in file `XParsingContext.java`. It allows program to use arbitrary node of XPath grammar as entry point (starting symbol) and provides automatic connection of low-level parser with high-level one. The choice of entry point is extremely useful. Apart from testing, it allows to switch between XPath path expression support to full XPath 2.0 support or even XQuery 1.0 (if implemented) by simple code modification, making parser very universal tool.

4.3.3 Binding contexts

When we create an XPath expression representation in our database (see remarks in section 4.5), we have to access database metadata created during application of 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 dynamic context, since it differs for each evaluation.

NeXD in actual implementation doesn't support binding external variables for they are not supported by implemented API. So, static binding consists of selecting the collection according to the URI used to connect to 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 XPath 2.0 subset without having to implement whole specification.

4.3.4 Retrieving XDM instances from database

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

Actual XPath step must be mapped to metadata, precisely to `XTables` available in system. Since tables are organized in hierarchical structure in the database, we can narrow the selection of appropriate document fragments even before touching 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 next step.

The biggest problem lies in axes, which are transitive closures or require a document to traverse 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 finding that a relation exists between two relational table does not guarantee the same relation exists for current step and evaluation context.

The evaluation of steps in sequence provides lower number of SQL joins, but higher number of SQL commands required to execute the query. This is expected behaviour, such as it was identified while performance of Hybrid method was compared to Shared and Basic methods in [50].

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

- For inlined tables, the query must be executed immediately, narrowing the possible elements; however
- 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 root of execution for

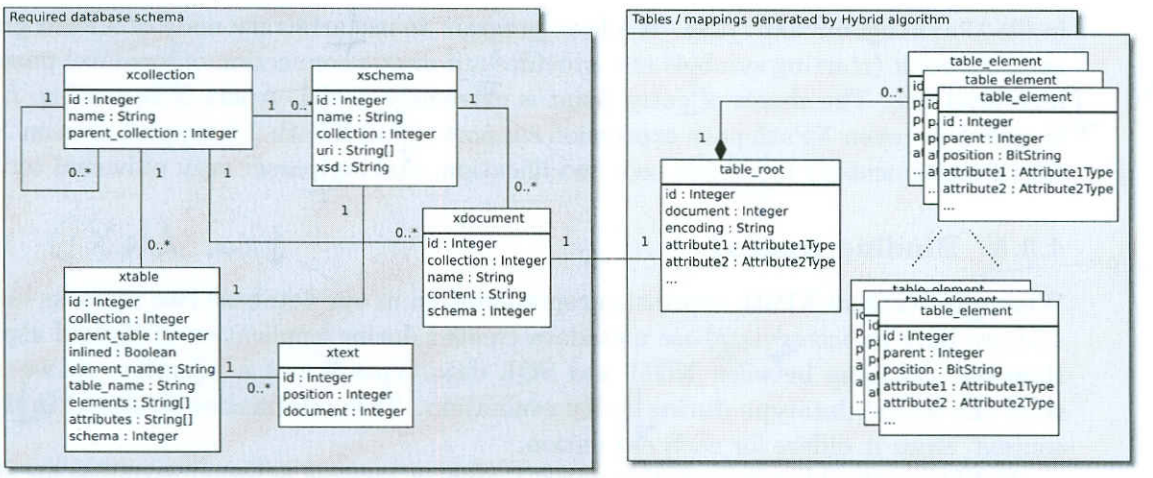

Figure 4.1: Metadata tables used in system

The tables used for metadata, not presented before, are:

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

XCOLLECTION XCollection encapsulates documents in sets, modelling hierarchy; and

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.

4.3.2 Parsing XPath language expression

The core for language parsing 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 **L**anguage **R**ecognition **T**ool [2]) parser. This tool was chosen because it allows easy binding between Java and grammar 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 arbitrary textual input or, using an high level abstraction, an abstract syntax tree (AST) representation of the input. NeXD uses the ANTLR most 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 query must be actually parsed twice, it is more time consuming. Fortunately, the speed is compensated by much easier parser modification, ability to recover for errors, thus providing better 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 FLOWR 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. Our parser is represented

The main difference in processing of simple and complex elements in algorithm 4.1 lies in 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 creation of the standalone table, as allowed by Hybrid method presented in section 3.3.1.

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

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

4.3 XPath processing

As already stated before, NeXD implements a XPath 2.0 language parser for querying data. In this section we focus on implementation details of 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 mentioned in previous section, Hybrid method uses `XSCHEMA` and `XTABLE` metadata tables. However, these are not only metadata tables required. NeXD must store information about documents and collection. In this concept collection represents a set of documents. However, collection can contains other collections as well, so in database collection structure is modeled as a tree. Each XSD document is bound to the XSD schema. Relations are illustrated on figure 4.1. NeXD caches the metadata in memory, making subsequent queries faster by skipping metadata loading phase.

4.2.1 Algorithm life-cycle

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

NeXD implements DOM traversal using standard `TreeWalker` interface, which is a part of W3C DOM 2 specification [6]. However, JAXP does not ensure that 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. Algorithm perform recursive descent, using name or complex type attributes as references. The first step creates empty set of traversed elements and empty map of created tables. NeXD identifies the root element from XSD. The root is used as *current* element. The recursive traversal can be simplified to:

Algorithm 4.1 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 XML element or attribute, which contains only one child, which is of `xs:complexType` type; or *ii*) XSD element represents typed element, which is not generic and that is complex; or *iii*) XSD element references complex type element.

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

schema is present as source D.1):

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

XAPI enforces possibility of multiple schemas for documents of in one collection (to be precise, XAPI does not define type of documents in collection at all). However, NeXD limits this to arbitrary number of schemas with 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 are always rejected; and

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 are used for quick querying of the content. XTable contains a pointer to the table generated by Hybrid algorithm. The way how data are queried is further described in section 4.3.

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

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`

Table 4.2: Mapping between XDM and SQL data types

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

The last problem to be solved for 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.

Property name	Default value	Explanation
`nexd.dbName`	`nexd`	The name of database on the machine used as 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 hostname or 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 is SSL should be user for connecting to database. This depends purely on setting of the underlying database.
`nexd.sslFactory`		The name of factory, which should be 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 CA registered within your Java environment.

Table 4.1: Java system properties accepted by NeXD

4.2 Implementing the Hybrid method

Hybrid method in NeXD creates relational schema from XSD document. The XSD document is either delivered together with input document in a place JAXP parser can found it; or, which is more usually the case, generated on-the-fly by *Trang*[21]. Trang is an open source tool, which is able to convert different schema types and generate a schema from XML document instance. It tries to create human readable schemas, which have led to minor advantages during 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 usage of XSD schema stored in our database tables, which was a major problem.

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

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

Source code 4.1: Running the *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 testsuite expects an *PostgreSQL 8.3+* instance running at `localhost:5432`, with user `nexd` authenticated by password `test` as an owner of database `nexd` including full rights. However, default properties defined in Maven's `pom.xml` file can be easily overridden by providing `-Dproperty.name=property.value` options during command line execution. The list of supported values, which can be used both launch NeXD from command line using *Maven* or testsuite is summarized in table 4.1.

NeXD contains approx. 80 testcases, going through the parser and both implemented APIs (see section 4.5). However, it is clear that it is not fully covered, so it would be nice to append the *EMMA*[9], a free Java code coverage tool, preferably as Maven plugin configuration to measure the coverage and identify weak points of the implementation. This is one of the improvement which will be surely added during 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 file present in *src/test/testng* directory.

Chapter 4

Implementing NeXD

This chapter is the core of master's thesis. It describes issues and pitfalls of current implementation, as well as its advantages. However, it is not a listing of source code snippets with comments, for reader interested in this kind of information should 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 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 Hybrid algorithm is implemented.

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

4.1 Selecting the build tool

NeXD has become a project which will be used by multiple people, in various environments, therefore have chosen *Maven*[15] as the build management system for our project. 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 many more others.

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

The concept of plugins allows appending 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 an 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.

additionally store each XML document in RDBMS as a BLOB object. This way, we are able to preserve additional data such as comments and processing instruction 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.

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 faster selection. In this version, we are even not supporting XQuery Update Facility 1.0, XUpdate or any other mean of updating data except direct change by an SQL command. Because we allowed the redundancy, complexity of an update operation will raise with respect to size of the document and its modification. However, both APIs we are focusing (see section 4.5) on support modification of persistent data, so this functionality may be added in future versions.

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

Next, if we allow modification of the schema by 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 sophisticated shredding and hurts performance, so we have decided to limit 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 seems 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 introduction to document retrieval. The next chapter show how these methods are implemented.

3.4 Shredding XML document into relations

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

1. Allowing storage of these documents by dynamically creating database relations based on their schema;
2. Store parts of documents not conforming to 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 them can allow us being more general but on the other hand it will greatly augment complexity of queries because unions of results gained from different schema mapping are required. Further, it is difficult for user to query schema non-conforming data, because he is simply not aware of the additional XML elements existence.

When queried data are an input for another processing tools, elements not defined in XML schema unnecessarily leverage complexity of these tools. As the result, we prefer rejecting not-conforming XML documents with 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 user define if the not conforming part of document must either be transformed or cut off and the rest of document stored in database. Since insertion of data into DBMS 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 [40]. Once document are transferred 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 with 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 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 performance.

The two fundamental operations, extraction and querying (described in section 2.3) have different time and space complexity characteristics when Hybrid method is used. When XML document is shredded into relations, extraction is extremely difficult and time consuming operation. To overcome the problem, we allowed data redundancy. Thus, we

the next steps. Case *1*), 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 database as well. The DOM tree, which represents 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, *2*), represents the core of 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 at figure 4.2

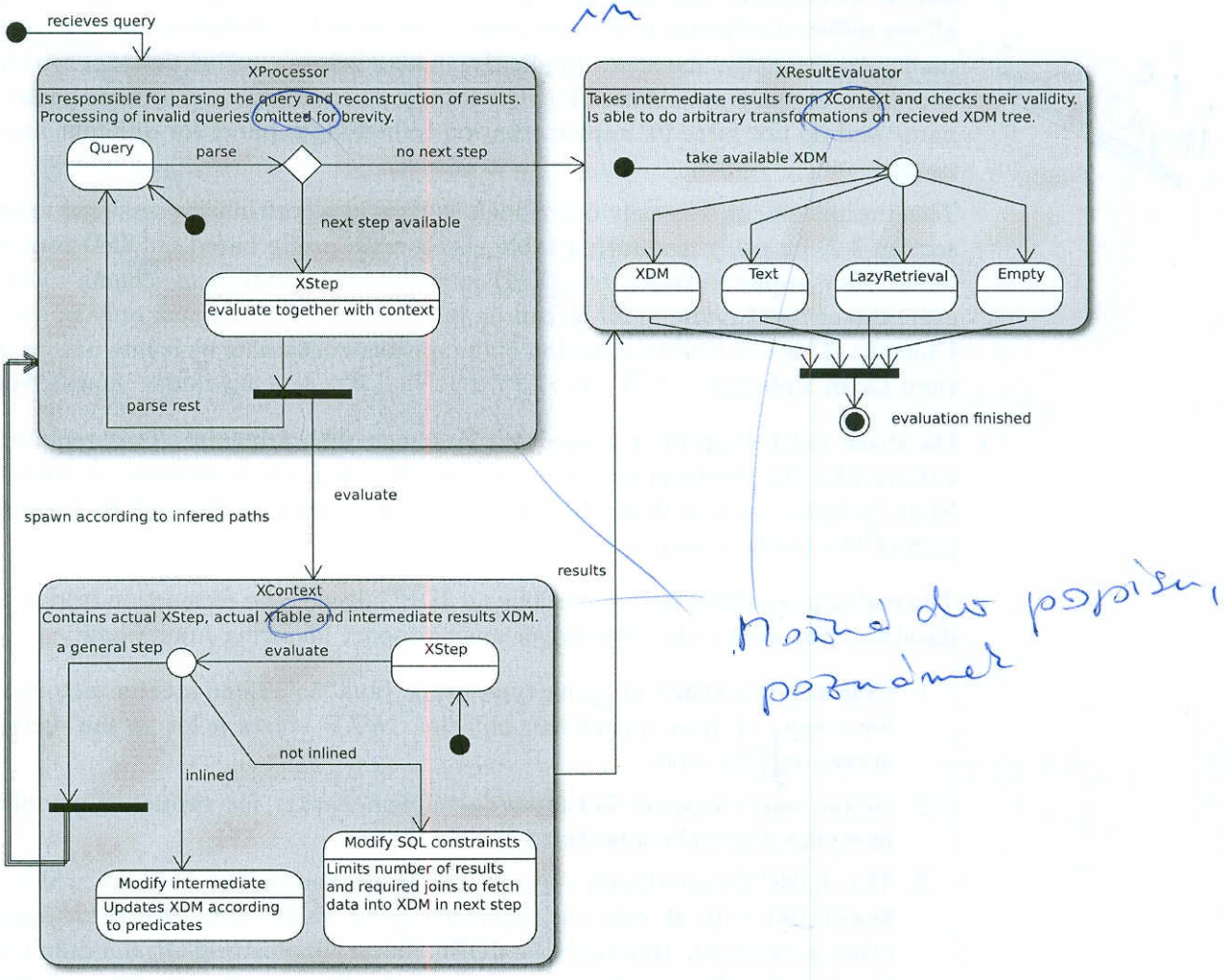

Figure 4.2: Process of XDM instance retrieval

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. A part of the section explains binding to the database.

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

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

- 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 user, but this is not used. Additionally, database implicitly creates schema called `public`, which is not defined in the SQL standard. 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 Hybrid algorithm processing, explained in section 4.2, by using generating table names with prefix based on XSD root element name. Our implementation uses 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 D.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 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.

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

[3]Surprisingly, the code of the relational database has proven useful while searching for limitations of schema naming, as explained in [45].

language extensions and authentication based on user name and password, although *indent* method should work as well.

NeXD can be ported to other database which holds the constraints. However, it does not support any database abstraction layer, and all SQL statements are written in native language. Thus the migration represents revisiting the SQL statements (not necessarily all of them) and binding the JDBC data source properties to 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 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[20]) focuses rather on XML document parsing, XSLT transformations and DOM model then on query functionality. JAXP is in fact used to retrieve platform available XML parser for processing XML documents and XML schema representation in NeXD.

4.5.1 XML:DB API

XML:DB API (XAPI [23]) 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 minumum 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. 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.

However, implementing only 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 have led to situation, where general interoperability is hard to achieve. Therefore, NeXD used rather modification of API 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 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[19]) is a generic data access framework which provides a uniform interface for XQuery implementations in 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; and
- Create an 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 which may result in significantly faster execution for repeated queries.

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

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 [42] was massively rewritten. Moreover API comments were translated from Czech to English, to make project useful for 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 worthing it.

Chapter 5

Evaluation of performance

As was stated in the chapter 1, NeXD aims to be a fast retrieval engine system with support of querying by XML query language, that is act as an native XML database. During implementation we described strong and weak points of the implementation, so we have to verify the theoretical output with 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 complex schema generation, exercising Trang and Hybrid method components;
- Ratio between real size of document and space used in NeXD;
- Memory requirements for processing documents;
- Query time, including different means of serialization; and
- Clustering performance

NeXD contains TestNG unit tests, which act both as smoke and functional tests. NeXD contains an interface, which allows easy testing of 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 [41]. Even if 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 dynamic context is not portable and thus this frameworks could not be used.

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 testing system described in appendix A. Queries we tested are summarized in table 5.1, documents from [16] were used as test input.

Name	XPath query
1	`/weather/head/locale`
1	`/weather/dayf/day[1]/part/wind`
1	`/weather/dayf/day[1]/part/wind/*`
1	`//wind`
1	`//cc/wind/*`
1	`//day[2][hi > 75]/part/wind/*`
1	`//cc[obst='Brno, CZECH REPUBLIC']/wind`
1	`//day[@t='Saturday']/part[1]`
1	`//day[@t='Saturday']/part[@p='n']`
1	`//part/wind`
1	`//part/wind/*`

Table 5.1: Test queries used to evaluate NeXD performance

5.2 Comparing performance of NeXD with other databases

Chapter 6

Conclusion

6.1 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. However, his code was not in production quality, was bound to the NetBeans IDE[22] is a such way that is was impossible to build it externally. This might not be seen as a big limitation, however, we wanted to grant 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 local development and synchronization with master repository located at *gitorious.org/nexd*.

Despite the fact that 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 try to inform about passing milestones on *Twitter*. The project will be hosted on *sourceforge.net*, with code present both there and at *gitorious.org* and with artifacts available on *sonatype.org*.

Once the choice of tool was decided (we sticked with *Maven 2.x*), the project itself was made public. Because wanted to present NeXD to 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*[25], 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 poster session and had a quick presentation over the full audience. Moreover, our presentation is a part of the conference proceedings from Institute for Theoretical Computer Science[48].

Another successful story is the presentation of NeXD at Student EEICT 2010, a confer-

Appendix B

XML schemas for 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 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>
```

Appendix D

Relational database schema of NeXD

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

Source code D.1: NeXD database schema

```
-- Postgresql
--
-- Owner property will be replaced during Maven resource:resource 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 (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 D.2 to ensure database schema generated by Hybrid method is pruned as well once `XSchema` table row is deleted.

Source code D.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 $$
  BEGIN
          EXECUTE 'DROP SCHEMA ' || OLD.name || '_' || OLD.id || ' CASCADE';
          RETURN NULL;
  END;
$$ LANGUAGE plpgsql /

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