VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

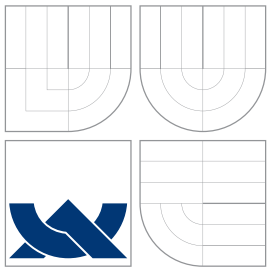
NATIVE XML INTERFACE FOR A RELATIONAL DATABASE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

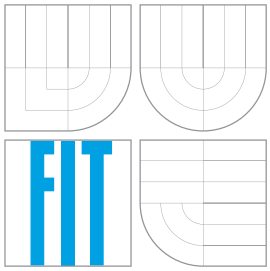
AUTOR PRÁCE
AUTHOR

Bc. KAREL PIWKO

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NATIVNÍ XML ROZHRANÍ PRO RELAČNÍ DATABÁZI

NATIVE XML INTERFACE FOR A RELATIONAL DATABASE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. KAREL PIWKO
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR CHMELAŘ
SUPERVISOR

BRNO 2010

Abstrakt

XML je dominatním jazykem pro výměnu dat. Vzhledem k velkém množství dostupných XML dokumentů a jejich vzájemnému přenosu, vzniká protřeba jejich ukládání a dotazování v nich. Jelikož většina firem stále používá systémy založené na relačních databázích pro ukládání dat, a často je nutné kombinovat nově získané XML data s původním daty uloženými v relační databázi, je vhodné se zabývat uložením XML dokumentů v relačních databázích.

V této práci jsme se zaměřili na strukturované a semi-strukturované XML dokumenty, protože jsou nejčastěji používanými formáty pro výměnu dat a mohou být snadno validovány pomocí XML schémat. Předmětem teoretického rozboru je modifikovaný Hybrid algoritmus pro rozdělení dokumentu do relací na základě XSD schémat a dále umožnujeme zavést redundanci pro urychlení dotazování. Naším cílem je vytvořit systém podporující nejnovější standardy, který zároveň poskytne větší výkon a vertikální škálovatelnost než nativní XML databáze.

Abstract

XML has emerged as leading document format for exchanging data. Because of vast amounts of XML documents available and transfered, there is a strong need to store and query information in these documents. However, the most companies are still using a RDBMS for their data warehouses and it is often necessary to combine legacy data with the ones in XML format, so it might be useful to consider storage possibilities for XML documents in a relation database.

In this thesis we focused on structured and semi-structured data-based XML documents, because they are the most common when exchanging data and they can be easily validated against an XML schema. We propose a slightly modified Hybrid algorithm to shred documents into relations using an XSD scheme and we allowed redundancy to make queries faster. Our goal was not to provide an academic solution, but fully working system supporting latest standards, which will beat up native XML databases both by performance and vertical scalability.

Klíčová slova

XML, perzistence XML v RSŘBD, Hybrid algoritmus, XSD mapování, XML:DB API, XPath LL(*) parser, výkonnost dotazování, redundance dat.

Keywords

XML, XML persistence in RDBMS, Hybrid algorithm, XSD mapping, XML:DB API, XPath LL(*) parser, query performance, data redundancy.

Citace

Karel Piwko: Native XML Interface for a Relational Database, diplomová práce, Brno, FIT VUT v Brně, 2010

Native XML Interface for a Relational Database

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Petra Chmelaře.
Uvedl jsem všechny literální prameny a publikace, ze kterých jsem čerpal.

.....
Karel Piwko
May 26, 2010

Contents

Chapter 1

Introduction

The XML [11] language has emerged as the most commonly used language for data description and information exchange nowadays. It virtually replaced all proprietary solutions used before. Obviously, these amounts of data must be stored and users would like to query them.

Data is usually stored in a database. Although, there are various ways how to define a database, we consider database being a storage and retrieval engine and we prefer properties bounded to traditional relation databases over high availability and horizontal scaling, as some of databases, for instance Apache HBase [3] or Bigtable [42], which are not object relational databases and which follow NoSQL movement.

We are convinced that relational databases are still the best solution for managing middle size data (up to 10 GiB) within a company, although the *one-solution-fits-all* might not be ideal in general [54]. However, there are more comfortable ways to work with data than using multiple SQL commands glued together with an arbitrary middleware, although even in this area things are getting easier for developers, such as new Java JSR-299 specification which will be a part of Java EE 6.

By all means, using REST (**RE**prestational **S**tate **T**ransfer) interface and XQuery [30] language, possibly combined even with XForms to get XRX [33], is much more comfortable to an average user[50]. However, we must often integrate legacy data with these relatively newly obtained XML documents and provide unified access to both of them. Naturally, storing the XML documents in a relational database seems the most appropriate solution.

1.1 Aim of the work

This work shortly describes XML, XML schemas and XML query languages, in summary an environment established around XML. Since we would like to persist its state, the means of storing and querying are described as well. The scientific research was led in late 90s and in the beginning of this century, such as documented in [53, 38, 39, 49, 35]. Therefore, we have chosen and modified already existing methods, with respect to our needs, that is the generality of the solution and its performance. A part of this work was developed as term project, namely skeletons for chapters 2, 3 and partial decisions of implementation details described in 4.

Storing XML documents in a relation database has a lot of advantages, such as mentioned in chapter 3. However, the reader of this work should get deeper knowledge of its disadvantages as well, that covers possible weak points of the architecture with respect

to his application domain. We are convinced that the description of implementation in chapter 4 will show mapping between two areas (the XML world emerging after 1996 with its hierarchy structure on the one side and the relational one, based on solid mathematical theory by E. F. Codd in 1970, on the other one).

Moreover, this work was not considered as a master thesis, but even as awesome opportunity to become a part of the open source community, represent the results to other developers and coordinate the work within tight time schedule. Because the work was already promoted as an open source project called *NeXD*[1], it is an ideal candidate to be developed even after the master's thesis is finished, improving its functionality either by author itself or by other developers from recruited from the community and the academic sphere.

1.2 Organization of the work

This paper is organized as follows. Chapter 2 describes in further details the XML language, its validation using XML schemas and languages used for querying XML documents. Section 2.3.1 represent the core for readers interested in query language used in the implementation. The following chapter (chapter 3) describes possible means of storing XML documents with focus on relational databases, detailing the method called *Hybrid*.

The largest chapter of the document, chapter 4, contains both description details and limitations of our implementation as well as fine grained explication of the code base with respect to the query language. The results are evaluated in chapter 5, which describes testing framework and comparison of performance to other established XML database. The conclusion is presented in the last chapter (chapter 6).

[1] See project page and source code at `http://gitorious.org/nexd`

Chapter 2

XML, XML schemas and query languages

The XML language was created in 1996, as a formal simplification and application of the SGML language [11]. Its aims are to be flexible, readable and independent of both character encoding and language. Shortly after its creation, XML language was massively adopted. In this chapter, we briefly describe the language itself and a document content classification, followed by means of XML validation. Then we will show how querying XML documents has evolved in time, focusing on current status and details needed for our implementation.

2.1 XML language and document types

It was mentioned before that XML language is nowadays de-facto standard for information exchange. The Internet itself is adopting XML more and more often, going for HTML5 and XHTML5 standards, using XML for Web Services and even frequent *Web 2.0 buzz-word*, AJAX, is an abbreviation which contains XML in its name. Lets conclude reasons which have led to massive adoption of XML language [50]:

Flexibility XML allows to describe arbitrary data structures, including recursion. It is not language or character encoding dependent and it can be used both by humans (visual representation) and by computers (raw data). XML itself can be used as a metadata language.

Validation Processing XML documents by a computer can be automated because we expect documents to be well-formed [11]. Additionally, further constrains on the document content can be posed by using an XML schema or even an composition of XML schemas.

Tooling support Virtually every programming language contains support for XML language. The documents can be parsed, transformed to another document instances, combined or queried. User can build an ecosystem based on XML, using XML only as an intermediate format.

Since XML documents have an arbitrary content, we classify them into three distinct categories [50], which are important for us because each category has different requirements for the persistent storage. Documents can be divided into *document-centric*, *data-centric* and *semi-structured* types, the last one being a subtype of the data-centric one.

2.1.1 Document-centric XML type

Document-centric XML instances have an irregular structure, data is represented by bigger fragments, such as paragraphs, pages or even the document itself. Elements are often mixed and nested, and their ordering is always important. Usually, these documents are intended to be read by an user and they are created manually. A perfect example of such document is a master's thesis written in DocBook [5]. We are not interested in this kind of documents for database storage, the reasons are explained in chapter 3.

2.1.2 Data-centric and semi-structured XML type

Data-centric XML documents have very regular structure in contrary, as they are usually intended to be processed by a computer. Data is represented by single elements or attributes on the level of atomicity. Very often, these documents are created as intermediate transport format between programs or companies. Any existing information, which can be divided to atomic elements, can be easily dumped into XML format. Science measurements are natural sources of these documents, with one (fictional) presented as 2.1 example.

Semi-structured documents have regular structure, however this structure changes in time frequently. The same information can be represented by multiple means, such as using different elements. Data can be very sparse, interleaved with metadata information, making the storage in relational databases challenging.

Source code 2.1: Cassini: an XML (data-type) document example

```
<nasa-data>
    <probe>
        <name>Cassini</name>
        <launch-date>
            <day>15</day>
            <month>October</month>
            <year>1997</year>
        </launch-date>
    </probe>
    <measure id="1234ABC">
        <distance>
            <value>1000</value>
            <unit>km</unit>
        </distance>
        <destination>Titan</destination>
        <data>
            <water>0.7</water>
            <albedo>0.23</albedo>
            <temperature>93.7</temperature>
        </data>
    </measure>
</nasa-data>
```

2.2 XML schemas

The need to specify and constraint information contained in documents had led to the creation of XML schemas. These usually restricts ordering and nesting of elements and

attributes, their data types and can even enforce element uniqueness and referential integrity [29]. The validation of an XML document is performed after the document is proven to be well-formed. This is usually an automatic operation, as most of XML parsers can be set to validate during the document parsing phase.

This section describes available schema languages, explaining their advantages, disadvantages and details important for document persistence. The most commonly used XML schema languages are DTD, XML Schema (written with the uppercase "S") and RELAX NG. The XML schema languages are:

DTD DTD (**D**ocument **T**ype **D**efinition [7]) has been created as a markup declaration language for SGML languages. This schema describes the document content by nested lists of possible elements and attributes. It does not allow further constraints on an element or an attribute (e.g. value type or length). The schema itself is not an XML document and it usually too general [47], so programmatic construction of a graph based on it is unnecessarily complex.

XML Schema XSD (**X**ML **S**chema **D**ocument [34], a W3C recommendation) defines documents as collections of elements and attributes, modeling relations of elements as well. Moreover, it defines types of elements and attributes, their default values and accepted content. XSD enforces mapping of the XML data types to the types of hosted language. For Java, these mapping are a part of JAXB (**J**ava **A**rchitecture for **X**ML **B**inding [13]) specification. It also allows to define order and to constraint repetition of elements by exact numbers. Additionally, XSD supports namespaces, so multiple schema definitions can be easily nested.

While validating a document against XML Schema, this schema must be referenced from the document for each namespace explicitly or namespace mapping must be supplied to parser. XSD is not only used for validation, it frequently serves for document or code generation. The latter allows XML documents to be represented as a primary object of hosted language (using JAXB), allowing programmer to use XML without any knowledge of it.

XSD has it disadvantages as well, namely the schema for document is very awkward to be read, it lacks mathematical background, does not support unordered content well and it is not consistent within its specification (for instance, the description between elements and attributes differ in XSD language) [18]. The other schemas, such as RNG, were introduced to overcome this problems, however we have chosen XSD due to solid Java support and tooling. Moreover, the most of research explained in section 3.2 holds for schemas based on XSD. XSD schema for document example 2.1 is presented as appendix B.1.

RELAX NG RELAX NG (**RE**gular **LA**nguage for **X**ML **N**ext **G**eneration [18]) or simply RNG defines documents as patterns, which are compared to elements in document instance. It is based on a formal theory of tree automata. RNG provides both XML and non-XML (compact, RNC) syntax for defining schemas, based on EBNF (**E**xtended **B**ackus-**N**aur **F**orm) and regular expressions, it aims to be simple for users with knowledge of regular expressions and tries to unify elements with attributes as much as possible. As well as XSD, RNG supports namespaces, data types and complex definitions. The research applied to XSD holds for RNG as well, because these schema languages can be transformed between each other with very few exceptions [23].

We wanted to use this schema language because it is much more readable and easier to learn, however the tools we have chosen for schema generation were not able to create RNG schemas without XSD intermediate step, which will obviously make document insertion unnecessarily slower. The RNG and RNC samples for the Cassini document (source 2.1) are presented as appendices B.2, resp. B.3.

Schematron Schematron [19] is rule based validation language, which asserts either presence or absence of an pattern in the XML tree. It is represented as mix of XML and XPath language, which is quite similar to XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformations). However, Schematron can constraint documents in way XSD nor RNG cannot. For example, it can control content of element by its sibling, or require parent element to contain specific content. Additionally, it can specify relations between multiple XML files. Moreover Schematron provides an easy way how to show application defined errors during validation.

Others To complete the enumeration, we list the obsolete XML schemas: XDR (**X**ML-**D**ata **R**educed), RELAX, TREX, SOX (**S**chema for **O**bject-Oriented **X**ML), DDML (**D**ocument **D**efinition **M**arkup **L**anguage) and DSD (**D**ocument **S**tructure **D**escription) and schema validation frameworks: DSDL (**D**ocument **S**chema **D**efinition **L**anguages), which contains both RNG and Schematron.

2.3 XML query languages

Two years after XML (that is in year 1998), there was already enough XML documents that W3C has discussed need of existence of a query language for Web, in particular for XML and RDF (**R**esource **D**escription **F**ramework) documents. Requirement for such language were identified from lots of proposals. Since W3C was developing another XML-aware specifications, such as XSL, XSLT and XPointer in the same time, the first standardized XML query language, XML Path Language 1.0, XPath [26], provides a common syntax between those specifications.

There are basically two types of operation when retrieving data from a persistence storage [47]. The first one, called *extraction*, in its pure way retrieves the whole document. Extraction can be combined with *selection*, to get a fragment of a stored document, so we have to be able to access its elements, attributes and values and then evaluate predicate conditions. The latter operation is called *querying*, it further applies transformations, sorting and aggregation on a result of extraction, or multiple extractions from different documents. Querying basically creates documents with XML schemas, which never existed in original documents.

The XPath 1.0 version, released in 1999, is very limited in its query abilities. It can select and aggregate nodes from the XML document tree and test the content of elements and attributes using very simple functions returning string, numeric or boolean values. The XPath language consists of sequence of steps, where each contains an axis, a node test and optional predicates. Each step selects a subtree from the current context. In example 2.2 we can see both of XPath syntaxes, an unabbreviated one, which contain all of three elements (axis `self`, node test `destination` and predicate `[.='Titan']`) as well as abbreviated one (operator `//` actually represents `/descendant-or-self::node()`). The results of this query is shown as example 2.3. We will explain the axis meaning in section 2.3.1 considering XPath 2.0 language.

Source code 2.2: XPath 1.0 expression on the Cassini document - query

```
//measure[id='1234ABC']/self::destination[.='Titan']/preceding-sibling::distance
```

Source code 2.3: XPath 1.0 expression on the Cassini document - result

```
<distance>
    <value>1000</value>
    <unit>km</unit>
</distance>
```

2.3.1 XPath 2.0 language

As it was said before, XPath 1.0 language is quite limited. Up to few exceptions, we can retrieve results by instantiating an XML as a DOM tree, and then execute steps in the sequence they were defined. Each step use previous step result as its own DOM tree. The limitation has led to specification of XPath 2.0 language, in 2007. The XPath 2.0 (henceforth "XPath") is much larger than its predecessor, and even some basic concepts of the language such as its data model and type system are changed. XPath is in fact a subset of XQuery 1.0 language, sharing XDM data model. The language provides backward compatibility mode, which enforces the same behaviour as the older version of the specification, however its availability is implementation dependent.

The XDM (W3C **X**Query 1.0 and XPath 2.0 **D**ata **M**odel [31]) is a data model for XPath, XSLT 2.0 and XQuery. The model defines all permissible values of expressions and valid inputs for an arbitrary language processor. All mentioned languages are closed with respect to XDM, that means any evaluated expression is part of XDM as well. XDM was created in order to support following features: Support of XML Schema types, representation of document collections, complex values, atomic types and ordered, heterogeneous sequences (n-tuples). The data model does not specify concrete binding to any programming language, is simply states what information must be accessible.

XDM models document as a tree of items. Item is either an atomic value (a primitive type, e.g. `xs:string` or type derived from primitive one) or a node (document, element, attribute, text, namespace, processing instruction or comment). Each item has a content and a type assigned. The list of available types with relations between them is shown in figure 2.1.

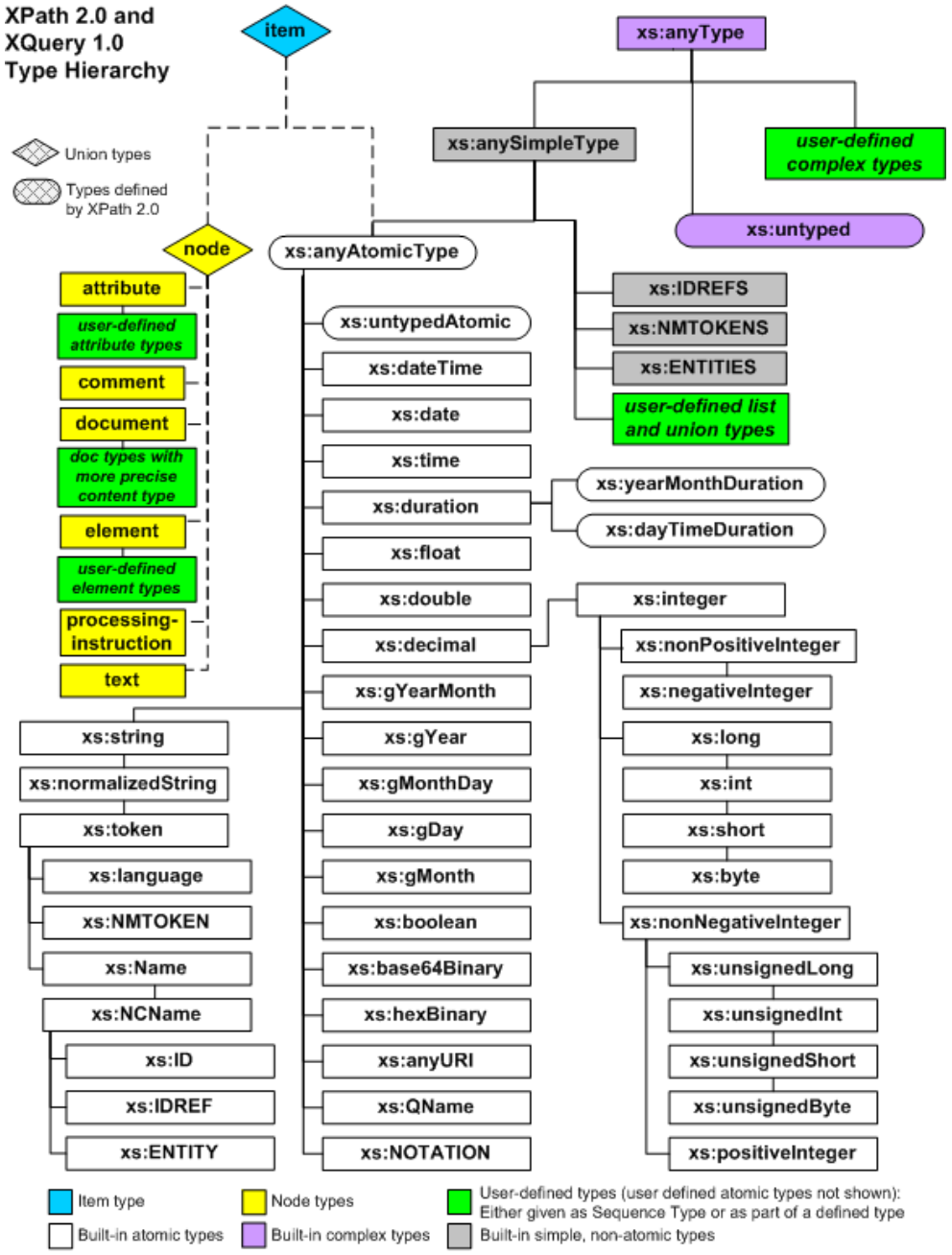


Figure 2.1: XDM model types hierarchy, original source at [31]

In our implementation, we map atomic XDM datatypes to SQL ones. Because XDM uses XML Schema, we can validate documents against XML Schema(s), shred them to generated relational tables and store in the persistent layer. The query languages use XDM model as well, which provides us advantage while reconstructing documents. More details are provided in chapter 4.

XPath language allows processing of XDM conforming values. The evaluation of expression is always a sequence allowed by XDM. The main distinction between XPath versions 1.0 and 2.0 apart from underlying data model lies in existence of expression context, which

affects the query evaluation. The context can be divided into two parts:

Static context Static context contains information available during static analysis, prior to expression evaluation. It contains many elements, we mention statically known namespaces, default element, type and function namespace, in-scope schema definitions, in-scope variables, which are used for evaluation, and statically known documents and collections, which act as the input source if no other source is defined.

Dynamic context In contrary to the static context, the dynamic one is available during expression evaluation. If evaluation relies on a part of dynamic context without assigned value, implementation raises an error. The context consists of many elements, such as context item (the definition of item from XDM holds), context position and size, variable values, function implementations, available documents and collections and the default collection.

XPath expressions are evaluated in two phases, static and dynamic one. The first one basically prepares input XDM instance by resolving function names, variable names and namespace specific information from static context, while the latter uses dynamic context to assign values and creates an output sequence. The resulting XDM sequence is returned to the client *as-is* or transformed by serialization to a string.

The grammar of XPath language is vast and path expressions, which have actually given language its name, are only a part of it. The language defines a limited FLWOR expression explained in section 2.3.3. We will enumerate path expressions because implementation details provided in section 4.3 require their knowledge by reader.

XPath 2.0 path expression extends sequence of steps as known from XPath 1.0. Step domains are larger and filter expressions using arbitrary condition over dynamic context are added. Their EBNF definition is available in appendix C.1. The axes available in XPath are summarized in tables 2.1 and 2.2 [27].

Table 2.1: List of XPath 2.0 forward axes

Axis name	Explanation
`child`	Represents all children of the context node. Only document and element nodes have children. The child of node can either be an element, a processing instruction, a comment or a text node.
`descendant`	Represents transitive closure of the child axis, that is all descendants of the context node.
`attribute`	Contains the attributes of the context node. This axis is allowed only for elements.
`self`	Contains node itself.
`descendant-or-self`	Contains context node itself and its descendants.
`following-sibling`	Contains children of the context node's parent that occur after the context node in document order.
`following`	Contains all nodes which occur in tree defined by root node, are not descendants of context node and occur after context node in document order.
`namespace`	Contains namespaces of the context node. This axis is deprecated, and should be used only if backward compatibility mode is enabled, otherwise static error should be raised.

Table 2.2: List of XPath 2.0 reverse axes

Axis name	Explanation
`parent`	Contains parent of context node, empty sequence if the node has no parent. Attribute nodes can have a parent as well, their enclosing element node.
`ancestor`	Represents transitive closure of the parent axis, that is all ancestors of the context node.
`preceding-sibling`	Contains children of the context node's parent that occur before context node in document order.
`preceding`	Contains all nodes which occur in tree defined by root node, are not descendants of context node and occur before context node in document order.
`ancestor-or-self`	Contains context node itself and its ancestors.

After the axis is evaluated and appropriate nodes are selected, a node test is executed on each item of resulting sequence. The node test basically match name (of the element, attribute etc., depending on the axis type) or a wildcard represented by `*`. In XDM, names consist of namespace prefix and local name delimited by `:`, called QName (**Q**ualified **N**name). A wildcard can match either both the prefix and the local name, just the prefix or just the local name.

The filtered nodes which fulfilled conditions posed in previous step, are then matched against predicates. In XPath, the predicate can be an arbitrary expression including nested

queries. This makes the evaluation quite difficult to implement. According to XPath specification, predicates are evaluated from left to right against an *inner focus*, the context item for the current predicate evaluation (that is one of the filtered nodes). Nodes are then sorted according to the document order (if a forward axis predicate is used) or the reverse document order (for reverse axis predicates); they maintain the document original order otherwise. The possibility of a reverse ordering during evaluation does not alter ordering of the results. The example of XPath query (check if some measure in the document is further than 1,000 km) with multiple predicates is present as sources 2.4 and 2.5.

Source code 2.4: XPath 2.0 path expression with multiple predicates on the Cassini document - query

```
(: this is a comment, we check the distance of measure :)
some $d in //distance[value][unit/text()='km']/value satisfies $d > 1000
```

Source code 2.5: XPath 2.0 path expression with multiple predicates on the Cassini document - result

```
false
```

2.3.2 Complex XPath expressions

The complex expression can either pose a requirement on the position in the document, such as operator << does. Therefore, we have to index positions of elements in files. We propose DLN (Dynamic Level Numbering [41]), and store this information additionally within tables. The advantage of DLN is that it can be easily stored in bit vectors, thus compared rapidly and this method of indexation can cope even with updates and removals. DLN is based on Dewey Decimal Classification, which is a sequence of ordinals and delimiter characters. We can use it to index XML documents of arbitrary lengths, including streamed and unbalanced ones.

The DOM graph with DLN classification for a part of input document 2.1 is shown in figure 2.2. Further details, especially about how updating documents modifies DLN indexes can be found in [41, 50].

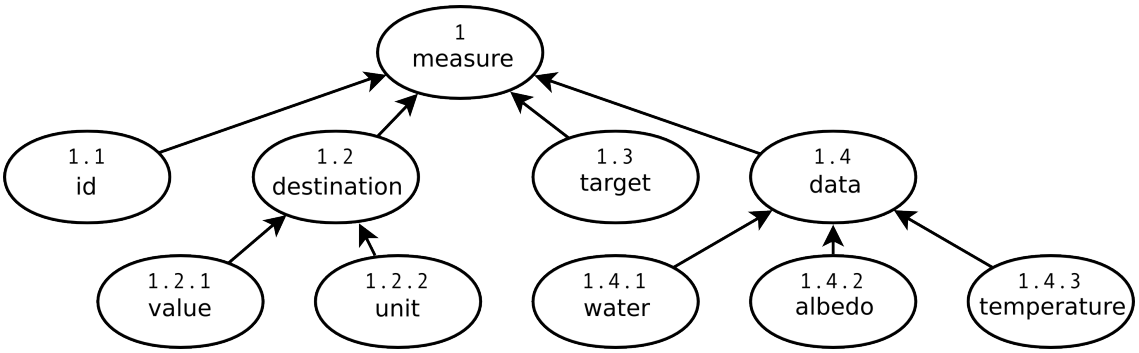


Figure 2.2: DLN indexation example

2.3.3 XQuery 1.0 language

XQuery 1.0 (henceforth "XQuery") represents a superset of the XPath language. It was designed as language in which queries are easily understood, concise and versatile enough to handle the broad world of XML. The usage of the same XDM model often leads to a single XQuery/XPath parser, because any expression syntactically valid in XPath can be executed in XQuery 1.0 and must yield the same results. XQuery binds each expression two contexts (dynamic and static ones), whose content is very similar to the one described in 2.3.1.

For the purposes of selecting an input, XQuery (as well XPath) defines following functions:

- `fn:doc`, with an argument representing URI of the document in available documents; and
- `fn:collection`, with an optional argument representing URI of the collection in available collections, without arguments it returns the default collection.

As mentioned before, the core of XQuery expressions lies in the FLWOR expression, which supports iteration and binding of variables to intermediate results. This can be used for joining multiple input sources and transforming data. The abbreviation FLWOR (pronounced "flower") stands for `for`, `let`, `where`, `order by` and `return`.

The `for` and `let` clauses generate a sequence of tuples, consisting of bound variables. The sequence is filtered by a `where` clause, which is optional as well as an `order by` clause, which enforces the filtered sequence order. Results of the query are transformed according to a `return` clause and then returned as an ordered sequence (that is, either in the document order or in its reverse). The source example 2.6 returns average temperature on Titan, a satellite of Saturn, based on Cassini document[1].

Both `for` and `let` clause can contain more than one variable binding. Although they seem very similar, the manner how variables are bound differs. The `for` clause iterates over the sequence using the `return` clause as a cycle body. Contrary, the `let` clause creates a sequence, with value accessible under a bounded variable. A very nice example, which illustrates the difference can be found in section 3.8.3 of W3C XQuery 1.0 recommendation available as [30].

Source code 2.6: XQuery 1.0 query for the average temperature on Titan - query

```
for $d in fn:doc("cassini.xml")/nasa-data
let $e := $d/measure
where $e/destination = 'Titan'
return
    <titan>
        {
        <measured-by>{ $d/probe/name }</measured-by>,
        <temperature>{ fn:avg($e/data/temperature) }</temperature>
        </value>
        }
    </titan>
```

The result of the query is a completely different document (shown as source example 2.7), still a valid instance of XDM model.

[1] Averaging one value is nonsense, but according to XML schemas, the document can contain more measures, which is not the case of the sample.

Source code 2.7: XQuery 1.0 query for average temperature on Titan - result

```
<titan>
    <measured-by>Cassini</measured-by>
    <temperature>93.7</temperature>
</titan>
```

Even such a powerful language as XQuery has its limitations and it is still developing (there is currently XQuery 1.1 processed by an W3C committee). The main restriction of the XQuery language is that it does not allow updates. This is addressed by XQuery Update Facility 1.0 [32], which introduces new types of expression allowed for XQuery, extensions to its processing model, static context and minor grammar updates. However, XQuery Update facility, though partially supported in our parser, was not a subject of the work, as further explained in chapters 3 and 4.

2.4 Summary

While working with the XML language, due to its flexibility, we often have to validate input documents. There are multiple XML schemas, which are used for this validation. Before the validation itself, the document must be verified to be well-formed. Because validation means automatic processing, we will bind to the validation life cycle and use it to persist XML documents, as it is explained in the next chapter.

XML query languages, especially XQuery 1.0 are very versatile, due to the XML structure. They can be used to query multiple documents, documents hierarchies and/or collections, which make them natural languages for obtaining data from XML databases.

Chapter 3

Persistence of XML schemas and XML documents

In the previous chapter we discussed XML and XML schemas and we have chosen the XSD schema. In the current chapter, we will describe the way how XML documents[1] can be stored in a persistent data warehouse. We focus on available approaches of their storing in a relational database. Most of the means were developed during late 90s, and their usage was and maybe still is considered obsolete by production XML databases. Since community is quite sceptic about the performance achieved by any of these methods, we will try to convince them that we can pick one of these methods (the Hybrid algorithm, to be precise) and by a few modifications we can obtain the implementation with quite promising results.

This chapter is organized as follows. Firstly, we provide the short categorization of available means, including the categorization of storage methods based on the document type. We narrow the possibilities: relational databases, which are the core of our work in section 3.2, later we focus on schema-driven methods, detailing the Hybrid method in section 3.3.1. To conclude, we will discuss the strong and weak points, which occur once we allow database modifications by XQuery language extensions in sections 3.4 and 3.5.

3.1 Storage means for XML documents

XML documents can be stored in multiple ways either in a specialized storage, e.g. modeled as a graph structure and stored into graph or specialized databases. The latter case, for instance *eXist* [10], usually creates huge number of indices. Another approach is to consider XML documents structured enough for application purposes and store them in a key-value based storage and/or use some kind of a middleware to provide XML querying functionality. Obviously, the native approach seems to be the most effective, because storage techniques are tailored to fulfil needs of XML tree structure.

The choice of best available technique depends on purpose of the application. Either *a*) we store XML documents as whole; *b*) we use XML only for data transfers, so we store only values which it contains; and *c*) we use an XML data model.

The XML document-type documents are ideal candidate to be stored using the first case. Documents are stored in a key-value storage, file system hierarchy or as objects in

[1] We do not discuss storage of XML schemas in a relational database and retrieval of the schema from relational schema created by the process described in this chapter. Nevertheless, an XML schema in the XSD form is an ordinary XML document, which can be stored by the same way as the other XML documents.

a relational database. Typically, whole documents will be retrieved in most queries and cache query will be hit during an evaluation of smaller fragments. Stored documents are expected to be kept intact or being updated rarely. Additionally, XML schema generated is either too complicated or the suspect to change frequently [47].

The second case is a perfect use case for relational databases. Firstly, for each document we will determine its XML schema, which is later used to create an appropriate relational schema. The document is divided into fragments, which correspond to the relations to be stored (*shredding*) and these are inserted in the database. When querying data, the query must be translated into series of SQL queries and the result is reconstructed (*publishing*). This expects that XML schemas are changing rarely and they can be easily mapped using an algorithm presented in section 3.2. We don't have to preserve original document's structure and in general, the structure of the extracted documents differ.

Lastly, the third case models data in a XML model, which is the most convenient way how to store semi-structured XML documents. XML model represents data as well as hierarchic, relational, object and other models, the closest similar model is the hierarchical one. Typically, there is a ordered direct acyclic graph (DAG) with typed and named inner nodes and unnamed leaves for storing data themselves. The XML model must contain at least the order of elements in the document, their attributes and PCDATA (**P**arsed **C**haracter **DATA**). There is no limitation on persistent storage, so model can be stored either in relational, hierarchic or object-oriented database, as a collection of indexed files or in another proprietary format [40].

We consider storing XML documents in a relational database, which provide us following advantages:

- Technologies used in the relational databases have been developed for a long period of time and the are considered mature, effective and covered by a solid mathematical background;
- Relational databases provide transaction (more precisely, ACID transaction) support and allow multi-concurrent access with a locking scheme.
- Interface provided and programming language support of RDBMS is huge.
- Relational (legacy) and XML content can be easily mixed in one application.
- Relational databases can be used as source for data-mining much easier than XML documents, although there is a research related to information retrieval in XML data warehouses as well [52].

On the other hand, we have to cope with following drawbacks, diminished if we sacrifice some properties of RDBMS (to be more precise, properties associated with the design of RDBMS-based applications, such as data redundancy):

- We have to develop techniques of storing the XML tree structure in relations.
- We cannot scale horizontally easily [55], or with enormous costs, which makes our system unusable in the environment where high-availability is required.
- We have to establish a way of querying data in a relational database and reduce the price of XML fragment reconstruction.

For our system, we will use XDM stored in a relational database. The XDM model was described in section 2.3.1. However, our system will rather be a hybrid between cases *b*) and *c*), because we focus on data-type documents, but at the same time, we have to track relations between elements in source documents, that is to provide the same functionality as native XML databases. For querying mixed elements, we will rely on XML-awareness of the underlying RDBMS and we introduce specialized storage structure for mixed elements.

3.2 Storage of XML data in relational databases

As explained in the previous section, we choose the storage method according to the document type. We will omit the whole document method, as it will be awfully slow during query processing and it will limit maximum document size to available memory, since every document is supposed to be retrieved from database and parsed into a DOM or XDM tree respectively while evaluating the query. The remaining methods can be divided into three categories, readers interested in further details can follow [49].

3.2.1 Generic methods

Generic methods do not use any schema of document, since it is not guaranteed to exist and/or it can change very often. The relational schema must be either created general enough to handle all types of documents or usage of this storage is restricted to a limited set of documents.

The first type representative, *Generic tree mapping* models XML document as a DAG, assigning unique identifier to each inner node, leaves carrying either element or attribute content. Relations between elements are mapped by named edges, where name stands for the element (or attribute) name. We have different means of storing this DAG in a relational database, edges can be stored in different tables according to their type (element, attribute, inner) or in one universal table. A leaf table can be created for each datatype or leaves are backed by one super table. The universal approach creates lots of `NULL` values, whereas the fine-grained table leads to numerous joins, causing a low performance of Generic tree mapping [53].

When modelling a DAG, we can use even an algorithm called *Structure-centred mapping*, which maps nodes instead of edges. The relations between nodes are represented by a list of children, each node consists of a type, a name, a text content and a list of children. It allows database to traverse and reconstruct an arbitrary document using depth-first-search traversal. However, the approach poses constraints on the node identification, instead of simple number, dynamic level numbering should be used to allow updates.

Describing unlimited generic methods, we mention *Simple-path mapping*, which was an ideal mapping for documents queried in XPath 1.0 language. This mapping stores an XPath for each node, including the position and the order in the DOM tree. Then we can directly map XPath operator to SQL ones, however updating documents and retrieving their fragments will be costly operations.

The last generic method representative *Table based mapping* generates a database schema, which exactly fits the document. This is ideal for data transfers, however too much limiting for our purposes.

3.2.2 User-defined and user-driven methods

In these methods, user manually creates a mapping between an XML document and tables in a relational database. This is no doubt the most flexible method and furthermore the easiest one to be implemented. However, this approach is interactive and user is required to be skilled both in XML and relational databases to yield the most efficient schema mapping.

User can generate the schema either completely or with the aid of a middleware, using declarative mapping, such as annotations in XML documents. The latter approach is called *user-driven mapping*. It provides reasonable default fixed mapping and ability to influence it. User simply selects fragments whose storage methods will differ and defines how they will be stored within available mapping bounds. The method used in Mapping Definition Framework [35] or in XCacheDB[37].

3.2.3 Schema-driven methods

Relation schema is a transformation from either an existing XML schema or the XML schema is generated from document sample(s). The relation schema can be further fine-grained and optimized by various methods. This area will be the main focus of proposed work.

However, the way how to classify the methods is not standardized. Another classification is presented in table 3.1, adopted from [47].

Table 3.1: Alternative classification of XML to RDBMS mapping

Approach			**Exploited information**
Fixed (schema-obvious)			XML data model
User-defined			Purely user-specified mapping
Schema-driven	DTD-driven XSD-driven Constants preserving		XML schema
Adaptive	Cost-driven	Single-candidate Multiple-candidate	XML schema, XML documents and queries
	User-driven	Direct Indirect	XML schema, annotations

3.3 Schema driven mapping

Schema driven mapping tries to generate an optimal relational schema by using the following concept: For each DOM element generate a relation, which contains attributes and element content. Mappings between DOM nodes are represented by database primary and foreign keys. Schema driven mapping tries to overcome limitations of relational schemas derived from ER diagrams based on elements present in the document, which likely leads to the excessive fragmentation.

During the description of this approach, we focus on data-type XML documents and XSD schema. It must be said, that the most of schema-driven transformations have these limitations and therefore require an enhanced functionality of underlying relational database:

- Identity constraints of the schema are usually mapped to constraints on relational tables, but a relational table contains just a subset of the whole XML document, where the schema constraint is valid.
- Wildcards enable storing of an arbitrary element at place of the definition in the schema and thus it can be stored only in a general data type column of a relational database, which must obviously be XML-aware to process wildcard elements.

Methods driven by a schema traversal can be divided into two categories, the *fixed* one or the *flexible* one [36]. The former is based on XML schema only while the latter uses more than one XML schema for a set of XML documents and evaluates speed of sample queries performed by relational database. Further information of query evaluation can be found in sections 3.5 and 4.3.

For both of the methods, XML schema of a document can be further simplified and transformed up to the following constrains are met [53]:

- Any document conforming to the XML schema can be stored in resulting relational schema.
- Any XQuery executable over XML document can be executed in relational database instance.

The basic idea of the schema simplification is to follow repeatedly a three types of transformation, that is *a*) flattening structure (e.g. inlining elements into their parents); *b*) reduction of unary operators to the single one; or *c*) grouping sub-elements (e.g. optimizing of sub-element possible count for parent element) [53]. In document [38] was further shown that grouping sub-elements into more than one group (more precisely, for `1..*` ER mapping, create groups of size `1` and `*`) can leverage performance depending on the statistic distribution of the sub-element. Dealing with elements groups make processing of a schema more difficult, but does not involve any limitations of getting this done.

According to the simplification mechanisms, we classify following fixed methods, which are [46]:

Basic The Basic method creates a relation for each element in the table, allowing any element to be the root. The children nodes are inlined into the parent table for every possible case (that is all except wildcard and recursion descendants). This can lead to the creation of multiple relations for single element, if used in the XSD differently.

Shared On the other hand, the Shared method tries to map each element only once. It generates inlined elements in the same way as the Basic algorithm does. However, it stores elements in standalone relations only if they satisfy the degree of the possible appearance or they are ancestors of a wildcard node. The conditions are presented in document [53]. The Shared method is able to process recursive element definitions by creating the relation for only one of the mutually linked elements.

Hybrid The Hybrid method combines Basic and Shared method, to reduce the number of created relations. The method is described in section 3.3.1.

Constraints preserving mapping This mapping generates an extended entity-relational diagram. It models elements and complex data types as an entities, an attributes. Relations between elements and their degrees of presence are represented as cardinalities. Other meta information extracted from XSD (for instance, data types) are mapped as well.

3.3.1 The Hybrid method

The Hybrid method was proposed in [53]. It is based on the Shared method, however it additionally inlines elements, which match given conditions: *1*) their in-degree (number of edges coming into the elements in the DAG representation) is greater than `1`; *2*) they are not recursive; and *3*) they are not reached through a wildcard node. The Hybrid method works for both ordered and unordered documents and according to [47], it can achieve fourth normal form (4NF) decomposition into relations.

The method is considered as the best fixed methods, because it reduces fragmentation of the document and when reconstructing elements, it is on a par with the Shared method considering number of required joins and SQL queries. Because Hybrid inlines more than Shared, it has lower number of joins but greater number of queries. Therefore, we postpone the execution of the queries (more details in section 4.2) until the longest possible one is constructed and thus we have lower number of required inner SQL joins, reducing state space during query evaluation.

The Hybrid method was created for generation of relational schemas based on DTDs, which do not contain so much information about elements as XSDs, which NeXD is using. We modified it not to inline elements which are complex, including its nested child. This reduced the time of schema traversal during its generation by removing unnecessary recursion descent. The schema generated for Cassini document is shown as source 3.1.

Source code 3.1: Relational schema generated for the Cassini document, PostgreSQL dump

```
                Table "public.nasa-data_1.nasa-data_0"

          Column          |         Type          |    Modifiers
--------------------------+-----------------------+-------------------------
 __id                     | integer               | not null default
                        nextval('"schema_1.nasa-data_0___id_seq"'::regclass)
 __document_id            | integer               |
 __encoding               | character varying(15) |
 probe.name               | text                  |
 probe.launch-date.day    | integer               |
 probe.launch-date.month  | text                  |
 probe.launch-date.year   | integer               |
 measure.destination      | text                  |
 measure.distance.value   | integer               |
 measure.distance.unit    | text                  |
 measure.data.water       | numeric(10,6)         |
 measure.data.albedo      | numeric(10,6)         |
 measure.data.temperature | numeric(10,6)         |

Indexes:
    "nasa-data_1.nasa-data_0_pkey" PRIMARY KEY, btree (__id)
Foreign-key constraints:
    "nasa-data_1.nasa-data_0___document_id_fkey" FOREIGN KEY (__document_id)
        REFERENCES xdocument(id) ON UPDATE CASCADE ON DELETE CASCADE
```

The example identified the problem of each schema-basen mapping, that happens if input document does not state explicitly multiple occurrence of the elements. The source 3.1 thus contains only one table, because NeXD had no clue multiple elements `measure` can

occur. This means that user have to carefully choose the first document stored in database, because it actually generates schema for the rest of `nasa-data` documents. If the document is not appropriately chosen, the other documents can be rejected considered not valid.

3.4 Shredding XML document into relations

After a relational schema is generated, we have to divide the documents into such pieces, which represent records in created relations. This operation is called shredding. The main question is whether to allow the storage of documents not conforming to given schema. We have basically four possibilities:

1. Allow the storage of these documents by dynamically creating database relations based on their schema;
2. Store parts of documents not conforming to the schema using general tables – we call them *junk* tables;
3. Reject the documents as not being valid for our storage schema; and
4. Transform documents on-the-fly to the schema in our database.

On the one hand, accepting the non-conformant documents can allow us being more general but on the other hand, it will greatly augment the complexity of queries because unions of results gained from different schema mapping are required. Further, it is difficult for an user to query data non-conforming to the schema, because he is simply not aware of their existence.

When queried data is an input for another processing tools, elements not defined in the XML schema unnecessarily leverage complexity of these tools. As the result, we prefer rejecting not-conforming XML documents with the detailed description why that happened. This allows user to visualize them and verify the need of invalid data or to convert them easily.

The most appropriate approach would be to let an user define if the not conforming part of document must either be transformed or cut off and the rest of document stored in the database. Since the insertion of data into RDBMS should be non-interactive, this part of data preprocessing is not performed automatically and user is recommended to clean data himself. We recommend to use VisualXML, which was presented in [43]. Once the documents are transfered to the valid schema, they can be inserted into our system seamlessly.

However, simply avoiding the documents not conforming the XSD schema does not solve the whole shredding problem. Still, mixed elements of the document must be stored. XSD marks mixed elements explicitly by `mixed` attribute. Therefore, NeXD introduces a special table which can be used for storing text within mixed elements called `xtext`. This table contains text parts, which are bound to the enclosing element. The implementation details can be found in the following chapter, namely in sections 4.2 and 4.6.

3.5 Retrieving and modifying XML data

NeXD was from the very beginning considered a fast storage system. XML query languages specified in section 2.3 work on XDM model, which is created on-the-fly from the relational

database. In this section, we describe possible approaches of speeding up the model creation, with respect to the performance.

The two fundamental operations – extraction and querying (described in section 2.3) have different time and space complexity characteristics when the Hybrid method is used. When an XML document is shredded into relations, the extraction is extremely difficult and time consuming operation. To overcome the problem, we allowed data redundancy. Thus, we additionally store complete XML document in RDBMS as a `CLOB` object. This way, we are able to preserve additional data, such as comments and processing instructions, and speed up the query as well, because it becomes basically a simple `SELECT` operation. It would be nice to provide the same data redundancy for other frequently retrieved and relatively large XML chunks.

We expect our storage system to be used mostly for retrieval of information, which are rather static and do not modify over time much. The naïve way how to change parts of the document is to construct a new document, then to generate its XML schema, shred it into relations by the Hybrid algorithm, wipe out the original data and insert the new document. This way, we make the change at document level granularity. This approach keeps the consistency of the database, but is highly inefficient.

Next, if we allow a modification of the schema by the previous operation, we will eventually end up with lots of nearly empty tables, junk tables, mixed content or we will shade modifications by another data structure. This dilutes all the advantages provided by the sophisticated shredding and hurts the performance as well, so we have decided to limit the update of the data to document level. By all means, our system aims to be a storage with fast retrieval of either XML chunks or whole XML documents and fine grained query evaluation optimizations would raise its complexity significantly.

3.6 Summary

In this chapter, we described means of storing XML documents, focusing on relational databases. Since there are multiple ways how to use a RDBMS as the persistence storage, we classified available methods into generic, schema-based and user-driven, resp. user-defined categories. The schema-based methods seem the most promising for our implementation, so we followed with their description, more detailed for Hybrid method. Once the method is chosen, the documents are shredded to generated relations, with respect to the facts included in this chapter. We concluded the chapter with an introduction to the document retrieval. The next chapter shows how these methods are implemented.

Chapter 4

Implementing NeXD

This chapter is the core of master's thesis. It describes issues and pitfalls of the current implementation, as well as its advantages. However, it is not a listing of source code snippets with comments, for a reader interested in this kind of information please read Javadocs and even better the source itself, but it provides deeper description of relations between technologies and modules used in NeXD.

We describe the shift performed from the older implementation, the choice of the build tool and the emphasis on unit and integration testing in section 4.1. The description of XML query languages in chapter 2 will be used in section 4.3, considering the XPath parser, enriched with internal details. Section 4.2 explains how the Hybrid algorithm is implemented.

The chapter continues with constraints posed on underlying RDBMS in section 4.4. Once we have described all parts necessary for the implementation, we will follow with the selection of the API used for data storage/retrieval in section 4.5. The chapter is concluded in by a code overview in section 4.6.

4.1 Selecting the build tool

NeXD has become a project, which will be used in various environments, therefore we have chosen *Maven* [15] as the build management system for it. Maven is a well established tool in Java build process, used virtually by all important players on the Java EE market, such as Oracle/Sun, JBoss by Red Hat, IBM, BEA, Apache Software Foundation, Springsource and much more others.

It automates not only the building of the software itself, but even the testing (such as smoke and unit tests) and (continuous) integration testing. Additionally, Maven uses a concept of repositories, which are simply public servers including various packages, libraries and Maven plugins (together called *artifacts*). There are services, which provide a free of charge creation of the repository, thus making usage of NeXD easier for all users, who would like to include its functionality in their own project.

The concept of plugins allows appending an arbitrary functionality to the existing project. For example, the project files satisfying the contract for Eclipse [8] can be generated just by executing command `mvn eclipse:eclipse`[1]. This will establish a project metadata including all the library dependencies as well as their source and Javadoc if desired and activated by properties `-DdownloadSources=true`, resp. `-DdownloadJavadocs=true`. Obviously, there are plugins to generate the same for IDEA IntelliJ, NetBeans and JBuilder.

The IDEs themselves should integrate Maven automatically.

However, using Maven didn't only provide us the benefits. There were problems with internal dependencies, which were not yet *mavenized*, that is their developers didn't produced Maven artifacts yet. Fortunately, there are ways how to store an artifact in a local repository, which is used to obtain the artifact during dependency resolution phase. We provided a bash script distributed with NeXD to overcome this issue.

We focused on continuous integration to make our project rock stable, so *TestNG* testsuite together with *maven-surefire-plugin* was used to test NeXD automatically before each commit or even better after each code modification. The plugin will provide nice HTML (and XML as well!) report. To execute the testsuite from scratch[2], simply execute the commands printed in block 4.1.

Source code 4.1: Running NeXD testsuite

```
# Getting NeXD source code
git clone git://gitorious.org/nexd/nexd.git
# Install artifacts to local maven repository
cd nexd/notmavenized
./mvn-install-files.sh
cd -
# Executing testsuite
cd nexd
mvn test
```

The list of supported values, which can be used to launch NeXD or TestNG testsuite, defining the database database of NeXD, is summarized in table 4.1.

NeXD contains approx. 80 testcases, going through the parser and both implemented APIs (see section 4.5). We expect that NeXD code is not totally covered by unit tests, so it would be nice to append the *EMMA* [9], a free Java code coverage tool, preferably as a Maven plugin configuration to measure the coverage and identify weak points of the implementation. This is one of the improvements, which will be surely added during the project evolution.

[1]It is possible that the Eclipse workspace might not be initialized to contain the variable that links to local Maven repository. This can be easily solved by executing another plugin goal, called `eclipse:configure-workspace`.

[2]User is expected to have installed both *Git* and Maven 2.x. The first run can take a long time, since Maven must download all artifacts for its run and establish the local repository. The size of the testsuite can be reduced by modifying TestNG[20] file present in *src/test/testng* directory.

Table 4.1: Java system properties accepted by NeXD

Property name	Default value	Explanation
`nexd.dbName`	`nexd`	The name of the database on the machine used as the connection point. NeXD expects the right database schema including *root* collection. Obviously, any collection can be selected lated by its URI, but the `root` one is necessary. The testsuite creates the right database schema automatically.
`nexd.host`	`localhost`	Either the hostname or the IP address of machine where PostgreSQL database is running.
`nexd.port`	`5432`	The port number of PostgreSQL service.
`nexd.userName`	`nexd`	The name of user with fully granted access to the database specified above.
`nexd.password`	`test`	User's password.
`nexd.loginTimeout`	`10`	Time in minutes when the credentials are hold in memory in case of inactivity. Use 0 in case of long-running transactions.
`nexd.useSSL`	`false`	Specifies if SSL should be used to connect to the database. This depends purely on setting of the underlying database.
`nexd.sslFactory`		The name of factory, which is used to verify the SSL certificate against an certification authority. Set it to `org.postgresql.ssl.NonValidatingFactory` if you are connecting to the machine that has self-signed certificate or certificate not signed by a CA registered within your Java environment.

4.2 Implementing the Hybrid method

The Hybrid method in NeXD creates a relational schema from an XSD document. The XSD document is either delivered together with the input document in a place where JAXP parser can found it; or, which is more usually the case, generated on-the-fly by *Trang* [23]. Trang is an open source tool, which is able to convert different schema types and generate a schema from an XML document instance. It tries to create human readable schemas, which have led to minor advantages during the implementation.

NeXD modified Trang to be able to use in-memory representations of the schema output. This was necessary to remove an intermediate step, which required generation of the XSD to the hard drive and then parsing it again to obtain the XML schema. Moreover, the implementation of Trang didn't allow the usage of XSD schema stored in our database tables, which was a major problem.

The Hybrid algorithm implemented in NeXD, requires metadata tables, which store information about created tables in the system. Moreover, we implemented NeXD to use a different namespace for each created schema, so the required tables are (complete database

schema is present as source E.1):

XSCHEMA XSchema table represents the generated schema in NeXD. Schema encapsulates the private namespace for tables created by the Hybrid algorithm. Schema is identified by `name`, which consists of the XSD root element name, and a database generated `id`. This way we can easily bound document to the schema with respect to the root element.

XAPI (see section 4.5) enforces the possibility of multiple schemas for documents of in one collection (to be precise, XAPI does not define type of documents in the collection at all). However, NeXD limits this to the schemas of unique names, because of shredding capabilities. This limitation has impact on number of allowed documents, once the XSD is generated for a document type, all documents that have the same root element but differ from the XSD stored in the same collection are always rejected.

XTABLE XTable encapsulates table metadata, by providing information about element present in XSD. Every table contains parent schema, collection, table, flag whether the table is inlined and sets of possible elements and attributes. The metadata is used for the quick querying of the content. XTable contains a pointer to the table generated by the Hybrid algorithm. The way how data is queried is further described in section 4.3.

However, SQL types are more general than XSD ones. We provided a mapping between domains, which is enumerated in source file `XDM.java`. The part of the mapping is present in table 4.2.

Table 4.2: Mapping between XDM and SQL data types

XDM type	SQL type	XDM type	SQL type
`xs:string`	`TEXT`	`xs:NCName`	`TEXT`
`xs:Name`	`TEXT`	`xs:QName`	`TEXT`
`xs:ID`	`VARCHAR(100)`	`xs:IDRef`	`VARCHAR(100)`
`xs:integer`	`INTEGER`	`xs:positiveInteger`	`INTEGER`
`xs:int`	`INTEGER`	`xs:negativeInteger`	`INTEGER`
`xs:byte`	`SMALLINT`	`xs:base64Binary`	`BLOB`
`xs:long`	`BIGINTEGER`	`xs:boolean`	`BOOLEAN`
`xs:decimal`	`DECIMAL(10,6)`	`xs:double`	`DOUBLE PRECISION`
`xs:float`	`REAL`	`xs:date`	`DATE`
`xs:dateTime`	`TIMESTAMP WITH TIMEZONE`	`xs:gDay`	`VARCHAR(2)`
`xs:duration`	`TEXT`	`xs:gMonthDay`	`VARCHAR(5)`
`xs:time`	`TIME WITH TIMEZONE`	`xs:anyURI`	`TEXT`

SQL datatypes can further be restricted by using integrity checks, such as `CHECK x > 0` for `xs:positiveInteger`. However, these checks are not necessary, because we don't allow data modification and the validation against XSD is performed during resource storage phase.

The last problem to be solved for the Hybrid algorithm is the storage of mixed elements. NeXD can identify them in XSD schema and provide special treatment of their content for the shredding phase.

4.2.1 Algorithm life-cycle

The Hybrid method is implemented in source file `XMapper.java`. It uses Trang generated XSD, which is transformed using JAXP to a DOM tree. However, the DOM structure, representing XML as a graph, does not directly match the requirements of the algorithm, because element can contain references to other nodes.

NeXD implements DOM traversal using a standard `TreeWalker` interface, which is a part of W3C DOM 2 specification [6]. However, JAXP does not ensure that the available XML parser implements this functionality. Therefore, NeXD has its own implementation, which comes from jStyleParser [14].

The tree traversal is used to navigate in the document tree. The algorithm performs recursive descent, using name or complex type attributes as references. The first step creates an empty set of traversed elements and an empty map of created tables. NeXD identifies the root element from XSD. The root is used as the *current* element. The recursive traversal can be simplified to:

Algorithm 4.1 The Hybrid algorithm traversal

```
1:  traversed ← traversed ∪ {current}
2:  if current represents XML element then
3:    if current is complex type then
4:      table ← process current as complex
5:    else if current is simple type then
6:      table ← process current as simple
7:    end if
8:    for ∀child of current do
9:      process child with current context table
10:   end for
11: else if current represents XML attribute then
12:   column ← create column from current
13:   add column to table
14: else
15:   for ∀child of current do
16:     process child with current context table
17:   end for
18: end if
19: return tables
```

Hybrid uses following conditions to match that element is complex: *i*) Either XSD element is of `xs:complexType` type or it represents an XML element or an attribute which contains only one child, which is of `xs:complexType` type; or *ii*) the XSD element represents a typed element, which is not generic and that is complex; or *iii*) the XSD element references a complex type element.

In a similar fashion, the element is considered simple if: *i*) Either XSD element is of `xs:simpleType` type or it represents an XML element or an attribute, which contains only one child, which is of `xs:simpleType`; or *ii*) the XSD element represents XML element or attribute typed with generic type (a subset of generic types was provided in table 4.2); or *iii*) the XSD element represents typed element, which is simple; or *iv*) XSD element references the simple type element.

The main difference in processing simple and complex elements in algorithm 4.1 lies in the fact, that simple elements are either represented as standalone tables or they are added to the current table as columns, whereas the complex elements, if their occurrence lies within allowed bounds, can be inlined to the current context table. However, both simple and complex elements can lead to the creation of the standalone table, as allowed by the Hybrid method presented in section 3.3.1.

The Hybrid method, using described kind of traversal, however, has following limitations:

- The selection of the root element is quite simplified, because we do not construct a DAG to find the root element, but we use Trang generated schema to deliver the root element. Trang intends to generate human readable XML schema, which ends in the generation of an XSD root node which matches the Hybrid root node definition as well. However, this behaviour was verified only on a set of tested documents and it is not formally verified.
- Due to the recursive descent, it is difficult to process elements with circular dependencies, which usually occur if XSD contains complex recursive definitions.

4.3 XPath processing

NeXD implements a XPath 2.0 language parser for querying data, as already stated before. In this section we focus on implementation details of the query processing. XPath is quite complicated language and according to details presented in 2.3.1, we describe here the evaluation of path expressions. The evaluation itself can be divided into three distinct parts, which are: *i*) parsing and validating query; *ii*) binding static and dynamic contexts; and *iii*) converting result to an XDM instance or its serialization to a string. We describe the processing of simple XPath path expressions, because they are more illustrative and, in general, their execution flow is the same as for complicated ones. To process an XPath expression, NeXD must load metadata from the database.

4.3.1 NeXD metadata

As it is mentioned in the previous section, the Hybrid method uses `XSCHEMA` and `XTABLE` metadata tables. However, these are not only metadata tables required. NeXD must store information about documents and encapsulating collections. In this concept collection represents a set of documents. However, a collections can contain other collections as well, so in the database collection structure is modeled as a tree. Each XSD document is bound to the XSD schema. Relations are illustrated in figure 4.1. NeXD caches the metadata in memory, making subsequent queries faster by skipping the metadata loading phase.

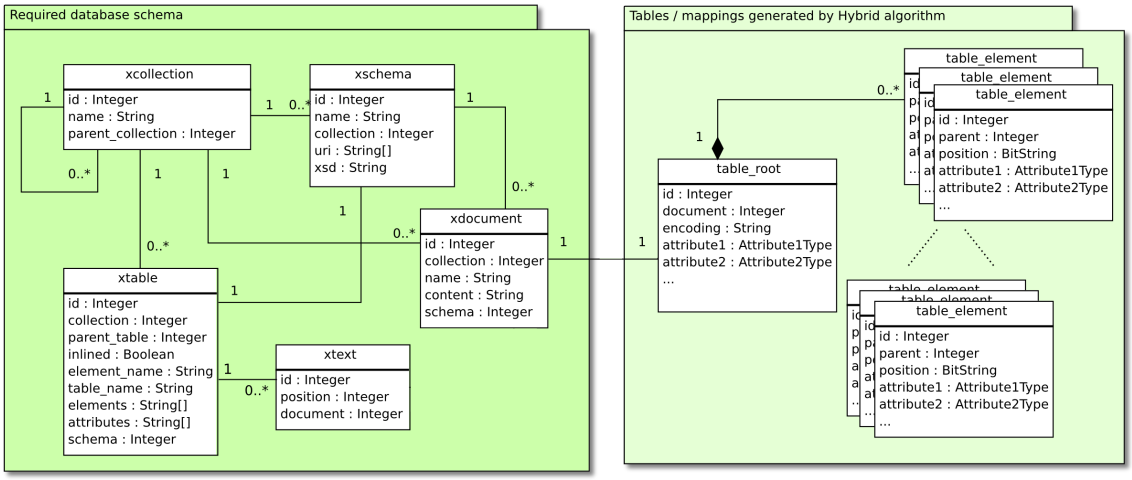


Figure 4.1: Metadata tables used in system

The tables used for metadata are (including tables already presented before):

`XDOCUMENT` XDocument represents the document content as it was stored in the database. Table further contains document identifier, generated by NeXD and used for managing documents in collection and the document name.

`XCOLLECTION` XCollection encapsulates documents in sets, modelling hierarchy.

`XTEXT` XText is used for storing text of mixed elements, linking them to enclosing parent table element and storing their relative position with respect to the parent element.

`XSCHEMA` XSchema table represents the generated schema in NeXD. Schema encapsulates the private namespace for tables created by the Hybrid algorithm. Schema is identified by `name`, which consists of the XSD root element name, and a database generated `id`. This way we can easily bound document to the schema with respect to the root element.

XAPI (see section 4.5 enforces the possibility of multiple schemas for documents of in one collection (to be precise, XAPI does not define type of documents in the collection at all). However, NeXD limits this to the schemas of unique name, because of shredding capabilities. This limitation has impact on number of allowed documents, once the XSD is generated for a document type, all documents that have the same root element but differ from the XSD stored in the same collection are always rejected.

`XTABLE` XTable encapsulates table metadata, by providing information about element present in XSD. Every table contains parent schema, collection, table, flag whether the table is inlined and sets of possible elements and attributes. The metadata is used for the quick querying of the content. XTable contains a pointer to the table generated by the Hybrid algorithm. The way how data is queried is further described in section 4.3.

4.3.2 Parsing XPath language expression

The language parsing core is based on LL(*) type grammar, which describes XPath 2.0 together with part of XQuery 1.0 language. The grammar is written in ANTLR (**AN**other

Language **R**ecognition **T**ool [2]) parser. This tool was chosen because it allow an easy binding between Java and the grammar itself and author's previous experience with it (jStyleParser, a part of CSSBox toolkit).

NeXD uses ANTLR both for parsing XPath expressions and parsing URIs of XAPI collections, as specified in section 4.5. The ANTLR parser can process an arbitrary textual input or, using a high level abstraction, an abstract syntax tree (AST) representation of the input. NeXD uses the latter way, that is combining input lexer and parser to generate a stream of AST nodes, which is then parsed using high level structure definitions.

The approach has drawbacks, since the query must be actually parsed twice, it is more time consuming. Fortunately, the speed is compensated by an easier parser modification, better ability to recover for errors, thus providing nicer error messages and possibility to change lexer implementation on-the-fly. Although, this is not used in NeXD, since XQuery is not implemented, usual approach combines XML and XQuery parser, because FLWOR expressions can contain arbitrary XHTML constructs.

The speed reduction can be balanced by caching of pre-compiled expressions. This possibility is proposed by XQJ API, however it is not implemented. The parser is represented in file `XParsingContext.java`. It allows program to use an arbitrary node of XPath grammar as entry point (starting symbol) and provides an automatic connection of the low-level parser with the high-level one. The choice of entry point is extremely useful. Apart from testing, it allows to switch between XPath path expression support to the full XPath 2.0 support or even XQuery 1.0 (if implemented) by simple code modification, making the parser very universal tool.

4.3.3 Binding contexts

When creating an XPath expression representation in NeXD (see remarks in section 4.5), we have to access database metadata created during the application of the Hybrid algorithm. However, mapping between XDM and SQL data types is not a bijection, so we have to store the actual datatype during query evaluation. This information is stored in the dynamic context, since it differs for each evaluation.

NeXD in actual implementation doesn't support binding external variables for they are not supported by the implemented API. So, static binding consists of selecting the collection according to the URI used to connect to the database and making it available as default collection during path expression execution. This way, we follow required API contract and we are able to evaluate the XPath 2.0 subset without having to implement the whole specification XQJ specification.

4.3.4 Retrieving XDM instances from database

A part of the dynamic context, called the evaluation context, consist of an actual XPath step, an actual table and intermediate results. Evaluation context is used to chain execution of the SQL commands. It is modeled by class `XContext.java`.

The actual XPath step must be mapped to metadata, precisely to `XTable`s available in the system. Since tables are organized in a hierarchical structure in the database, we can narrow the selection of appropriate document fragments even before touching the database content itself simply by determining if such path can be reached in metadata extracted from documents. Every step is implements its SQL command fragment, which return intermediate nodes, entry points for the next steps.

The biggest problem lies in axes, which are transitive closures or require a document to traverse its structure. Obviously, we can't mirror relations between elements in shredded document to the schema, because one element can be used in multiple positions in the document and so the fact that a relation exists between two relational tables does not guarantee the same relation exists for the current step and the evaluation context.

The evaluation of steps in sequence provides lower number of SQL joins, but the higher number of SQL commands required to execute the query. This is expected behaviour, such as it was identified while performance of Hybrid was compared to Shared and Basic in [53].

Before an XDM instance is created, the table structure determined for the XPath axis is used to impact following SQL query which selects elements from the database. The query differs based on the inline flag:

- For inlined tables the query must be executed immediately, narrowing the possible elements.
- for standalone tables we can append SQL `WHERE` clause which will filter elements.

The result of the query is used for two purposes: *1*) To construct the XDM instance, if the step is the last one in the path expression; or *2*) To identify the root of execution for the next step. The first case becomes complicated if the last step retrieves an element, which is not a leaf node in the graph representation, because transitive closure of child relation must be retrieved from the database as well. The DOM tree, which represents an XDM instance, is constructed from nodes which are transformations of relational database records. This way, NeXD execution chain prefers selection of atomic values instead of whole document fragments.

The latter case represents the core of the NeXD retrieval system. Intermediate results are transformed to a sequence, and for each item the rest of XPath step expression is executed. The entire execution chain is shown in figure 4.2

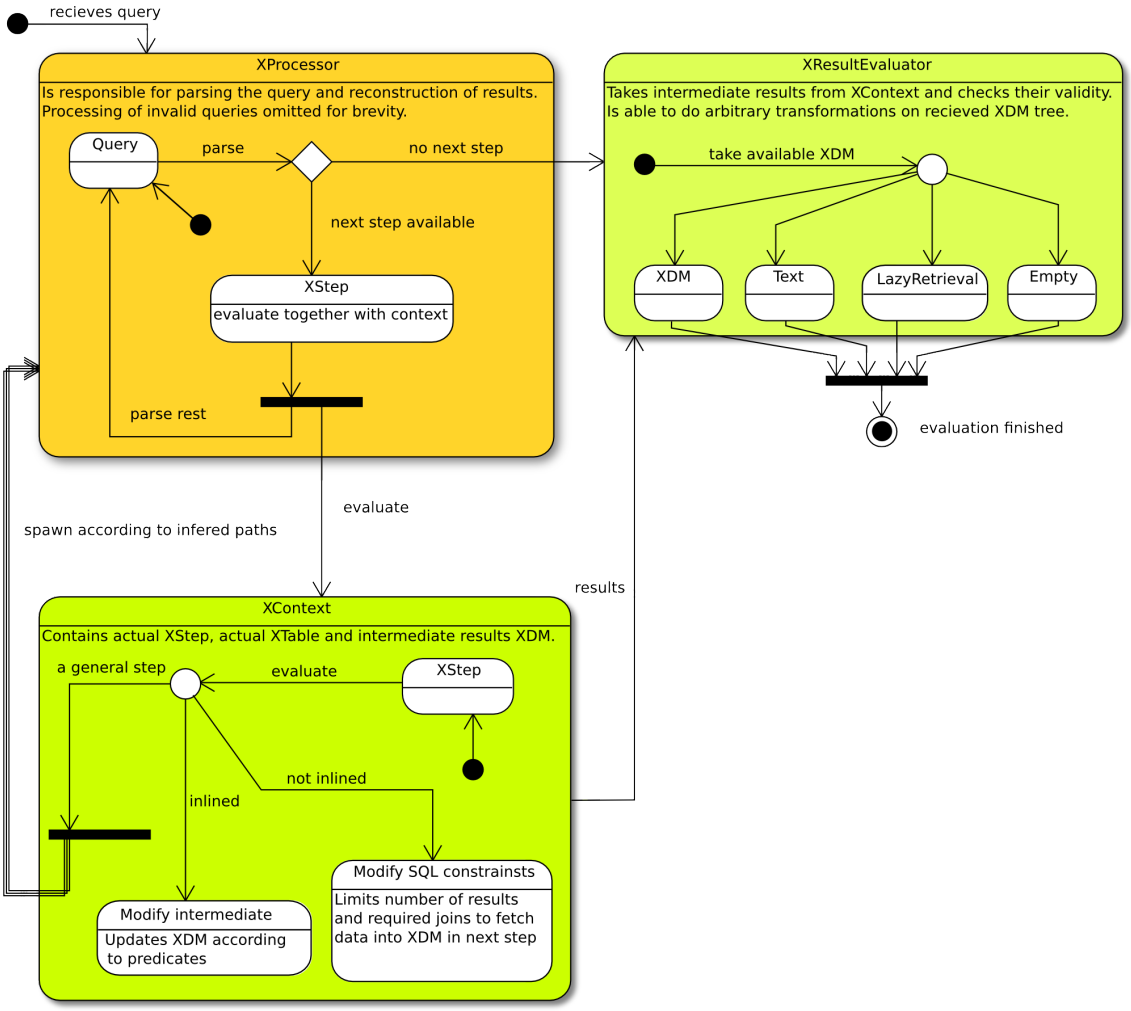


Figure 4.2: Process of XDM instance retrieval

4.4 Selecting the underlying database

NeXD is the implementation of native XML database over a relational one. In this section we explain what RDBMS we have chosen, what are the limitations of our solution and what functionality is required from the underlying relational database.

NeXD aims to be an open source project, so while selecting the database we wanted to use an open source RDBMS as well[3]. The selection was narrowed both by the required functionality as well as production versions installed, namely NeXD was required to run on the PostgreSQL [17] 8.3.x database instance installed on server `minerva2.fit.vutbr.cz`. Moreover, we consider PostgreSQL the most advanced open source database.

The query functionality relies on a raw JDBC with precompiled SQL commands. We identified following restrictions on the database side:

[3]Surprisingly, the code of the relational database has proven useful while searching for limitations of the schema naming, as explained in [48]. The schema name is limited to 63 characters using lower part of UTF-8 table.

- Database must support *schemas* or other means how to nest table names under an unique namespace. Schemas are supported quite well in PostgreSQL 8.x. PostgreSQL allows different schemas in the database to be owned by different users. Additionally, database implicitly creates schema called `public`, which is not defined in the SQL standard, and it is implicitly used when no schema is defined. The SQL standard actually defines schema based on user name and do not force its implementation to include support for different namespace names.

  This limitation can be easily overridden in the Hybrid algorithm processing, explained in section 4.2, by using generating table names with prefix based on a XSD root element name. Our implementation uses the XSD root element as a part of schema name. However, the element name in XML can be in Unicode, schema name only in a subset of Unicode. This can lead to rejection of documents containing elements named in other than Latin alphabet.

- Database must support a procedural language with triggers. This requirement is enforced by the previous one, since we are dropping whole namespace once the row from `Xschema` table is deleted. Such a trigger written in PL/pgSQL language is a part of the thesis as source E.2.

- The code poses additional restrictions on JDBC driver, the connection bridge between database and Java code. The driver must support following functionality:

  1. Creating the `ARRAY` of given type using `java.sql.Connection` factory method for arrays of Java (primitive) objects. NeXD stores a lot of the metadata in arrays on the SQL side.
  2. Driver must support `Datasource` implemenation for connection pooling and proxying physical connections.
  3. The JDBC driver should support `getGeneratedKeys()` method, which returns `ResultSet` with all columns representing the values obtained from sequences or other generators. However, the JDBC driver for PostgreSQL 8.3 didn't support the functionality, so the code was eventually rewritten to form of `RETURING` inserts statements. `RETURNING` is an extension of PostgreSQL, even if the same concept is used in Oracle database.

- The database must support SQL/XML specification from SQL 2003 standard. This is used to reconstruct content of mixed elements from database relations.

- The database must hold the ACID (**A**tomicity, **C**onsistency, **I**solation, **D**uration) constraints.

Our implementation was tested against PostgreSQL 8.3 and 8.4. NeXD expects to run under user who has enough rights to create schema, database with installed PL/pgSQL language extensions and a authentication based on user name and password, although *indent* method should work as well.

NeXD can be ported to an other database, which holds the constraints. However, it does not support any database abstraction layer, and all SQL statements are written in the PostgreSQL dialect. Thus the migration represents revisiting the SQL statements (not necessarily all of them) and binding the JDBC data source properties to the properties supported by NeXD.

4.5 Selecting the supported APIs

Currently, there are two APIs which are standard for XPath/XQuery enabled storage systems within Java. The first one, XML:DB API was proposed to be implemented in the formal specification, the latter one XQuery for Java is newer and its draft was finalized after the master's thesis specification was created. In this section, we shortly describe both APIs and show what parts we have implemented in NeXD. Java also defines the third API, which was not considered since it does not allow processing richer languages than XPath 1.0. JAXP (**J**ava **A**PI for **X**ML **P**rocessing [22]) focuses rather on XML document parsing, XSLT transformations and DOM model then on query functionality. JAXP is in fact used to retrieve a platform available XML parser for processing XML documents and XML schema representation in NeXD.

4.5.1 XML:DB API

XML:DB API (XAPI [25]) was designed as a common access to XML databases. It allows applications to store, retrieve, modify and query data in the database. The API claims to be equivalent with technologies such as JDBC or ODBC.

XAPI is very modular and allows vendors to implement functionality beyond the specification. The specification is based on core levels, which show what parts of XAPI had been implemented. These are:

Core Level 0 This is the minimum level to claim the conformance. Database must implement API base and `XMLResource` modules; and

Core Level 1 Additionally contains `XPathQueryService` module

NeXD implements Core Level 1, however the XPath language version specified by XAPI is unclear. NeXD implements a subset of XPath 2.0. It the base API consists of definition and abstractions of collections, resources (documents or their parts) and their sets. `XPathQueryService` allows (originally) usage of XPath 1.0 against the database. NeXD additionally provides `DocumentLoaderService`, which can store any resource defined by URI. This can be used for example to allow your application to store documents using REST protocol, which is quite easy to be implemented using a third party library, however, it is not present in the current version of NeXD.

However, implementing XAPI in the application has several flaws. First, XAPI was defined in 2001, and it wasn't clear in some points important for vendors, furthermore the draft wasn't standardized. This has led to situation, where general interoperability is hard to achieve. Therefore, NeXD used rather the modification of XAPI proposed by eXist, which additionally allows running the testsuite in an easier way. Second, it simply seems that nobody forces XML:DB API evolution and the project is virtually dead. Therefore, we have decided, that having support for another API will be very convenient.

4.5.2 XQuery API for Java

XQuery API (also known as XQJ, Java Specification Request Java 255 [21]) is a generic data access framework, which provides a uniform interface for XQuery implementations in the Java language. Applications using XQJ can execute queries, bind data and process query results. XQJ, as an enterprise specification, provides support for J2SE 1.4 and its goals can be stated as following:

- Ensure consistency with XQuery 1.0 specification.
- Provide access to any XQuery data source.
- Create a simple API, which may resemble JDBC, which is already familiar to many developers.

NeXD implements XQJ up to a limited part, it just provides a way how to obtain a data source, collection and execute a query. However, since the functionality of XPath was reduced to simple expressions, we do not implement external data binding and sophisticated result processing. We do not support a precompiled expressions at the moment. The precompiled expressions will skip ANTLR generating phase. Since `XQConnection` class already caches expressions, we can cache expressions for longer periods, which may result in significantly faster execution for repeated queries.

NeXD uses XQJ, which has a better implementation for XQuery processing, as a system wide API for querying. XAPI, is additionally able to control the database itself, that is to control creation of (nested) collections and to store data to the database. The connection between two APIs seems promising, as it virtually allowed as to have very limited XQJ support for free, and overhead caused by injecting XQJ into XAPI is very limited.

The bridge is implemented in `XPathQueryService`, mentioned in the previous section. The service simply wraps XAPI Collection to its XQJ equivalent, injects the collection to the XQuery static context default collection. This ensures there are available data in the collection bounded to the service only. Additionally, the bridge didn't force any modification of XQJ contract, apart from reduction of XPath expression power, which was done anyway.

4.5.3 Relational access

NeXD allows modification of the data directly by accessing the relational tables by any mean convenient for relational databases. However, user must be aware that the modifications are not automatically reflected to the document cache, which may lead to inconsistent query results. Moreover, because of redundancy, it user modifies the atomic relation data, for instance any row in a table representing a Hybrid generated element, it will end up with stale data in `XDOCUMENT` table.

Therefore, the only operation advised by an unexperienced user is removing a schema from the database. By deleting a line from the `XSCHEMA` table user removes database schema generated for this schema, including nested tables and documents as well, because the deletion triggers the cleaning trigger.

4.6 NeXD code overview

The NeXD code is mavenized, so it follows Maven's typical contract for Java project. That is, in the root directory checked out from the Git [12] repository, you will find only Maven Project Object Model (POM) file `pom.xml` and directory `src`, which contains subdirectories with own source code (`main` and test source code directory `test`). As an add-on, the directory contains `notmavenized` directory, which contains artifacts not available in public Maven repositories and an installation script for their installation.

The original code base from [45] was massively rewritten. Moreover API comments were translated from Czech to English, to make project useful for the community. This was an extremely tiring task, but at least the traversal provided me deep knowledge of the system,

both its benefices and limitations. Nowadays, NeXD API is considered stable. It includes following packages, considering namespace prefix `cz.vutbr.fit.nexd`:

common Contains both files shared by XML:DB API and XQJ API and utilities;

map Contains generator for XSD from XML documents, shredder to fragments and XDM (re)constructor;

xapi Implementation of XML:DB API;

xqj Partial implementation of XQJ; and

xquery Parser of XPath language, generated by ANTLR.

NeXD was compiled and tested on various versions of Java 6 (Sun/Oracle 1.6.0_17, IBM 6 SR7, OpenJDK 1.6.0_18), all of them 64bit versions. We considered backward support for Java 5 as well, but since Java 5 entered *end-of-life* phase in 2007 and even *end-of-service-life* in 2009, this was considered an extra work not worth of it.

Chapter 5

Evaluation of performance

As was stated in chapter 1, NeXD aims to be a fast query engine system with the support of querying by an XML query language, that is to act as a native XML database. During the implementation we described strong and weak points of the implementation, so we have to verify the theoretical outputs with the real based data measured on current implementation.

The measuring performance of the application in not an easy task. Firstly, we have to decide which parts and which kind of benchmark we are interested in and then we have to find means, which perform the measuring itself. Following evaluations were found interesting for NeXD:

- Storage time for various XML document, including the complex schema generation, exercising Trang and the Hybrid method components.
- Ratio between the real size of document and the space used in NeXD.
- Memory requirements for processing of documents.
- Query time, including different means of serialization.
- Clustering performance.

NeXD contains TestNG unit tests, which act both as smoke and functional tests. NeXD contains an interface, which allows easy testing of the query time, the results are presented in current chapter. Benchmarking of XML query languages is quite difficult nowadays, because the standardized testsuite does not exist yet. Sure, benchmarks do exist for particular application scenarios, but there are no standardized specifications [44]. Despite the facts, there exists an excellent service, called XQBench [1], which allows measurement of XML query performance, however it does not support other language than XQuery. Still, this would allow us using a subset of XQuery to test the performance. Alas, our limitation, which bounds XAPI collection to the dynamic context is not portable and thus this frameworks could not be used.

The real performance of NeXD depends on the ratio between select and modification queries. No doubt, because we allowed redundancy of the data, we preferred the faster selection. In this version, we do not supporting XQuery Update Facility 1.0, XUpdate or any other mean of updating data except direct change by an SQL command. The complexity of an update operation will raise with respect the to size of the document and the impact of the modification on it. XUpdate Facitily will be added in a future version.

5.1 Testing framework

We have decided that query time was the most important characteristic of our system. We have selected and identified XPath queries, which evaluates most of the NeXD functionality. These queries were executed on the testing system described in appendix A. Queries we tested are summarized in table 5.1, 1,000 documents from [16] were used as the test input.

Table 5.1: Test queries used to evaluate NeXD performance

Name	XPath query
Q1	`/weather`
Q2	`/weather/head/locale`
Q3	`/weather/dayf/day[1]/part/wind`
Q4	`/weather/dayf/day[1]/part/wind/*`
Q5	`//wind`
Q6	`//cc/wind/*`
Q7	`//part/wind`
Q8	`//part/wind/*`

5.2 Comparing performance of NeXD with other databases

We compared the query performance on NeXD and eXist 1.4, rev 10440. eXist database was installed in the standard way, using the installer distributed within the jar file. For both databases, we used a command line based user interface (CLI) (see appendix D for NeXD CLI). As NeXD is not able to run in a daemon mode, accepting and processing requests, we expected a bit worse results then NeXD is able to achieve in reality.

This has proven true during insertion of the documents into a special collection called `perf`. While eXist used about 13 seconds to store 1,000 documents, NeXD took up to 14 minutes. The difference can be explained by comparing operations needed to store the documents:

- eXist is running in the daemon mode. For each XML file in the directory, eXist simply inserts the document to the collection and create basic indices.
- NeXD, on the other hand, does not allow the insertion of multiple files in a batch. This can be easily overcome by a script, but the execution chain then performs following operations:

  1. NeXD is initialized, metadata is loaded;
  2. the document XSD schema is generated and compared to existing schema with the same root element loaded from database (if such schema already exists in database, otherwise schema is stored);
  3. the document is shredded to the database and stored; and
  4. NeXD is terminated.

  If we provide NeXD with a daemon accepting requests, we can omit phases 1 and 4 for each of the documents. This will make the storage a bit faster. However, we still expect a lower performance compared to the eXist one.

5.2.1 Running XPath queries

There are many possibilities how to measure query performance. The possibilities can be combined in a matrix, such as columns are represented by *1*) Measure the time required to deliver the first result; *2*) measure the time to retrieve first n results; and *3*) measure the time to retrieve all results , whereas rows can be divided into *a*) retrieval of DOM instances; and *b*) retrieval of serialized DOM instances .

As it was already said before, NeXD uses a DOM tree to represent the intermediate results. Therefore, we test the performance of a query retrieval in the serialized content form, because the other decision would give an advantage to NeXD. The other decision was to retrieve all results at once. The results themselves were not printed to standard output, but redirected to `/dev/null` device, to minimize the impact of the terminal. We measured the time using a standard Linux utility `time`, using the combination of the time spent in both userspace and kernel. Both NeXD and eXist are using an advantage of the multi-core system, because the user time was ofter higher than the real execution time. For queries retrieving more results, the advantage was diminished because of an overhead caused by result serialization. The results are summarized in table 5.2.

Table 5.2: The time required to retrieve results

Name	NeXD time	eXist time	Total number of results
Q1	5,142 ms	6,123 ms	1,000
Q2	2,713 ms	5,603 ms	1,000
Q3	5,670 ms	7,116 ms	2,000
Q4	5,616 ms	16,062 ms	8,000
Q5	10,166 ms	20,338 ms	11,000
Q6	2,952 ms	11,564 ms	4,000
Q7	9,539 ms	18,625 ms	10,000
Q8	9,250 ms	40,298 ms	40,000

You can see that NeXD performs better in all tested queries. The biggest performance gap we experienced in the queries, which contain a wildcard step at the end ($Q4$, $Q6$ and $Q8$). NeXD must query the database to reconstruct the whole element ($Q3$), or to retrieve all children ($Q4$), which represent the very similar operation over the relational database. Thee DOM operations required to construct an XDM instance actually made query $Q3$ slower than $Q4$, which is the opposite of eXist.

5.3 Summary

NeXD provides quite promising results comparing its performance to eXist. The results can be additionally improved by a cache, a daemon database mode and the other improvements mentioned in the thesis. Still, eXist is a project which provides more than just an XML database, it allows an user to add its own plugins, it has both a web-based and a Java-based GUI and it is capable to store other file types than XML documents as well. NeXD misses these advanced features, but concentrates on a raw performance.

Chapter 6

Conclusion

6.1 Summary

We have shown that data-centric XML documents which conform to XSD schema can be stored in relational database using the modified Hybrid algorithm with very satisfiable result. Because we allowed redundancy, we are able to pull out original document even with processing instructions, comments and other elements which are not stored in relation tables.

This work provides a solid theory to implement a XML-aware data store based on a RDBMS. We have shown how XPath queries can be processed in an efficient way. The query evaluation system is based on a LL(*) parser, which fetch data in incremental steps to avoid exhaustive and thus very expensive join operations. However, the XPath language has shown to be much more complex than expected and because code inherited from former project was not production ready and didn't implement even whole XAPI, we have reverted the functionality nearly to older version in means of supported XPath language.

The main contribution therefore lies in XPath parser, which supports expressions which are not implemented yet in retrieval system, however due to object inheritance can be created easily. Because main issues blocking SQL statement execution were identified and described, we do not expect any problems with further implementation. The parser itself actually contains parts related to XQuery and XQuery extensions, so in the next step NeXD can be shifted to XQuery 1.0 conformant implementation of the XML database. Additionally, we provide a skeleton of XQJ implementation, which we expect to become a new Java XML querying standard, so literally NeXD didn't miss the boat. After all, the current XAPI implementation is a proxy to XQJ implementation, so XQJ is working in standalone mode.

The decision to migrate to different query API, after project was started and its formal specification was created, was very costly. However, it will provide the starting point for all other NeXD developers and it will lead to performance speed up as well. Moreover, it will make NeXD a competitive project.

We can't omit the publication of the project to the open source community, meaning a lot of time spent which was closely related to the master's thesis, but not a direct part of it. The presentation of NeXD to other developers on XML Prague 2010, the possibility to consult strong and weak points and actually making some people convinced we might be the solution they are waiting for were no doubt astonishing results. The idea of publishing the work outside of the academic sphere was present since the very beginning and it was the main reason why I have decided to write the master's thesis in English.

6.2 Promoting NeXD as an open source project

NeXD, from its beginning was considered as an open source project. We liked to share the idea of having a fast native XML database based on proven technologies and moreover to share the implementation itself with the community. NeXD ascends from older master's thesis developed by Radim Hernych [45]. However, his code was bound to the NetBeans IDE [24], making impossible to build NeXD predecessor externally. This might not be seen as a big limitation, however, we wanted to grant the freedom of choice for all incoming developers, as nothing is more frustrating for an enthusiastic volunteer than getting familiar with completely new IDE.

It was clear from the beginning that inherited code will be polished and published in a repository. After discussion with the supervisor, it was decided that we want Git as version control system. Therefore, we have chosen the host *gitorious.org* from the list of public Git repositories available at [4]. Git provided us a local versioned development environment and synchronization with master repository located at *gitorious.org/nexd*.

Despite the fact that the code was published in beginning of March, NeXD is currently developed as a single-man project, due to the limitation posed by master's thesis. Once the thesis is finished, the project will continue its evolution by next natural step, that is including a way how to report bugs, issues and feature requests. Additionally a user forum and mailing list are the common way how to ask questions. The NeXD will inform about passing milestones on *Twitter*, using `#nexddb` nick.

Once the choice of the tool was decided (we sticked with *Maven 2.x*), the project itself was made public. Because wanted to present NeXD to the community, there were definitely better possibilities than just silently creating a project on *SourceForge.net*. Thus, during March, NeXD was presented at *XML Prague 2010* [28], a conference on XML.

I lead a very fruitful discussion with Adam Retter, one of the main eXist developers. As eXist is one of our main competitors (surely we can speak about competition even among open source projects), he was really anxious about our performance and suggested us to run tests against eXist 1.4 branch. Additionally, Adam explained me the why they modified XAPI and where are the pitfalls of the implementation. NeXD was presented there as a poster during the poster session and had a quick presentation over the full audience. Moreover, the description of NeXD is a part of the conference proceedings from Institute for Theoretical Computer Science, Charles University in Prague [51].

Another successful story is the presentation of NeXD at Student EEICT 2010, a conference held by by the Faculty of Electrical Engineering and Communication and the Faculty of Information Technology of the Brno University of Technology. The NeXD poster was presented there, jury was introduced to NeXD, its current status and roadmap & development. The project won the first place in its category (information systems), which was no doubt a big success.

Bibliography

[1] A XQuery benchmark service. http://www.xqbench.org, available in May, 2010.

[2] ANTLR parser generator. http://www.antlr.org/, available in December, 2009.

[3] Apache Hadoop HBase. http://hadoop.apache.org/hbase/, available in December, 2009.

[4] Comparison of free software hosting facilities. http://en.wikipedia.org/wiki/Comparison_of_free_software_hosting_facilities, available in May, 2010.

[5] DocBook.org. http://www.docbook.org, available in May, 2010.

[6] Document Object Model Traversal. http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal, available in December, 2009.

[7] Document Type Declaration. http://www.w3.org/TR/REC-xml/#dt-doctype, available in May, 2010.

[8] Eclipse.org home. http://www.eclipse.org/, available in May, 2010.

[9] EMMA: a free Java code coverage tool. http://emma.sourceforge.net, available in May, 2010.

[10] eXist: Open Source Native XML database. http://www.exist-db.org/, available in December, 2009.

[11] Extensible Markup Language. http://www.w3.org/XML/, available in December, 2009.

[12] Git - Fast version control system. http://git-scm.com/, available in May, 2010.

[13] Java Architecture for XML Binding (JAXB). http://java.sun.com/developer/technicalArticles/WebServices/jaxb/, available in May, 2010.

[14] jStyleParser. http://cssbox.sourceforge.net/jstyleparser, available in May, 2010.

[15] Maven: Welcome to Apache Maven. http://maven.apache.org/, available in May, 2010.

[16] National and Local Weather Forecast, Hurricane, Radar and Report. http://www.weather.com/, available in December, 2009.

[17] PostgreSQL: The world's most advanced open source database. http://www.postgres.org/, available in December, 2009.

[18] RELAX NG. http://www.relaxng.org/, available in December, 2009.

[19] Schematron. http://www.schematron.org, available in May, 2010.

[20] TestNG. http://www.testng.org/, available in May, 2010.

[21] The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 225. http://jcp.org/en/jsr/detail?id=225, available in May, 2010.

[22] The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 5. http://jcp.org/en/jsr/detail?id=5, available in May, 2010.

[23] Trang. http://www.thaiopensource.com/relaxng/trang.html, available in May, 2010.

[24] Welcome to NetBeans. http://www.netbeans.org, available in May, 2010.

[25] XML Database API Draft. http://xmldb-org.sourceforge.net/xapi/xapi-draft.html, available in May, 2010.

[26] XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath/, available in May, 2010.

[27] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, available in December, 2009.

[28] XML Prague 2010. http://www.xmlprague.cz/2010/index.html, available in May, 2010.

[29] XML schema. http://en.wikipedia.org/wiki/XML_schema, available in May, 2010.

[30] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, available in December, 2009.

[31] XQuery 1.0 and XPath 2.0 data model (XDM). http://www.w3.org/TR/xpath-datamodel/, available in December, 2009.

[32] XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/, available in December, 2009.

[33] XRX. http://en.wikibooks.org/wiki/XRX, available in December, 2009.

[34] XML Schema Part 0. http://www.w3.org/TR/xmlschema-0/, available in March, 2008, 2004.

[35] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the XML-to-relational mapping problem. In *In WIDM'04: Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*, pages 31–38, New York, NY, USA, 2004.

[36] Sihem Amer-Yahia Att. Storage techniques and mapping schemas for XML, 2003.

[37] A. Balmin and Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Database. *The VLDB Journal*, 14(1):30–49, 2005.

[38] Phil Bohannon, Juliana Freire, Prasan Roy, , and Jérôme Siméon. From XML schema to relations: A cost-based approach to XML storage. In *In ICDE*, pages 64–75, 2002.

[39] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Prasan Roy, Jérôme Siméon, and Maya Ramanath. Legodb: Customizing relational storage for XML documents. In *In VLDB*, pages 1091–1094, 2002.

[40] Ronald Bourret. XML and Databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm, available in January, 2010, 2005.

[41] Timo Böhme and Erhard Rahm. Supporting efficient streaming and insertion of XML data in RDBMS, 2004.

[42] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *IN PROCEEDINGS OF THE 7TH CONFERENCE ON USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - VOLUME 7*, pages 205–218, 2006.

[43] Petr Chmelař, Radim Hernych, and Daniel Kubíček. Interactive visualization of data-oriented xml documents. In *Advances in Computer and Information Sciences and Engineering*, Computer Science, pages 390–393. Springer Verlag, 2008.

[44] Peter M. Fisher. XQBench - A XQuery Benchmarking Service. In *XML Prague 2010 - Conference Proceedings*, ITI Series, pages 341–355. MATFYZPRESS, 2010.

[45] Radim Hernych. Transformace a persistence XML dat v relační databázi. Master's thesis, FIT BUT in Brno, 2009.

[46] Jaroslav Rychta and Irena Mlýnková. Přednášky k předmětu PGR036, Technologie XML, 2009.

[47] Mary Ann Malloy and Irena Mlýnková. Closing the Gap between XML and Relational Database Technologies: State-of-the-Practice, State-of-the-Art and Future Directions.

[48] Graeme Mathieson. PostgreSQL schema name restriction. http://woss.name/blog/2005/8/1/postgresql-schema-name-restriction.html, available in May, 2010.

[49] Irena Mlýnková and Jaroslav Pokorný. XML in the world of (object-)relational database systems. In *13th International Conference on Information Systems Development Advances in Theory, Practice, and Education*, pages 63–76, 2005.

[50] Karel Piwko. Nativní XML databáze. Bachelor's Thesis. FIT BUT in Brno, 2008.

[51] Karel Piwko, Petr Chmelař, Radim Hernych, and Daniel Kubíček. NAXD. In *XML Prague 2010 - Conference Proceedings*, ITI Series, pages 307–316. MATFYZPRESS, 2010.

[52] Juan M. Pérez, Torben Bach Pedersen, Rafael Berlanga, and María J. Aramburu. IR and OLAP in XML document warehouses. In *Advances in Information Retrieval*, volume 3408, pages 536–539, Springer Berlin / Heidelberg, 2005.

[53] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, and Igor Tatarinov. A general technique for querying XML documents using a relational database system. *ACM SIGMOD Record*, 30(3):20–26, 2001.

[54] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[55] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Scalable transactions for web applications in the cloud. In *Proceedings of the Euro-Par Conference*, Delft, The Netherlands, August 2009.
http://www.globule.org/publi/STWAC_europar2009.html.

List of used abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AJAX	Asynchronous JavaScript and XML
ANTLR	Another Toolkit for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
AT	Advanced Technology
ATA	AT Attachment
BLOB	Binary Large Object
CDATA	Character Data
CLI	Command Line Interface
CLOB	Character Large Object
DAG	Direct Acyclic Graph
DDML	Document Definition Markup Language
DLN	Dynamic Level Numbering
DOM	Document Object Model
DSD	Document Structure Definition
DSDL	Document Schema Definition Languages
DTD	Document Type Definition
EBNF	Extended Backus-Naur Form
EE	Enterprise Edition
EEICT	Electrical Engineering, Information and Communication Technologies
ER	Entity-Relationship
FLWOR	For, Let, Where, Order by, Return
GNU	GNU's Not Unix
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTML5	HTML version 5
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
JDBC	Java Database Connectivity
JSR	Java Specification Request
J2SE	Java 2 Standard Edition
LL(*)	Left to right, Leftmost derivation with an arbitrary lookahead
LUKS	Linux Unified Key Setup
LVM	Logical Volume Manager
ODBC	Open Database Connectivity

PCDATA	Parsed Character Data
PL/pgSQL	Procedural Language/PostgreSQL Structured Query Language
POM	Project Object Model
QName	Qualified Name
RDBMS	Relation Database Management System
RDF	Resource Description Framework
RELAG NG	Regular Language for XML Next Generation
REST	Representational State Transfer
RNC	RELAX NG Compact
RNG	RELAX NG
RPM	Rotations per Minute
SGML	Standard Generalized Markup Language
SOX	Schema for Object-Oriented XML
SQL	Structured Query Language
TREX	Tree Regular Expressions for XML
UCS	Universal Character Set
URI	Uniform Resource Identifier
UTF-8	8-bit UCS/Unicode Transformation Format
W3C	World Wide Web Consortium
XAPI	XML:DB API
XDM	XQuery 1.0 and XPath 2.0 Document Model
XHTML	Extensible HTML
XHTML5	XHTML version 5
XML	Extensible Markup Language
XML:DB	XML Database
XQJ	XQuery for Java
XRX	XForms/REST/XQuery
XSD	XML Schema Document
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations
4NF	Fourth normal form

List of appendices

Following appendices are enclosed as a part of this work:

Appendix A

Configuration used for testing

A.1 Hardware

- Intel(R) Core(TM)2 Duo T9600 @ 2.80GHz
- 4GB of PC2-8500 1066MHz DDR3
- Serial-ATA/150 160GB disk, 7200 RPM

A.2 Software

- eXist revision version 10440
- Java(TM) 1.6.0_17
- PostgreSQL 8.4.4.
- GNU Linux, distribution Fedora 12, kernel 2.6.32.11-99.fc12.x86_64
- ext4 file system, present in LVM and encrypted by LUKS

Appendix B

XML schemas for the Cassini document

NeXD generated schema presented as source B.1 for Cassini document.

Source code B.1: XSD schema for Cassini document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <xs:element name="nasa-data">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="probe"/>
        <xs:element ref="measure"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="probe">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="launch-date"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name" type="xs:NCName"/>
  <xs:element name="launch-date">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="day"/>
        <xs:element ref="month"/>
        <xs:element ref="year"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="day" type="xs:integer"/>
  <xs:element name="month" type="xs:NCName"/>
  <xs:element name="year" type="xs:integer"/>
  <xs:element name="measure">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="distance"/>
        <xs:element ref="destination"/>
        <xs:element ref="data"/>
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:NMTOKEN"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="distance">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="value"/>
        <xs:element ref="unit"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="value" type="xs:integer"/>
  <xs:element name="unit" type="xs:NCName"/>
  <xs:element name="destination" type="xs:NCName"/>
  <xs:element name="data">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="water"/>
        <xs:element ref="albedo"/>
        <xs:element ref="temperature"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="water" type="xs:decimal"/>
  <xs:element name="albedo" type="xs:decimal"/>
  <xs:element name="temperature" type="xs:decimal"/>
</xs:schema>
```

Ignoring the fact that XDM provides datatypes for defining dates in a better mapping, RELAX NG schema for Cassini document is shown as source B.2.

Source code B.2: RNG XML schema for the Cassini document

```
<element name="nasa-data" xmlns="http://relaxng.org/ns/structure/1.0"
    datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
    <element name="probe">
        <element name="name">
            <text/>
        </element>
        <element name="launch-date">
            <element name="day">
                <data type="positiveInteger"/>
            </element>
            <element name="month">
                <data type="string"/>
            </element>
            <element name="year">
                <data type="gYear"/>
            </element>
```

```
        </element>
    </element>
    <oneOrMore>
        <element name="measure">
            <attribute name="id">
                <data type="ID"/>
            </attribute>
            <element name="distance">
                <element name="value">
                    <data type="positiveInteger"/>
                </element>
                <element name="unit">
                    <choice>
                        <value type="string">km</value>
                        <value type="string">AU</value>
                    </choice>
                </element>
            </element>
            <element name="destination">
                <text/>
            </element>
            <element name="data">
                <interleave>
                    <optional>
                        <element name="water">
                            <data type="float"/>
                        </element>
                    </optional>
                    <optional>
                        <element name="albedo">
                            <data type="float"/>
                        </element>
                    </optional>
                    <optional>
                        <element name="temperature">
                            <data type="float"/>
                        </element>
                    </optional>
                </interleave>
            </element>
        </element>
    </oneOrMore>
</element>
```

It is obvious that the XML syntax is extremely verbose. It can be shortened significantly when written in RNC (RELAX NG Compact syntax), as shown in source B.3.

Source code B.3: RNC schema for the Cassini document

```
element nasa-data {
    element probe {
        element name { text },
        element launch-date {
            element day { xsd:positiveInteger },
            element month { xsd:string },
```

```
            element year { xsd:gYear },
        },
        element measure {
            attribute id { xsd:ID },
            element distance {
                element value { xsd:positiveInteger },
                element unit { string "km" | string "AU" }
            },
            element destination { text },
            element data {
                element water { xsd:float }?
                & element albedo { xsd:float }?
                & element temperature { xsd:float}?
            }
        }+
}
```

Appendix C

XPath 2.0 and XQuery 1.0 grammar snippets

This appendix presents parts of XPath and XQuery grammars important for our implementation.

C.1 XPath 2.0 grammar snippets

The path expression in XPath 2.0 can be represented by EBNF as source C.1, with further details omitted.

Source code C.1: EBNF for path expression

```
PathExpr            ::=     ("/" RelativePathExpr?)
                          | ("//" RelativePathExpr)
                          | RelativePathExpr

RelativePathExpr    ::=     StepExpr (("/" | "//") StepExpr)*

StepExpr            ::=     FilterExpr | AxisStep

AxisStep            ::=    (ReverseStep | ForwardStep) PredicateList

ForwardStep         ::=    (ForwardAxis NodeTest) | AbbrevForwardStep

ReverseStep         ::=    (ReverseAxis NodeTest) | AbbrevReverseStep

PredicateList       ::=     Predicate*
```

C.2 XQuery 1.0 grammar snippets

The FLOWR expression can be represented by EBNF as source C.2, with further details omitted.

Source code C.2: EBNF for FLOWR expression

```
FLWORExpr   ::=   (ForClause | LetClause)+
                        WhereClause? OrderByClause? "return" ExprSingle
```

```
ForClause   ::=   "for" "$" VarName TypeDeclaration? PositionalVar? "in"
                    ExprSingle
                ("," "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle)*

LetClause   ::=   "let" "$" VarName TypeDeclaration? ":="
                    ExprSingle
                  ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*

TypeDeclaration ::=     "as" SequenceType

PositionalVar ::=     "at" "$" VarName

WhereClause     ::=     "where" ExprSingle

OrderByClause ::=     (("order" "by") | ("stable" "order" "by"))
                        OrderSpecList

OrderSpecList ::=     OrderSpec ("," OrderSpec)*

OrderSpec     ::=     ExprSingle OrderModifier

OrderModifier ::=     ("ascending" | "descending")?
                        ("empty" ("greatest" | "least"))?
                        ("collation" URILiteral)?
```

Appendix D

NeXD command line interface

NeXD provides a binary represented by a single jar file created during Maven `package` phase. To generate the binary, without executing tests, simply run Maven with arguments `package -DskipTests`. The binary is generated in the `target` directory. The usage is present in source D.1.

Source code D.1: NeXD CLI

```
usage: java -jar <jar-with-dist> uri [-D <property=value>] -d
       <database-name> [-h] [-i <insert-document> | -n <new-collection> |
       -q <query> | -r <remove-collection>]  [-p <password>]   -U
       <username>
 -D <property=value>                          use value for given property
 -d,--database-name <database-name>           Database name to connect to.
 -h,--help                                    Show usage of application
 -i,--insert-document <insert-document>       Inserts a file into the
                                              database
 -n,--new-collection <new-collection>         Creates a new collection in
                                              the database
 -p,--password <password>                     Password used to connect to
                                              the database
 -q,--query <query>                           XPath query to be performed
                                              on database
 -r,--remove-collection <remove-collection>   Deletes a collection from
                                              the database
 -U,--username <username>                     User name used to connect to
                                              the database.
```

Appendix E

Relational database schema of NeXD

NeXD requires that the database contains the schema defined as E.1. Please note that the collection `root` is required, since all other collections are its ancestors. This way we allow to select whole database content.

Source code E.1: NeXD database schema

```
-- PostgreSQL
--
-- Owner property will be replaced during Maven resources:resources phase

--DROP SCHEMA public CASCADE;
--CREATE SCHEMA public;

DROP TABLE xcollection CASCADE;
DROP TABLE xschema CASCADE;
DROP TABLE xtable CASCADE;
DROP TABLE xdocument CASCADE;
DROP TABLE xtext CASCADE;


CREATE TABLE xcollection (
   id SERIAL,
   name VARCHAR(30),
   parent_collection INTEGER REFERENCES xcollection
                     ON UPDATE CASCADE ON DELETE CASCADE,
   PRIMARY KEY(id),
   UNIQUE(name)
);

CREATE TABLE xschema (
   id INT NOT NULL,
   name VARCHAR(100),
   collection INTEGER REFERENCES xcollection
              ON UPDATE CASCADE ON DELETE CASCADE,
   uri TEXT[],
   xsd TEXT,
   PRIMARY KEY(id),
```

```
   UNIQUE (collection, name)
);

CREATE SEQUENCE xschema_id_seq OWNED BY xschema.id;

CREATE TABLE xtable (
   id SERIAL,
   collection INTEGER NOT NULL REFERENCES xcollection
              ON UPDATE CASCADE ON DELETE CASCADE,
   schema INTEGER REFERENCES xschema
          ON UPDATE CASCADE ON DELETE CASCADE,
   parent_table INTEGER REFERENCES xtable
                ON UPDATE CASCADE ON DELETE CASCADE,
   inlined BOOLEAN,
   element_name VARCHAR(50),
   table_name VARCHAR(50),
   elements TEXT[],
   attributes TEXT[],
   PRIMARY KEY(id),
   UNIQUE (schema, table_name)
);

CREATE TABLE xdocument (
   id INTEGER NOT NULL,
   collection INTEGER NOT NULL REFERENCES xcollection
              ON UPDATE CASCADE ON DELETE CASCADE,
   schema INTEGER REFERENCES xschema
          ON UPDATE CASCADE ON DELETE CASCADE,
   name VARCHAR(30),
   content TEXT,
   PRIMARY KEY (id),
   UNIQUE (name, collection)
);

CREATE TABLE xtext (
   id SERIAL,
   xtable INTEGER NOT NULL REFERENCES xtable
          ON UPDATE CASCADE ON DELETE CASCADE,
   position INTEGER NOT NULL,
   context TEXT,
   PRIMARY KEY (id),
   UNIQUE (position, xtable)
);

CREATE SEQUENCE xdocument_id_seq OWNED BY xdocument.id;

ALTER TABLE xdocument OWNER TO ${nexd.userName};
ALTER TABLE xtable OWNER TO ${nexd.userName};
ALTER TABLE xcollection OWNER TO ${nexd.userName};
ALTER TABLE xschema OWNER TO ${nexd.userName};
ALTER TABLE xtext OWNER TO ${nexd.userName};


-- insert default collection
```

```
INSERT INTO xcollection VALUES(DEFAULT, 'root', NULL);
```

Additionally, NeXD uses trigger E.2 to ensure the schema generated by the Hybrid method is pruned once the `XSchema` table row is deleted.

Source code E.2: Deletion trigger in NeXD

```
-- PostgreSQL
--
-- drops complete schema when a record is deleted from xschema table
CREATE OR REPLACE FUNCTION drop_schema () RETURNS trigger AS $$
  BEGIN
    EXECUTE 'DROP SCHEMA "' || OLD.name || '_' || OLD.id || '" CASCADE';
    RETURN OLD;
  END;
$$ LANGUAGE plpgsql /

CREATE TRIGGER drop_schema_trigger BEFORE DELETE ON xschema
  FOR EACH ROW EXECUTE PROCEDURE drop_schema() /
```