query Execution Engine to stop the processing of expressions [5] immediately, when the result can be computed. When a boolean value must be computed, e.g. in conditional expressions in an `if` statement, the evaluation stops, when a node is found. Finding any more nodes or atomic values does not change the result.

Lazy evaluation of variables [15] is another feature of our implementation. Lazy evaluation implies that only those variables which are actually used during the execution of a query will be evaluated. In some cases, this technique leads to faster processing and less memory usage; consider the example in Figure 2.

```
declare variable $doc := doc('book.xml');
declare variable $en := $doc//part[@lang='en'];
declare variable $de := $doc//part[@lang='de'];

if($doc/@lang = 'en') then $en else $de
```

Figure 2: Lazy Evaluation Example

Here only one variable is used; which one is determined at runtime. Using lazy evaluation the Query Execution Engine evaluates a variable only if and exactly when it is actually used. Therefore, lazy evaluation is a useful technique to improve the performance of XML databases.

3.5 Data Management

The native storage of XML data is realized using the Object Management System (OMS), which is based on the liveCache technology of MaxDB. It allows storing variable and fixed size objects of arbitrary structure and thereby storing XML data directly as a hierarchical tree of linked objects. Generally speaking, a native storage means that for each XML element, attribute etc. in the XML tree an object is created in the database. As mentioned in section 1, this allows to store all kinds of XML documents, even with unknown structure and without schema definitions. Each object has an unique object identifier (OID) which is defined by the OMS and these identifiers are used for linkage between objects. In NaXDB the Persistent Object Storage Manager (POSM) encapsulates this functionality.

NaXDB has two key optimizations implemented in the POSM: First, the structure of the nodes is stored separately from the content of the nodes. This is profitable especially for navigation, not because the potentially huge node has to be handled, but only a small object containing an OID, a minimal set of links allowing bi-directional-navigation, and the link to a content-object. With this approach it is possible to cluster all structure-relevant objects on a small number of pages and therefore gain further performance improvements for navigation. The idea is to keep (a big part of) the XML-structure in memory, which leads to accelerated navigation. Caching of these structure-objects is automatically handled by the POSM. The second optimization is storing all qualified names and namespaces in a global hash table, which can be cached. Especially in data-centric documents a lot of redundancy can be avoided, e.g. the element and attribute names are stored only once. The OID's of the structure-objects are stored in the table, which in turn point to the content-objects. Additionally, performance improvements are achieved by comparing the hash values.

Furthermore, the OMS offers multi version concurrency control (MVCC), which provides every transaction with a consistent view on the XML data. Each transaction gets a version number and works on a consistent snapshot of all objects. As well known from database theory, new versions of objects can be created and only if two transactions modify the same object, one has to be rolled back. This enables multi user read access for NaXDB as well as multiple users to manipulate the same XML document. As long as the users do not change the content or the structure of the same node, the queries will succeed. Otherwise an error message will be returned and the transaction is rolled back.

For convenience reasons, an intermediate Data Object Model (DOM) layer is internally provided by the *DOM Handler*. Thus, the Query Execution Engine handles the persistent objects as known from DOM. These objects are also light-weight structure-objects pointing to the content objects. A *Temporary DOM* holds such DOM objects with references to non-persistent POSM objects. The Temporary DOM is used for two reasons: 1.) For constructed elements or attributes from XQuery statements. 2.) For new nodes from an XUpdate statement. In the first case these nodes are automatically deleted after the transaction finishes and the result is returned; in the second case the nodes become persistent POSM objects, when the XUpdate statement succeeds. In case of `insert` or `update` statements, the new objects are inserted (or updated) in the tree. This means, new (versions of) structure-objects are created and the references of the neighbour nodes are updated. By using the MVCC mechanism, the POSM creates new versions of these neighbouring objects and thus, already running transactions can still read the old structure. This is especially useful, because now transaction handling is independent from query processing. With this approach it is further possible to provide direct access to a DOM layer, which would allow applications to navigate on persistent XML documents like on a DOM representation in main memory.

4. LESSONS LEARNED FROM NAXDB

During the implementation of NaXDB we learned found and solved some of XQuery's difficulties. Especially regarding pipelined query processing, which is not part of the specification, numerous severities became visible. This section discusses some problems and solutions.

4.1 Pipelined XQuery Processing

The XQuery specification as well as the Formal Semantics [8] suggest a non-pipelining approach for XQuery evaluation, which is straight forward and intuitive at a first glance. This implies to handle intermediate result sequences directly, i.e. analyze and process them directly. Therefore, the intermediate results have to be held in memory, which possibly needs enormous resources. Without optimization, intermediate results can contain the complete document, sometimes even more than once. For instance, queries with the `descendant-or-self` axis might not be problematic for a single, rather small document, but in a production environment, a great amount of XML data is processed concurrently and documents can be large. Obviously, holding intermediate results in the memory leads to a major performance drawback. Therefore, a pipelining approach attempts to minimize the need of memory during query processing. This is done by avoiding to build intermediate result sequences, i.e. the pipelining approach tries to evaluate the next expression exactly when an item of the currently processed expression is found.

```
doc('book.xml')/article/chapter
```

Figure 3: Intermediate Results Example

When processing the query in Figure 3, for each `article` element found, the NaXDB Query Execution Engine evaluates a child axis fetch, searching for child elements named `chapter`. If one is found, it is added to the final result sequence. Thus, there is no sequence of articles built at any time during the query processing. Nevertheless, implementing the pipelining approach leads to some difficulties.

4.2 Problems of Pipelined XQuery Processing

Several XQuery expressions are *breaking the pipeline*. For instance, parenthesized expressions naturally require to evaluate the intermediate result completely. Additionally, set expressions like `union` demand both resulting sequences of the subexpressions to be held in memory. In both cases the Query Execution Engine has to evaluate the corresponding expression completely, before evaluating the next expression. Thenceforwards, the query processing can take place in pipelining mode again.

```
doc('book.xml')/article/chapter[last()]
```

Figure 4: The `last()`-Function Example

Sometimes specific information about intermediate results are needed but are missing in pipelined processing, like the example in Figure 4 illustrates. The query uses the build-in `last()`-function from the XQuery function library. This function returns the position of the last item in the current sequence (here: sequence of `chapter` elements). Since XQuery numbers the first item with `1`, the last position number equals the item count of the sequence. To get this count, all items of the sequence must be retrieved before the predicate containing the `last()`-function can be computed, i.e. the pipeline has to be broken. In NaXDB this is realized in the static analysis phase, which leads to a reorganized Command Tree. The resulting Command Tree is equal to the tree of the query shown in Figure 5.

In contrast to the `last()`-function the `position()`-function as well as numeric predicates are only problematic in certain scenarios. To be be more precise, these operations only brake the

```
doc('book.xml')/article/(chapter)[last()]
```

Figure 5: NaXDB's interpretation of the `last()`-Example

pipelining, when combined with unordered sequences. The examples in Figure 6 use numeric predicates. The Query Execution Engine does not count all occurrences of items (here `chapter` elements), but tests if the position of the current item equals the given number of the numeric predicate. Nevertheless, if the current expression results in an unordered sequence, the intermediate result collects the items in an incorrect order and thus the Query Execution Engine assumes wrong position numbers.

On that score, it is vital to be able to determine if the correct order of a sequence can be guaranteed. If so, the query processing can be pipelined, but otherwise for a numeric predicate the corresponding sequence must be sorted before evaluating the predicate. Obviously, all items of a sequence are needed to bring it into document order. To sum up, only numeric predicates that lead to sorting are breaking the pipeline.

```
doc('book.xml')/article/chapter[2]
doc('book.xml')/article/preceding::author[1]
```

Figure 6: Numeric Predicates-Examples

In the first example in Figure 6, no ordering operation is needed. The second query contains a step on the child axis followed by a step on the preceding axis. This results according to [12] in an unordered sequence. This is handled by NaXDB in the static analysis phase, i.e. by the Optimizing Translator, which inserts an `Order Result Sequence` operator before the operator for numeric predicate evaluation. A description of the sorting logic is presented in the next section.

4.3 Sorting in Pipelined XQuery Processing

The XQuery specification exacts the results of axis steps and set expressions (like `union`) to be in document order. Document order is informally defined as

> "...the order in which the nodes appear in the XML serialization of a document [3]."

Since ordering means sorting of a sequence of possibly numerous nodes, every implementation tries to minimize the number of ordering operations. Jan Hidders and Philip Michiels developed an automaton, which decides if the result of a given sequence of axis steps is in document order or not [12]. NaXDB implements such an automaton to minimize the number of ordering operations. However, further information is needed to determine if at a given step of processing, the intermediate result should be sorted or not. Apparently, an intermediate result in document order does not required sorting. Nevertheless, some expressions require sorting in any case, as mentioned above.

Sorting every time the automaton reports an intermediate result to be in incorrect order, is not appropriate. In some cases an unordered sequence gets ordered again after the next axis step [12]. Additionally, it is irrelevant if an intermediate result is in document order or not. For instance, for expressions computing the effective boolean value (EBV) [3], the order within the sequence is not relevant. An example for a realization in NaXDB is the conditional expression of `if`-expressions, where the Query Execution Engine does not sort intermediate results (except the cases mentioned above).

Because every ordering operation *breaks the pipeline*, it lessens the performance advantage of pipelined XQuery processing. NaXDB solves this by determining beforehand whether sorting is necessary or not using the depicted technique. Nevertheless, if sorting in necessary, this is done using the structure-objects. For instance, the correct order of two nodes can be determined by stepping up the parent axis of the tree until a common parent is found. Since the structure is realized by light-weight objects in memory, this can be done efficiently and only the OID's have to be compared.

5. EXAMPLE

In this section, the sample query in Figure 7 will be used to explain the Command Tree and the Query Execution Engine of NaXDB. The sample query selects specific book from the `bib.xml` file in a NaXDB repository and returns a number of specific chapter elements.

```
for $book in doc('bib.xml')/bib/book[abstract]
where $book/author = 'Paul Meier'
return
 $book/chapter[5]/preceding-sibling::chapter
```

Figure 7: Sample XQuery

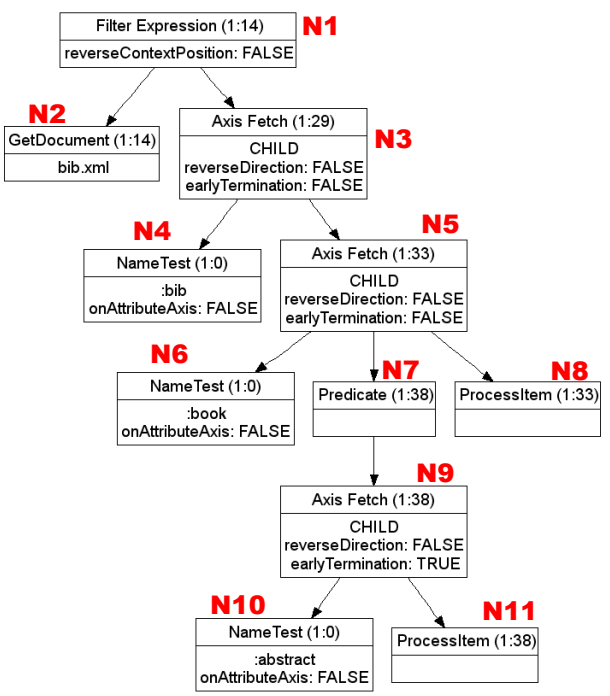

Figure 8: Sample Command Tree - Part 1

In Figures 8 and 9 the Command Tree resulting from the sample query is shown. These figures are created by using a debugging feature of NaXDB that exports a graphical representation of the Command Tree. For better readability, labels **N1**-**N18** are added. It has to be mentioned that the tree is actually split in two parts to fit in the paper. The Command Tree starts in the root node of Figure 8 (with the label **N1**). From the rightmost leaf-node of this tree (**N8**), the execution goes on with the root node of Figure 9 (**N12**). Each node has a textual representation of the operation in the top half of the node and `line:column` (of the query statement) in brackets. The bottom half of the node shows flags and parameters. The remainder of this section describes the meaning of the graphical representation and points to used optimization features shown in the sample trees.

Generally speaking, the processing logic imitates a top-down, from left to right traversal of the tree. Therefore, the evaluation starts at the Filter Expression root node (**N1**) of Figure 8. All items are collected that are returned by the `GetDocument` operator (**N2**, realizing the `doc()`-function) and for each item **N3** is evaluated. **N3** is the beginning of the path expression starting with a child step (**N3**) and a `NameTest` of this node for `bib`. The path continues with a steps on the child axis (**N5**), testing for `book` and starting a predicate (**N7**). This predicate tests the existence of `abstract` (**N10**) children (**N9**). If the predicate is fulfilled, i.e. a satisfying node is found, it can be processed (**N11**), which in turn continues the traversal at node **N8**. To sum up, the first part of the Command Tree (Figure 8) represents the binding variable sequence of the `for` expression.

For each item found in the expression so far, the `ProcessItem` operator (**N8**) calls the `ForStep` node (**N12** in Figure 9). **N12** binds the context item to the variable `book` and evaluates its child. The `IfExpr` operator (**N13**) represents the `where`-clause. The condition expression starts with **N14** and the `then`-expression with **N15**. Since no `else`-expression is needed representing the `where`-clause, an empty expression with no children (**N16**) is the last child of **N13**.

Futhermore, two nodes of the Command Tree are of particular interest:

1. **N18** in Figure 9 is an `OrderResultSequence` operator. This is necessary, because the path expression in line 4 of the query possibly returns an unordered sequence of `chapter`-elements. Since the element at position 5 is needed, a result sequence in correct order is necessary here.

2. The `AxisFetch` node **N9** underneath the `Predicate` operator (**N7** in Figure 8. The `earlyTermination` property is set `TRUE`, because it is sufficient to find exactly one `abstract` element to decide whether the `book` element should be returned or not. This means the predicate is fulfilled when the first `abstract` element is found.

Given this information, the explanation of the other nodes is straight forward. The example shows how the Query Execution Engine handles XQuery statements by traversal of the Command Tree. It has to be mentioned at this point that this leads to navigation on the structure-objects, comparisons using the hash table for global names and fetching the content-objects as in Section 3.5. Thereby navigating in the object tree of the XML document is implemented. Additionally, optimizing features are realized within the Command Tree of NaXDB.

6. DISCUSSION AND OUTLOOK

NaXDB is a prototypical implementation of a native storage for XML data and supports advanced XQuery and XUpdate processing. XQuery is a powerful and complex query language aimed at

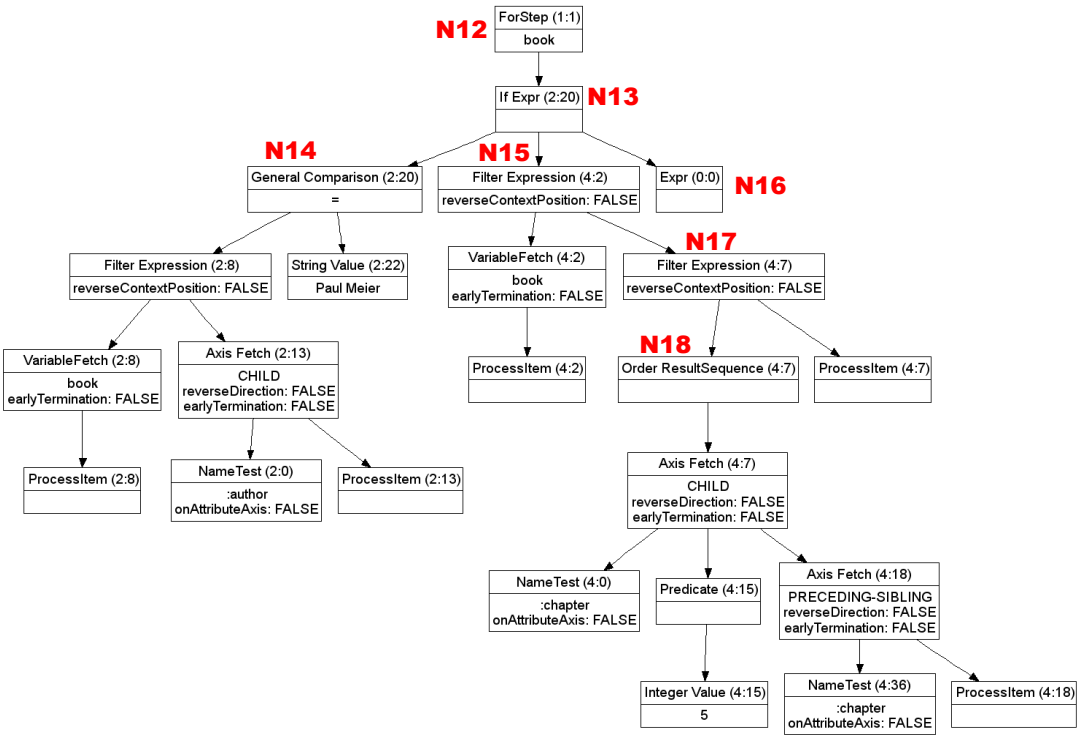

Figure 9: Sample Command Tree - Part 2

XML data. As we recognized during the NaXDB project, an efficient implementation of some aspects of the XQuery specification is a challenging task. A native storage of XML data as shown in Section 3 is advantageous since an intuitive execution of navigation steps can be realized, as explained in Section 4 and depicted by an example and Section 5. Hence, NaXDB realizes a pipelined query processing approach and handles several difficulties of this approach, e.g. sorting the intermediate results using the tree structure. The NaXDB Query Execution Engine implements several optimizing features; a few where mentioned in this paper.

Additionally, NaXDB is a fully-fledged DBMS by using several features of MaxDB by MySQL, e.g. MVCC for objects. Opposed to existing XQuery implementations, NaXDB features very strict whitespace handling by storing the XML Infoset as native tree and realizing a persistent XML storage that allows concurrent modifying access to the stored documents. The current version supports a huge subset of XQuery, including path expressions for all axes, arithmetic, comparison and logical expressions as well as `for let where return` statements. NaXDB does currently not support XML schema processing. Additionally, calls to (predefined) functions are implemented (although the function library is not completely implemented). NaXDB's implementation leaves out typing statements, like `cast` or `typeswitch`. This is scheduled for future work.

Next to realizing the above mentioned, a future step should be exploitation of benchmarks on our implementation and compare the results with other native XML-DBMS, like Software AG's Tamino [10] or Natix [18], relational DBMS with XQuery support and XQuery implementations like Saxon [16], BEA/XQRL [11] and Galax [9]. This could also lead to an analysis of strengths and should identify performance bottlenecks. Additionally, XML database connectivity should be concerned: Currently NaXDB uses JDBC and a next step will be to exploit a common XML repository API like the open source project XML:DB API [21] or the XQuery API for Java (XQJ) [13]. Further steps could lead to implementing advanced optimization techniques using views, static typing, indexes and statistics, which could be integrated in the existing architecture and concepts.

Acknowledgements

The authors would like to thank the rest of the NaXDB team: Anja Bog, Christian Braune, Kay Hammerl, Martin Probst, Johannes Scheerer and Lars Trieloff. We also thank Stefan Baier, Daniel Kirmse, Markus Özgen and Jürgen Primsch of the SAP Labs, Berlin for their encouraging support.

7. REFERENCES

[1] MaxDB by MySQL. `http://www.mysql.com/products/maxdb/`, 2005.

[2] A. Berglund, S. Boag, D. Chamberlin, M. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath`, Feb. 2005. W3C Working Draft 11 February 2005.

[3] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/`, Feb. 2005. W3C Working Draft 11 February 2005.

[4] M. Brundage. *XQuery – The XML Query Language*. Addison-Wesley, 2004.

[5] D. Chamberlin. Xquery: An xml query language. *IBM Systems Journal*, 41(4):597–615, 2002.

[6] A. B. Chaundri, A. Rashid, and R. Zicari, editors. *XML Data Management – Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

[7] J. Clark. The Expat XML Parser. `http://expat.sourceforge.net/`.

[8] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/xquery-semantics/`, Feb. 2005. W3C Working Draft 11 February 2005.

[9] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *Proc. of the VLDB 2003*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[10] T. Fiebig and H. Schoning. Software ag's tamino xquery processor. In *First International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, June 2004.

[11] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The bea/xqrl streaming xquery processor. In *Proc. of the VLDB 2003*, pages 997–1008, 2003.

[12] J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *Proc. of the 9th Int'l Workshop on Database Programming Languages (DBPL)*, Potsdam, Germany, Sept. 2003.

[13] Java Community Process. XQuery API for Java (XQJ) 1.0 Specification. `http://jcp.org/`, May 2004. Early Draft Review May 2004.

[14] H. Katz, editor. *XQuery from the Experts – A Guide to the W3C XML Query Language*. Addison-Wesley, 2004.

[15] M. Kay. XSLT and XPath Optimization. In *Proc. of the XML Europe 2004 conference*, Amsterdam, The Netherlands, 2004.

[16] M. Kay. Saxonica.com - Saxon XSLT and XQuery processor. `http://www.saxonica.com/`, 2005.

[17] A. Laux and L. Martin. XUpdate. `http://xmldb-org.sourceforge.net/xupdate/`, Sept. 2000. Working Draft September 2000.

[18] N. May, S. Helmer, C. C. Kanne, and G. Moerkotte. Xquery processing in natix with an emphasis on join ordering. In *First International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, June 2004.

[19] T. Parr. ANTLR, ANother Tool for Language Recognition. `http://www.antlr.org/`, 2005.

[20] SAP AG. MaxDB – The Professional DBMS. Available at: `http://www.mysql.com/products/maxdb/`, Feb. 2005. Version 1.0.

[21] K. Staken. XML:DB API. `http://xmldb-org.sourceforge.net/`, Sept. 2001. Working Draft September 2001.

[22] The Internet Society. HTTP Extensions for Distributed Authoring – WebDAV. `http://www.webdav.org/`, 1999.