Natix Visual Interfaces

A. Böhm*, M. Brantner**, C-C. Kanne, N. May, G. Moerkotte

University of Mannheim, Germany
alex|msb|cc|norman|moer@pi3.informatik.uni-mannheim.de

Abstract. We present the architecture of Natix V2. Among the features of this native XML Data Store are an optimizing XPath query compiler and a powerful API. In our demonstration we explain this API and present XPath evaluation in Natix using its visual explain facilities.

1 The Natix System

The Natix Project [3] was among the first to realize the idea of a native XML Data Store (XDS), which supports XML processing down to the deep levels of storage and query execution engine. Such native XDSs are now also being introduced by major database vendors [1]. Natix Version 2.0 provides most features of a native, enterprise-class XDS to application programmers, e.g. ACID transactions, efficient processing of XPath 1.0, and a rich set of APIs. Fig. 1 shows the modules contained in the Natix C++ library.

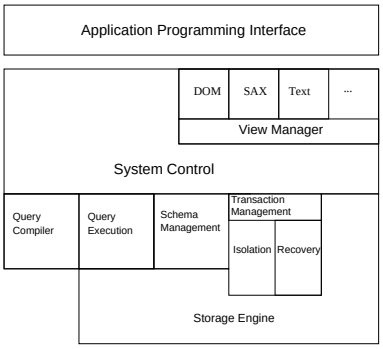

Fig. 1. Natix architecture

Applications use Natix' *schema management* facilities to organize their persistent XML data collections. The topmost organizational unit is the Natix *instance*. System parameters, such as main memory buffer sizes, are instance-specific. Both transaction and crash recovery operate at the instance level. Therefore, all operations of a particular transaction must be executed within the context of a single instance. Each instance consists of several *repositories* that contain documents of a particular application domain. For example, the product catalog of an online shop could be stored within one repository, while another one would be used for the business reports. A repository comprises *document collections*. Document collections represent an unordered set of related documents, typically having a similar structure and markup, although they are not required to conform to a conjoint schema. Applications use this hierarchy level for grouping documents together that are processed as a unit, for example, all items belonging to a particular commodity group of an online shop.

* This work was supported by Landesgraduiertenförderung Baden-Württemberg
** This work was supported by the Deutsche Forschungsgemeinschaft under grant MO 507/10-1

2 Application Programming Interface

Programming convenience, flexibility, and high performance are crucial for developing universal data management applications. Below, we describe the concepts that enable the Natix API to satisfy these requirements. Natix provides a variety of language bindings for the concepts. We present the C++ binding as an example.

2.1 Concepts

The Natix API [4] allows accessing and manipulating entities on all levels of the logical hierarchy through *request* objects. To perform a database operation, an application creates a request object and and forwards it to the system. Instance-level operations such as instance creation and destruction are performed by sending requests to the instance, while transaction-level operations are performed by sending them to corresponding transaction objects.

The API uses *fragments* as an abstract representation of all XML data that is handled by the Natix system. Fragments are an analogy to UNIX file descriptors, which provide a uniform interface to a variety of physical objects such as files, sockets, memory regions, etc. Natix fragments provide a uniform interface to a variety of XML data sources, for example documents stored in Natix, documents stored in the file system, entire document collections, or query results.

Another important concept are *views*. Generally, there are many ways to represent and access XML data, such as the DOM and the SAX API, each of them having their particular benefits and drawbacks depending on the requirements of the respective application. In order to provide maximum flexibility, Natix implements many different interfaces for accessing XML data. To access an XML data source through a particular API, the application opens a corresponding view on a fragment. The current Natix release includes, among others, views for both the DOM and the SAX API, a C++ stream interface, a document metadata view and various sequence views for iterating over elements contained in an organizational unit (for example, all documents of a particular collection).

The convenience and flexibility of the fragment/view concept is complemented by an efficient implementation mechanism. Natix uses a fragment/view matrix to obtain the most efficient implementation of a particular API for a given fragment type. For example, when accessing a document in the file system using a DOM view, a conventional parser is used to create a main-memory DOM representation. In contrast, if a DOM view is requested for a document stored in Natix, an object manager will make sure that only those parts that are actually required by the application are loaded from secondary storage, thereby reducing main memory consumption and avoiding unnecessary I/O overhead. As another example, if a SAX view is opened for a query result, the SAX events can be returned to the application in a pipelined fashion while the query is being evaluated.

2.2 C++ Language Binding

We will illustrate the Natix binding for C++ on the basis of the evaluation of XPath queries[1]. Two queries will be used as examples, one for selecting the book with the spe-

[1] The interface is capable of handling additional query languages such as XQuery or XSLT.

cific id (`//book[@id='2342']`) from a particular book collection, the other one for gathering all invoices that exceed a specific amount (`/invoice[sum(item/@price) > 200]`) from a document collection.

As proposed by the W3C, query execution in Natix is divided into two phases. We distinguish between a static and a dynamic evaluation phase. During the static phase, the query is compiled and prepared to be executed. A static query context is provided, which allows passing parameters to the compilation process. After the query is successfully prepared, it can be executed. A dynamic query context defines the environment for query execution, in particular the context item (e.g. a document or a collection). Figure 2 shows a few lines of code for executing a query on a single document. At the end, a DOMView provides a DOM representation of the requested book element. Note that the actual DOMNode object returned by the DOMView is a binary compatible instance of the C++ DOM binding of Xerces C++ [5].

```
 // start the transaction
Transaction trans(inst);

// create the query
QueryHandle queryHandle =
 CreateQueryTidy(trans,
                 "//book[@id='2342']");

// prepare the query
PrepareQueryTidy(trans, queryHandle);

// execute it and get a fragment
FragmentDescriptor queryResult =
 ExecuteQueryTidy(trans, queryHandle,
                  "store", "books",
                  "document.xml");

OpenViewTidy<natix::DOMView>
   domView(trans, queryResult);

xercesc::DOMNode *node=domView->getNode();
```

Fig. 2. Example for the query API

Executing the second example query on all invoices in one collection and opening a DocumentSequenceView would return an iterator for all qualifying documents.

For programming convenience, the Natix API makes use of several C++ language features such as automatic database resource deallocation when destroying the corresponding objects. In Natix parlance, all of the request objects used in the example are *Tidy* requests, which means that they free any resources when they are destroyed, for example if an exception has been raised.

3 Executing XPath Queries in Natix

Next, we give an overview of efficient and scalable XPath query compilation and evaluation. During this process an XPath query runs through different stages. Natix offers one visual explain facility for each of the following three steps of the compilation process.[2]

The result of the first stage — after parsing and normalizing the query — is an internal expression representation. It precisely describes the structure of a query, e.g. relationships between expressions or classification of predicates.

Continuing with this representation, our process departs from the conventional approach of interpreting XPath and enters the realm of algebraic query evaluation. For every expression we apply translation rules that yield an operator tree as a result. Fig-

[2] The tool used to trace the compilation process in the demonstration is also available online at http://pi3.informatik.uni-mannheim.de/xpc.html

ure 3 shows an operator tree for the first example query.

Besides special operators like unnestmap (Υ), this plan uses well-known operators such as a selection (σ), or aggregation ($\mathfrak{A}$). The detailed translation process is described in [2].

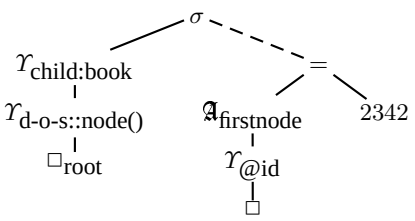

Fig. 3. Query evaluation plan

This *query execution plan* (QEP), which is a physical algebra plan, specifies detailed rules for evaluating the query. The last step is called *code generation* and produces code that can be evaluated efficiently in the *Natix runtime system* (RTS). The RTS implements a full-featured algebra as introduced in [2]. To provide scalability all sequence-valued algebra operators are implemented as iterators. The subscripts of these operators are implemented using assembler-like programs that are interpreted by the *Natix virtual machine* (NVM). These provide mechanisms to access the persistent representation of documents in the Natix page buffer or provide comparison and string functions.

4 Demonstration

Our demonstration consists of two main components: (1) We provide guidance of how to build XML applications and how to retrieve XML data using the various interfaces offered by Natix. Using several sample programs and a graphical user interface we demonstrate how to create and manage huge XML repositories. Opening views or executing queries shows a variety of possibilities for efficiently accessing the stored data. (2) We explain the internals of the XPath query compiler using the various graphical representations of query plans that are provided by Natix. Several example queries from, for instance, the XMark benchmark help to understand and interpret the facets of our XPath compiler after every compilation step using the different visual explain facilities. The resulting query evaluation plans provide deep insight into the Natix query execution engine.

The Natix XDS is available for download at [4].

References

1. K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. C. Truong, B. Van der Linden, B. Vickery, and C. Zhang. System RX: One Part Relational, One Part XML. In *Proc. SIGMOD Conference*, pages 347–358, 2005.
2. M. Brantner, C-C. Kanne, S. Helmer, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. ICDE Conference*, pages 705–716, 2005.
3. T. Fiebig, S. Helmer, C-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native xml base management system. *VLDB Journal*, 11(4):292–314, 2002.
4. Chair of Practical Computer Science III. *Natix Manual*. University of Mannheim, 2005. http://pi3.informatik.uni-mannheim.de/natix.html.
5. Apache XML Project. Xerces C++ parser. Project Web Site, 2002.