Univerzita Karlova v Praze
Matematicko-fyzikální fakulta
DIPLOMOVÁ PRÁCE

Ondřej Kmoch

Implementace XSLT v prostředí relační databáze

XSLT Implementation in Relational Database Environment

Katedra softwarového inženýrství

Vedoucí diplomové práce: Prof. RNDr. Jaroslav Pokorný, CSc.,
Studijní program: Informatika

2007

Na tomto místě bych chtěl poděkovat svému vedoucímu Prof. RNDr. Jaroslavu Pokornému, CSc. za odborné a přínosné rady, které přispěly ke zdárnému dokončení celé práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 15. srpna 2007

Ondřej Kmoch

Contents

List of Figures

List of Tables

Název práce: *Implementace XSLT v prostředí relační databáze*
Autor: *Ondřej Kmoch*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Prof. RNDr. Jaroslav Pokorný, CSc.*
e-mail vedoucího: *pokorny@ksi.mff.cuni.cz*
Abstrakt:

XML je rozšířený formát, který je využíván k ukládání všech druhů dat, a XSLT představuje standardizovaný jazyk pro transformaci XML dat a jejich struktury. Dnes existuje mnoho implementací XSLT, ale většina z nich udržuje zdrojové XML dokumenty ve strukturách přímo v paměti.

Implementace představená v této práci používá pro udržování těchto dokumentů relační databázi a využívá jazyk SQL k vyhodnocování všech XPath dotazů, které se používají v XSLT.

Nejdříve je představeno nahrávání zdrojových XML dokumentů do generického relačního mapování. Dále je představena transformace XPath dotazů přímo na SQL dotazy a nakonec je popsáno vyhodnocování XSLT transformací pomocí prostředků relační databáze.
Klíčová slova: *XSLT, relační databáze, SQL, XPath*

Title: *XSLT Implementation in Relational Database Environment*
Author: *Ondřej Kmoch*
Department: *Department of Software Engineering*
Supervisor: *Prof. RNDr. Jaroslav Pokorný, CSc.*
Supervisor's e-mail address: *pokorny@ksi.mff.cuni.cz*
Abstract:

XML is widely used format for storing all kinds of data and XSLT standard represents a standardized way, how to transform a XML document to a different structure. Many XSLT implementation has been introduced, but the most of them uses an in-memory representation of the transformed XML document.

The implementation done in this thesis uses the relational database engine to store processed document and takes advantages of SQL to evaluate XPath expressions used by XSLT.

First, importing source XML document into the generic relational mapping is described. For processing XPath expressions, the XPath to SQL convertor is introduced. Lastly, the processing of XSLT stylesheets by relational database engine is shown.
Keywords: *XSLT, relational database, SQL, XPath*

Chapter 1

Introduction

Information system integration and data exchange between systems are very important topics. There exists many different formats of data stored in different systems and an important problem is how to integrates this data together.

World Wide Web Consortium[1] created XML working group. Its target is to publish recommendation about Extensible Markup Language (XML), which is a simple text format primarily intended to be used for electronic publishing. Howeve, XML data format can be also used for interchange data between different systems and this role of the XML data format has become very important. Contemporary systems usually have a XML support and are able to export and import data in this format and these features are usually very uncomplicated to be used. But XML standard is flexible text format, which does not force users to use any specific structure of data. A data format can be written in many different XML forms. For that reason a part of the XML initiative is XSL working group, which created recommendations especially about how to describe transformations between XML formats. This recommendation is very important for interoperability of XML formats, because it is a standardized way, how to describe transformations of XML structure. It defines syntax, which is based on XML, and also defines semantics, how these transformations has to be interpreted.

However, XSL Transformations recommendation does not define, how these XSLT processors has to be implemented. This brings many different opinions, how a XSLT processor should work. This can be very important, because XML and XSLT are used for very different data and for very different

[1]W3C, `http://www.w3.org/`

amount of data. An implementation has to be quite mistakeproof and robust, but on the other side it has to be fast. These two objectives are against each other and today implementations are quite different in fulfilling of these targets.

Very important part of XSLT recommendation is the XPath query language heavily used by XSLT for addressing parts of XML documents and is also used by other applications and languages for orientation in XML documents and for selecting parts of documents.

This is contemporary situation of ideas about XML data and their transformations. However, there is another technology, Relational Database Management Systems (RDBMS). They are usually basic technology to store any data in systems. This technology is quite old, well examined, well documented and is used in most cases, if there is necessary to store any data. World of RDBMS and world of XML are quite different and were quite separated. But it has changed. Usual feature of wide spread RDBMSs is to export and import data in XML. It brings new target for XSLT implementations. Exported data from databases has to be processed to other formats and this can be done by XSLT.

1.1 Thesis target

The main target of this thesis is to demonstrate that a RDBMS[2] is able to work as a XSLT processor as well. RDBMSs are widely used and it should be very effective to use it for processing XML data. Today RDBMSs can usually store XML data in text formats[3], some implementations are able to import and export XML data from and to native tables and decompose XML into fields of tables. Usually XSLT stylesheets are already written for processing this XML data. It should be profitable to use these XSLT stylesheets for processing XML data stored in RDBMS. But today implementation of XSLT (e.g. Saxon [17], Microsoft XSLT[4], Xerces [22], libxslt[5]) are able to work on XML files, not XML stored in a database. Then it is necessary to export XML data from database, process it by a third party XSLT processor and then again import back to the database. It is not quite effective, because export into XML text format from database representation is not

[2]Relational Database Management System

[3]Uses Binary Large Objects (BLOB) to store large XML data in a table column

[4]http://msdn2.microsoft.com/

[5]http://xmlsoft.org/XSLT/

necessary. This thesis demonstrates, how to implement processing of XSLT stylesheets directly in a database. It tries to employ all database features like internal procedure languages and especially query language used in relational databases called SQL. Database implementations are very effective in evaluating SQL queries and this effectiveness is an advantage for processing XSLT stylesheets.

Today XSLT implementations usually follows recommendations done by XSL working group to use in-memory representation of processed XML data. This is very important for effectiveness of processing XSLT programs. But it is not unusual, that, e.g. extracts from RDBMS databases, are in XML formats and these extracts are very huge. Then it is not possible to load entire XML source file into memory. This thesis uses relational database itself as a temporary XML storage and XSLT program is executed directly on relational representation. This implementation tries to benefit from the relational database usage as much as possible and aims not to be dependent on the size of possible memory usage to be able to process large XML data.

For implementation the open source relational database is used. The chosen implementation is described further. This database (like many other relational databases) does not have tools, how to load external data from files into tables, so Java environment has been chosen to import data and export data to and from database. For parsing XML documents the standard SAX ([16]) interface for Java has been used and open source parser Xerces ([22]) is used for parsing.

Structure of tables, which are used for storing XML data can be very different, but analysis of possible choices, how to store XML data into database is beyond the scope of this thesis. Structure of tables and ideas about indexing of XML data has been taken from [23].

The target of this thesis is to implement XSLT interpreter by environment of relational database and very important task is to implement XPath query processing on the relational representation of XML. The XSLT implementation will use stored source XML files and stored XSLT programs. Output XML files will be primarily written into the relational database in the same way the source files are stored.

1.2 Thesis structure

Thesis begins with introduction of all used technologies. This is covered in Chapter 2. It describes XML standard in Section 2.1, XML namespaces in

Section 2.2, XPath language in Section 2.3 and a primary standard, XSLT, in Section 2.4. Than the actual implementation details follows. It begins with description of all used tools and third party products described in Chapter 3. It consists mainly of the Firebird engine in Section 3.1 and the Java environment described in Section 3.3. The following Chapter 4 describes the main parts and the main ideas of the XSLT implementation and is divided into Section 4.1 and Section 4.2. The former one is focused on the main parts of the XSLT implementation and the further one is focused on ideas and conception considered in the implementation. The next two chapters, Chapter 5 and Chapter 6 are core of this thesis. The Chapter 5 describes implementation of XML storage in relational database and Chapter 6 continues with description of the XSLT algorithm implementation. Found limitations of this implementation are summarized in Chapter 7. Chapter 8 contains practical instructions, how the implementation can be installed and used by users. There are tests and performance optimizations summarized in Chapter 9. The closing part is covered by Chapter 10. The thesis ends with appendices, which describe used relational structure in Appendix B, structures of demonstrating examples in Appendix C and Appendix D. A content of the enclosed CD is described in Appendix E.

Chapter 2

Technologies and standards

It is necessary to describe precisely all used technologies and standards. It means especially XSLT 1.0 recommendation, XPath 1.0 recommendation and partly XML recommendation.

2.1 XML

Extensible Markup Language[1] has to be mentioned first, because this W3C recommendation is a foundation for all the following standards. XML is a text based format for storing data and to store data structure within this format. It has been created as a subset of older SGML[2] and it was created to be a format for publishing data over the internet, to be easy to use. Today XML is used in many different cases and it is usual that data from any application or system can be exported to XML and imported back. This expansion of XML usage brought new opportunity to easily exchange data between applications and systems.

2.1.1 Logical structure

The structure of XML documents follows. It will be discussed in a logical manner. Syntax structure is a little mentioned too for the purposes of this thesis. However, detailed description of XML syntax is not part of this work and can be found in the original XML recommendation ([6]).

[1]XML

[2]Standard Generalized Markup Language, see `http://www.w3.org/MarkUp/SGML/`

The logical structure of XML document is divided especially into two parts — elements and attributes.

Elements

XML is a markup language. It means that all data in XML file are marked and this mark is called *tag*. Tags are of two kinds: *start tags* and *end tags*. One start tag and relevant end tag forms an *element*, which is the base structure of XML data. Each element can contain other elements or text. *Start tag* is made of char '`<`', element name and char '`>`'. *End tags* are similar to start tags. They have the same name, but they start with '`</`'. There is one exception, that there can be empty elements, which do not have any subelement or text inside. They can be made of start tag and immediately following end tag or they can be also made of char '`<`', name of element and of '`/>`'. It is demonstrated in Figure 2.1. There is one root element in the example called `book` — there is the start tag '`<book>`' and the end tag '`</book>`' of this element. And there is also an empty element called `empty` — it is defined by the tag '`<empty/>`'.

```
<book>
  <title>Designing XML applications</title>
  <author>
    <family>Nick</family><given>Marcus</given>
    <family>Bob</family><given>Pant</given>
  </author>
  <text>Example text</text>
  <empty/>
</book>
```

Figure 2.1: XML document example

Attributes

Data in XML format can be also represented by attributes. An *attribute* belongs always to an element — the *parent element*. An *attribute* is written in the start tag of the parent element. Syntax is shown in Figure 2.2.

There is one XML attribute in the example called `name`. This attribute has value '`XML Applications`' and its parent element is `book` element.

```
<book name="XML Applications">
  <title>Designing XML applications</title>
</book>
```

Figure 2.2: XML attribute example

A piece of information can be represented in XML by elements, by attributes, by values of attributes or by text written as a content of elements. It cannot be strictly decided, when to use each style of data storage kind in XML. The example in Figure 2.3 shows, how to store the same information as in Figure 2.2 by an attribute.

```
<book>
  <name>XML Applications</name>
  <title>Designing XML applications</title>
</book>
```

Figure 2.3: XML rewrite example

2.1.2 Kinds of XML documents

Elements, attributes, attribute values and text in a XML document hold information. Usage of this components of XML divides documents into the two kinds: *data driven documents* and *text oriented documents*.

Data driven documents

This kind is quite similar to the relational databases. XML documents hold information in attributes or as a content of elements. However, elements contain only text data and not mixed content. When element contains *mixed content*, it means, that it contains text data mixed together with subelements. Elements in data driven documents hold only text, only subelements or they are empty.

Data driven documents are very often generated by programs to export data, e.g. from a relational database. They can be very large and are not intended to be used or read by humans.

Text oriented documents

Text oriented documents are in previously mentioned points antipole of data driven documents. They heavily use *mixed content* (mentioned in previous subsection) for storing data. They are very often written and read by humans. They cannot be usually straightforwardly mapped into tables of a relational database. They are especially used to store documents, e.g. XHTML documents[3], Docbook documents[4], etc.

2.2 Namespaces in XML

Expansion of the XML language all over the world brought some problems. One of them was that names of elements and attributes defined by users become duplicated and element defined by one user could not be distinguished from elements with the same name of another user. For this purpose namespaces wee published in a W3C recommendation ([14]). This is a simple method, how to fully qualify element names and attribute names. Because of namespaces XML syntax has slightly changed. Each name, names of attributes or elements, is called *qualified name* and can have a prefix separated by a colon from a local name. However, the qualified name has always specified a namespace and a local name, despite it does not have specified prefix. Namespaces are shown in Figure 2.4.

```
<b:book b:name="XML Applications"
  xmlns:b="http://www.book.cz/book/2007">
  <b:title>Designing XML applications</b:title>
</b:book>
```

Figure 2.4: XML Namespaces example

There is one *namespace prefix definition* in the example. Definitions of namespace prefixes are done by XML attributes with *special namespace prefix* `xmlns`, which is reserved and cannot be used by users for other purposes. Local name of this attribute specifies prefix for defined namespace. Standard recommends for namespaces to use URIs for worldwide uniqueness, but it is not necessary. So there is namespace `http://www.book.cz/book/2007`,

[3]The Extensible HyperText Markup Language, see http://www.w3.org/TR/xhtml1/
[4]XML based format for documentation, see http://www.docbook.org/

which is represented by the prefix `b` and this prefix is used by all attributes of the XML document in Figure 2.4. For a namespace there can be unlimited number of defined prefixes. The same prefix redefines namespace prefix for the content of its parent. It is shown in Figure 2.5.

```
<b:book b:name="XML Applications"
  xmlns:b="http://www.book.cz/book/2007">
  <b:title xmlns:b="http://www.document.cz/document">
    Designing XML applications
  </b:title>
</b:book>
```

Figure 2.5: XML namespace redefinition

Element ‘`book`’ and attribute ‘`name`’ are from namespace `http://www.book.cz/book/2007`, but element title is from namespace `http://www.document.cz/document`.

It has been mentioned, that all qualified names has always specified namespace. For this purpose there is *default namespace* with no prefix. If it is not defined explicitly, value of default namespace is empty string. Definition and usage of default namespace is shown in Figure 2.6.

```
<book name="XML Applications"
  xmlns="http://www.book.cz/book/2007">
  <title>Designing XML applications</title>
</book>
```

Figure 2.6: Default namespace definition

Attribute `xmlns` without specified prefix defines default namespace and then all attribute and element names without specified prefix are considered to be within this namespace. This means, that in the preceding example all attributes and elements are from namespace `http://www.book.cz/book/2007`.

Namespace recommendation also defines how to validate namespace definitions, attribute name uniqueness of an element and more. This can be found in [14].

2.3 XPath

There is discussed transformation language for XML in Section 2.4, XSLT. As a part of this language the language for addressing parts of XML documents has been developed too. This navigation language called *XPath* is mentioned before the whole transformation language for better understanding of transformation examples.

This implementation uses XPath in version 1.0[5] ([24]). XPath recommendation defines syntax and semantics for navigation language to address parts of XML documents. Syntax of this language is stand-alone and have nothing to do with XML syntax. XPath syntax has been inspired by URLs and filenames. Examples are shown in Figure 2.7. The first example selects a `title` element of the root `book` element, if there is any. The second example selects all `chapter` elements of the root `book` element, where exists a `title` element, which string value is equal to value `'XML applications'`.

```
/book/title
/book/chapter[title="XML applications"]
```

Figure 2.7: XPath location paths

This two examples are called in XPath terms *Location Paths* and represent the essential XPath *expressions*. These paths specifies *Steps*, how to "walk" through XML document and find specified part of the XML document.

2.3.1 Expressions

Expression is a basic syntax construct in the XPath language. Each expression is evaluated relatively to the context. The result of the evaluation is an object, which is one of these types:

- node-set(a collection of nodes without duplicates)
- boolean (true or false)
- number (a floating point number)

[5]XPath is further used to reference this version of the language

- string(a sequence of characters)

The context for evaluation consists of these parts:

- context-node
- context-position
- context-size
- variables definitions and their values
- functions
- namespace declarations

Context-node is a reference to a node in a XML document, which is considered as a current node for evaluation. *Context-position* is an integer number. It is supposed, that the context-node is inside a node-set. Nodes in this node-set are ordered and the first node has a *context-position* equal to 1, the second node 2 and so on. The *context-size* is a count of nodes in the node-set.

Variables definitions is a set of pairs variable name and variable value. These variables can be used inside XPath expressions and their values are objects, which can be of any type of XPath expression: node-set, boolean, number or string.

XPath contains definitions of many functions, which can be used inside expressions. *Functions* part of context is a mapping from function name to its definition. XSLT adds to XPath *core functions*, which have to be supported by any XPath implementation, additional functions.

The last part of context, *namespace declarations*, is a set of pairs namespace prefix and namespace URI, which is used for expanding qualified names used by XPath expressions.

2.3.2 Data model in XPath

XPath operates on a tree built from a XML document. XPath defines own data model, which is used to built this tree, and is derived from XML Information Set Mapping [24].

The XML tree contains these kinds of nodes:

- root node
- element nodes
- attribute nodes
- namespace nodes
- processing instruction nodes
- comment nodes
- text node

The root node and element nodes have a list of *child nodes* and are *parent nodes* of these nodes. Every node except root node has exactly one *parent node*. Root node is always root of the tree. So the interior nodes of tree can be only element nodes and remaining kinds of nodes are always leaves of tree, because they do not have children.

Node kind	String value definition
root node	concatenation of string values of all descendants
element node	concatenation of string values of all descendant **text nodes**
attribute node	attribute value
namespace node	namespace URI
processing instruction node	content
comment node	content
text node	content

Table 2.1: String value of objects in XPath

XPath defines a *string value* of each of this node kind. The definition of this value is shown in Table 2.1. XPath also defines ordering called *document ordering* and it is defined for all nodes in a tree. This ordering is defined by occurrence of nodes in a XML document. The first node of the document is always root node. Then follow its descendants. The element nodes occur before their children. Then follow their namespace prefix definitions, attributes and then remaining kinds of nodes in order as they occur in document. Order of attributes and order of namespace definitions is not defined.

1. **Absolute location path:** `/step1/step2/step3`
2. **Relative location path:** `step1/step2/step3`

Figure 2.8: Location path types

2.3.3 Location paths

Paths are essential expressions of XPath. Every path consists of *location steps*, which are separated by '`/`'. Location path is evaluated from the left to the right. Each step is evaluated relatively to the context node, which changes during the evaluation process. The context node for evaluation of a location path is defined by the context (see Subsection 2.3.1).

If the location path starts with '`/`' (Item 1 in Figure 2.8), then the path is called *absolute location path* and the context node for location steps is the root node of the document containing context node. Otherwise the location path is called *relative location path* (Item 2 in Figure 2.8) and the context node is taken from the context.

Each step is evaluated relatively to the context node and the nodes selected by a step are context nodes for evaluation of the following step. The nodes selected by the last step are a result of entire location path.

Evaluation of a location path can be simplified into instructions shown in Figure 2.9.

2.3.4 Location steps

Location step has three parts: axis, node test and zero, one or multiple predicates. *Axis* specifies the base set of nodes, which can be in a result of a location step. The detailed description of all defined axes can be found in Subsection 2.3.5. *Node test* defines a simple filter on the node set returned by an axis. *Predicates* contain XPath expressions, which are used for accurate filtering of nodes, which are returned.

XPath defines a *principal node type* for each axis. The *attribute* axis has a *principal node type* attribute, the *namespace* axis has a *principal node type* namespace. For all other axes the *principal node type* is element. This node type is important for evaluating node tests in a expression.

Node test can be of two kinds: *name test* and *node type test.*

A *name test* is consisted of a fully qualified name, a namespace URI and

1. add the root node (for an absolute path) or the context node (for a relative path) to the temporary node set
2. loop for each step of the path
3. get all nodes of the axis, which is evaluated relatively to the all nodes in the temporary node set
4. filter all these nodes by the node test
5. for each node evaluate all predicates (predicates can contain location paths, which are evaluated relatively to the examining node)
6. clear temporary node set and put nodes, which has passed by all predicates, into it
7. if there is any next step, go to Item 2
8. return temporary node set as a result of the location path

Figure 2.9: Location paths evaluation

a name. A node passes the *name test*, if the type of the node is the principal node type of the axis and if the qualified name of the node is equal to the qualified name in the name test. The name in a name test can be replaced by '`*`', which means that name is not specified and all nodes principal node type are returned. Asterisk mark can be also used only for local name part of fully qualified name to filter nodes, which are within specified namespace.

A *node type test* can be of these types:

- comment()
- text()
- processing-instruction()
- node()

The first three types filter an axis to the nodes, which are of the corresponding type (a comment node, a text node or a processing instruction node). The last type, *node()*, does not filter axis at all. All nodes selected by an axis are returned in result.

The location step shown in Figure 2.10 specifies *child axis* (all axes are described in Subsection 2.3.5). The nodes from this axis are filtered by the node test, which specifies that only elements with name '`book`' will be in a result. Only elements are returned, because the *principal node type* of the child axis is the element type. At the end, the nodes are filtered by predicate. There is an expression in the predicate, which decides to return only nodes, which have a title child element with a content equal to '`XML`'. The predicate in this example contains another one location path, which is evaluated relatively to the nodes selected by the axis and the node test.

2.3.5 Axes

The XPath [24] defines following axes for selecting node sets:

- child

```
child::book[child::title='XML']
```

Figure 2.10: Location step example

- parent
- attribute
- namespace
- ancestor and ancestor-or-self
- descendant and descendant-or-self
- following
- preceding
- following-sibling
- preceding-sibling
- self

Child axis selects all child nodes of context node. If the axis is omitted in a location path, the child axis is the default one for evaluation.

Parent axis contains for each node except root nodes one parent node. That is a result from Subsection 2.3.2

Attribute axis selects all child attributes of current context node and *Namespace* axis selects all namespace prefix definitions of it. Both axes contain any nodes if and only if the context node is an element. Otherwise it is empty.

Ancestor and *ancestor-or-self* axes select parent node, parent's parent node and so on. These axes always contain root node. The root node does not have a parent node, thus the recursive definition always stop at a root node. *Ancestor-or-self* axis furthermore contains current context node.

Descendant and *descendant-or-self* axes contain child nodes of the context node, children of each child and so on. These axes never contain attribute nodes and namespace nodes. Again, *descendant-or-self* axis contains current context node besides others.

Following axis and *preceding* axis contain all nodes, which follow, respectively precede, the context node in document order. These axes does not contain descendants of the context node and does not contain attribute and namespace nodes.

Following-sibling axis and *preceding-sibling* axis contain following, respectively preceding, siblings of the context node. Siblings of the node are nodes with the same parent node. These axes are empty, when the context node is an attribute or namespace node, because document ordering is not defined for set of attributes of a node and set of namespace prefix definitions (see Subsection 2.3.2).

Self axis always contain exactly one node, which is the context node.

Each axis is *forward axis* or *reverse axis*. The *forward axis* returns nodes ordered in document order and these are axes, which returns context node or nodes, which follows context node in document order. Therefore ancestor, ancestor-or-self, preceding, and preceding-sibling axes are *reverse axes*. These axes return nodes in reverse document order. This fact influences the XPath context-position evaluation. Context-position in a result of a reverse axis is counted on nodes ordered in reverse document order.

The most of axes are shown in Figure 2.11. The picture does not contain variants of axes with "-self" suffix. These axes would contain context node moreover. The attribute axis and namespace axis are not presented too.

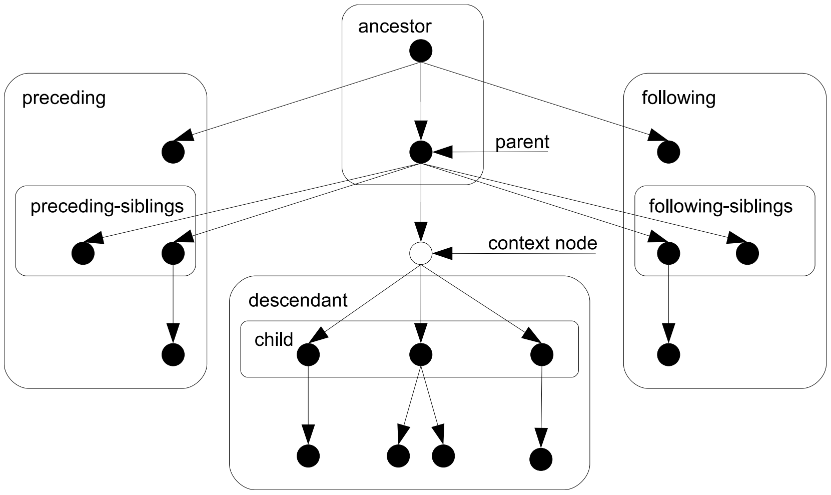


Figure 2.11: XPath axes example

2.3.6 Abbreviation in location paths

XPath defines abbreviations for most usually used expressions. All abbreviations are shown in Table 2.2.

Figure 2.12 shows an example of abbreviations.

The long syntax	Abbreviation type	The abbreviated syntax
parent::node()	location step	..
self::node()	location step	.
descendant-or-self::node()	location step	//
attribute	axis	@
child	axis	default axis, does not have to be specified

Table 2.2: Abbreviations in XPath

Abbreviated location path

```
//para/../text()
```

Unabbreviated location path

```
/descendant-or-self::node()/child::para/parent::node()/
  child::text()
```

Figure 2.12: Example of abbreviated location path

- select the last book in a document
  `/book[last()]`

- select the second book in a document
  `/book[2]`

Figure 2.13: Predicates with number value

2.3.7 Predicates

An axis (Subsection 2.3.4) together with a node test produce a node set, which can be furthermore filtered by predicates. A predicate is an XPath expression in square brackets (`[]`). Its result is converted into the boolean value by XPath *boolean* function, which is a part of XPath *core functions*. Description of this function can be found in Subsection 2.3.8. Only if the predicate is evaluated to the *true* value, the node is included into the result set.

There is one exception to the previously mentioned rule. If the result of a predicate is a number, the predicate is *true*, if the context position equals to this number. Otherwise the predicate is evaluated to *false*.

The example of the predicate with number is shown in Figure 2.13. The used function *last* is described in Subsection 2.3.8.

The first predicate of a location step filters the result set of an axis and a node step. The result set of the first predicate is the input of the next predicate of the location step. Therefore, the node is in the result set of entire location step, if it passed all predicates.

2.3.8 XPath core functions

XPath defines the set of functions, which have to be supported by any XPath implementation. The detailed description of these functions is not part of this thesis and can be found in [24]. A few of them are shown in Table 2.3.

The *boolean* function converts objects into boolean values by these rules:

- a number is true if it is not zero or NaN

- node-set is true if it is non-empty

- a string is true if its length is non-zero

Function name	Description
last()	returns the context-size
position()	return the context-position
count(node-set)	returns count of nodes in a node-set
boolean(object)	converts any XPath object into a boolean value
string(object?)	converts an object into a string value

Table 2.3: XPath core functions

The conversion of objects into a string value is shown in Table 2.1.

2.4 XSLT

For specifying formatting of XML documents the *Extensible Stylesheet Language* has been developed and a part of it *XSL templates*, shortly XSLT, has been designed. The target of XSLT is to prepare and transform XML document for final formatting process. XSLT in version 1.0 has been published and this version is used in this implementation. However, there is a new version of XSLT — 2.0. This version is mentioned further, but is not a part of this implementation. Thus XSLT abbreviation is used to reference XSLT in version 1.0 further.

A transformation described by XSLT is called *stylesheet*. Stylesheets are XML document with a specified structure defined by XSLT. The elements and attributes are defined in XSLT namespace with standard URI (see Figure 2.14). Each stylesheet contains set of *template rules*. These rules consists of two parts: *pattern* and *template*.

```
http://www.w3.org/1999/XSL/Transform
```

Figure 2.14: XSLT Namespace URI

Stylesheet is interpreted by a XSLT processor on a source document. The processor applies all template rules defined in a stylesheet on a source document and an output document is built.

The *template rule* contains *pattern*, which is an XPath expression. The template rule is applicated on a node, if the pattern matches examined node. The processing is done by finding template rule for a root node of a source

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    ...
  </xsl:template>
</xsl:stylesheet>
```

Figure 2.15: Stylesheet example

document. Template of the template rule can contain instruction to process child or descendants of the root node. When a template is applicated the context for this application is held. This context consists of a *current node* and a *current node list*.

It has been mentioned that patterns are XPath expressions. XSLT uses XPath in many attributes for selecting node, for evaluating conditions and for generating text output.

The output of the XSLT processor can be of these kinds: a XML document, a HTML document or a text document. XSLT can generate **non-wellformed XML documents**, because it does not have requirements on the children of a root node of an output document. Therefore it can generate e.g. documents with many root elements.

XSLT does not specify connection between a stylesheet and a source XML document. There is recommendation, that this should be done as described in [3]. This connection can be done by *stylesheet* processing instruction in the source XML document.

2.4.1 Stylesheets

Stylesheets are XML documents with *stylesheet* root element from the XSLT namespace (Figure 2.14). This element has to contain *version* attribute with specified version number. For XSLT in version 1.0 it has to contain `'1.0'` string. Furthermore stylesheet element can contain different controlling instructions. The example of a stylesheet with one specified template rule is shown in Figure 2.15.

Controlling instructions

The XSLT processor can be instructed to format an output document in many ways. *Strip-space* and *preserve-space* instructions can be used to control processor, when the whitespaces in a source document or in a stylesheet should be stripped or written to the output document. Format of numbers can be controlled by *decimal-format* instruction. The detailed description of these instructions can be found in [25].

Stylesheet importing and including

Stylesheet can contain *import* or *include* instructions, which reference by URI another stylesheet document. Templates from *included* stylesheet are added to the set of templates defined in the main stylesheet and are used for transformation process. It is an error, if any stylesheet is included multiple times directly or indirectly.

The stylesheet can also *import* another stylesheet. Meaning of import is nearly the same like include, but imported template definitions has a lower *import precedence* during XSLT processing. It means that templates in importing stylesheet overload templates in imported stylesheet. This can be used for rewriting some part of stylesheet function. The example of importing stylesheets is shown in Figure 2.16.

The example demonstrates, that the imported template rules have lower import precedence and so the template rule for `title` element is overridden and rule from stylesheet `myarticle.xsl` is used. The detailed description of template rule searching can be found in Subsection 2.4.5.

2.4.2 XPath and XSLT

XPath language is used by XSLT to select nodes, to process conditions and to generate text values. The data model used by XPath is used in XSLT too. XSLT defines the value of the context for XPath evaluation as follows:

- context-node is the *current node*
- context-position is the position of the *current node* in the *current node list*
- context-size is the size of the *current node list*

Source XML document book.xml

```
<book>
  <title>XML</title>
</book>
```

Stylesheet book2article.xsl

```
...
<xsl:template match="book">
  <article><xsl:apply-templates /></article>
</xsl:template>

<xsl:template match="title">
  <caption><xsl:apply-templates /></caption>
</xsl:template>
...
```

Stylesheet myarticle.xsl

```
...
<xsl:import href="book2article.xsl" />

<xsl:template match="title">
  <title>Title: <xsl:apply-templates /></title>
</xsl:template>
...
```

Result of processing myarticle.xsl on book.xml

```
<article>
  <title>Title: XML</title>
</article>
```

Figure 2.16: Stylesheet importing and overriding

- variables definitions are variables defined by *variable* or *parameter* instruction
- functions consists of XPath core functions and many XSLT specific is added (details can be found in [25])
- namespace declarations contains all namespace definitions, which are in the scope of the element containing the attribute, which content is an XPath expression

2.4.3 Template rules and processing

A stylesheet contains a set of template rules: pairs of a *pattern* and a *template*. The processing begins on the list of nodes, which contain only root node of the source document. Each node is processed by matching template rule with the highest *priority*. The result of processing the node is appended to the result document. Templates usually contain instruction for selecting descendant nodes and applying the processing algorithm recursively on this list. The processing ends, when there is no source node to process.

2.4.4 Patterns

A *pattern* is a set of conditions on a node. A node *matches* a pattern, if all conditions are for the node satisfied.

Pattern grammar is a subset of the XPath language (Section 2.3). A pattern is a set of *location path patterns* separated by '`|`'. These location path patterns are considered to be alternatives. It is evaluated as a boolean `or` operator. A *location path pattern* is an XPath location path, which uses only the child or the attribute axes. No other axes are permitted. However, predicates used in steps are not limited to these axes.

A pattern cannot use the *descendant-or-self* axis, but the '`//`' separator between steps can be used. The location path pattern is evaluated from the right to the left as follows:

- a node matches the pattern if and only if the rightmost step of the pattern matches the node
- if there is another step separated by '`/`', then the *parent node* must match the step

```
/book//section[title="XML"]
```

Figure 2.17: Location path pattern example

- if there is another step separated by '//', then there must exist an *ancestor node*, that match the step
- the definition repeats for all next steps from the right to the left of the location path pattern

The example of the location path pattern is shown in Figure 2.17. This pattern matches any `section` element in the source document, which has a `title` element and its string values is equal to 'XML'. The section can be any descendant of the root element `book`.

If the predicate is presented in a step, the node, which matches the node test, is the *context node* for predicate evaluation and siblings of this node, which match the node test, are content of the *context node list*.

2.4.5 Template rules

Subsection 2.4.3 mentioned, that the template rules are the basic construct for XSLT processing. The rules are defined in a stylesheet by `template` element with specified `match` attribute, which contains a *pattern*. When the *current node* matches the pattern, the template is interpreted by processor.

The template can contain `apply-templates` element, which instructs processor to recursively search for template rules for all children of the current node. This element can have specified `select` attribute. The content has to be an XPath expression, which is evaluated into a node-set. Then the searching for template rules is done only for these nodes. The expression of the `select` attribute are not limited and so it can select ancestor nodes, which can lead to nonterminating loops. The implementation should detect these loops.

That is possible that more than one template rule matches a node. Than the *priority mechanism* is defined and the template rule with the highest priority is used for transformation. If the processor cannot decide, the error can be reported and transformation is stopped. Or processor can choose from the remaining ones the template rule last occurred in the stylesheet and recover from this error. The following rules are used for decision:

- the all imported template rules, which have lower import precedence, are removed from the deciding process
- the priority is obtained from the `priority` attribute of the `template` element
- otherwise the default priority is computed:
  - priority is computed separately for all alternatives
  - paths, which consists of only one step with child or attribute axis and a node test (not using asterisk mark), have the priority 0
  - if the pattern is the same like previous, but uses asterisk mark for local name in the node test and specifies namespace, it has the priority -0.25
  - if the entire node test is specified by asterisk mark, the priority is equal to -0.5
  - otherwise the priority is set to 0.5

The algorithm for decision prefer more specific patterns. The most specific pattern, which contain more than one step, cannot be distinguished and have default priority 0.5.

2.4.6 Built-in template rules

XSLT defines set of standard built-in template rules. These template rules are used for nodes of the source XML document, when user does not define any template rule, but processor searches for a rule for the node. It defines behaviour for these nodes. The definitions are shown in Figure 2.18 and follows their semantics in the same order:

- Text nodes and attribute values are generated to the output document.
- Processing instruction nodes and comment nodes are ignored.
- Other nodes (element nodes and document root node) does not generate output, but the processor is instructed to search for templates recursively by apply-templates instruction. This instruction is described in the following section.

Details about used XSLT elements can be found in the following sections.

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="processing-instruction()|comment()"/>

<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

Figure 2.18: Built-in XSLT templates

2.4.7 Named templates

The Subsection 2.4.3 described the processing model of XSLT. The template rules are applicated on matching nodes. However, templates can be also instantiated by their names. Element `template` can contain `name` attribute. Its content is a qualified name, which can use namespaces ([14]). A template can contain `call-template` element, with specified `name` attribute, which specifies the template to be invoked. It is instantiated with the same current node and current node list as is for `call-template` element.

The priorities and modes are not considered during call process. That is an error, where exists more than one template, with the same name and import precedence. However, the named templates can be overridden by importing as described in Section 2.4.1.

2.4.8 Creating the result

The creating of the result can be in a template described in two very different ways: the *literals* can be used or it can be described by XSLT elements.

Using literals

The elements and attributes inside a template, which are not from XSLT namespace, are not interpreted by processor and are written to the output document. The content of an non-XSLT element is again considered to be a template and creates a content of the element.

Namespace nodes, which are the children of a non-XSLT element and not defines alias for XSLT namespace, are created in the output document too.

Example of generating result by literals is shown in Figure 2.19. The `article` element is a non-XSLT element, also called *literal element*. Its child attribute node and namespace node are literals and are created in the result document too. The `article` literal element contains the mixed content. Text node, which is the child of this element, is also created in the result document.

Template

```
...
<xsl:template match="book">
  <article s:owner="nobody"
    xmlns:s="http://www.book.cz/owner">
    Text of the book:
    <xsl:apply-templates />
  </article>
</xsl:template>
...
```

Result

```
...
<article s:owner="nobody"
  xmlns:s="http://www.book.cz/owner">
  Text of the book:
  ...
</article>
...
```

Figure 2.19: Template with literals

2.4.9 Creating the result by XSLT elements

Element and attribute nodes mentioned in Section 2.4.8, which can be generated by literals can be also generated by XSLT elements.

Elements in the result document can be created by XSLT `element` element, attributes by `attribute` element. The `attribute` elements must be the first child elements of the parent.

Text nodes can be created by `text` element, which can specify, how the whitespaces are handled.

All these possibilities are demonstrated in Figure 2.20. It is rewritten stylesheet from Figure 2.19 with the almost same result. The result can differ, how whitespaces are handled by processor. The details about whitespaces handling can be found in [25].

```
...
<xsl:template match="book">
  <xsl:element name="article">
    <xsl:attribute name="s:owner"
      namespace="http://www.book.cz/owner" />
    <xsl:text>Text of the book:</xsl:text>
    <xsl:apply-templates />
  </xsl:element>
</xsl:template>
...
```

Figure 2.20: Result generated by XSLT elements

The last passibility for generating result by XSLT elements is to use *copy-of* element. This element has to contain the `select` attribute, which contain an XPath expression. The selected fragment of the source document is created in the output document. This means that the node set selected by expression is created in the output document in the source document order. The tree fragment is the whole copied to the output. When the result of the expression is of other type, the object is converted by the XPath *string* function and the text node is created in the output document. This is very similar to the *value-of* element described in Section 2.4.10.

2.4.10 Generating text by XPath

XSLT uses XPath also for generating text values. The XPath expressions can be used for this purpose in two ways: with *value-of* element or in *attribute value templates.*

Value-of element

The *value-of* element generates a text value, which is a result of the XPath expression. The expression is written in the *select* attribute of this element. The expression is evaluate relatively to the current node and the current node list. The examples is shown in Figure 2.21. This example for each `book` element creates the `article` element in the result. Each `article` element contains `name` element. The content of these elements is generated by *value-of* element. It generates a text node, which contain the string value of nodes selected by XPath query `title`. This query can be rewritten into unabbreviated form (as described in Subsection 2.3.4 and following sections):

```
self::node/child::title
```

The query is relative to the *current node* and selects its child `title` elements. It does not select attributes or other types of nodes, because *element node type* is the *principal node type* of a *child* axis (details can be found in Subsection 2.3.4).

```
...
<xsl:template match="book">
  <article>
    <name><xsl:value-of select="title" /></name>
    <xsl:apply-templates />
  </article>
</xsl:template>
...
```

Figure 2.21: Value-of element example

Attribute value templates

This mechanism is able to compute text value of attributes, which are interpreted, by XPath expressions. It can be used e.g. by attributes of literal elements (described in Section 2.4.8). An XPath expression has to be surrounded by curly braces ({}). The content is then interpreted in the same way as the *value-of* element. The example is show in Figure 2.22. This example is very similar to the one shown in Figure 2.21. However, it generates name of the book as `name` attribute. The content is the same in both examples.

```
...
<xsl:template match="book">
  <article name="\{title\}">
    <xsl:apply-templates />
  </article>
</xsl:template>
...
```

Figure 2.22: Attribute value templates example

2.5 Variables usage

XSLT introduces mechanism of *variables*. The *variable* is the pair of a string name and a object, which can be any of the XPath object type. It can also point to *result tree fragment*, which is a new type defined by XSLT. The details about this can be found in [25].

2.5.1 Variables definition

Variables can be declared by two elements: *variable* and *param* element. Both has to contain `name` attribute with specified variable name. The content can be selected by XPath expression in the `select` attribute or as a content of the defining element.

The difference between variable and param is, that the param defines only the default variable value. It is usual, that the value is redefined *with-param* element, which can be contained by *apply-templates* element or by *call-template* element.

2.5.2 Variables value usage

The variables can be used inside all XPath expressions in templates. Variable name has to be prefixed by `'$'` char.

The example is of a variable is shown in Figure 2.23. The example demonstrates many features of XSLT. It uses recursive template calling for simulating classic *for* cycle from 1 to `$count_of_spaces`. It is a variable declared by `param` element. And for each step of the cycle it writes one space char into the output document. So this template has to be called with specified parameter and then it writes specified number of spaces to the output doc-

```
...
<xsl:template name="spaces">
  <xsl:param name="count_of_spaces"/>
  <xsl:if test="$count_of_spaces > 0">
    <xsl:text> </xsl:text>
    <xsl:call-template name="spaces">
      <xsl:with-param name="count_of_spaces"
        select="$count_of_spaces - 1"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
...
```

Figure 2.23: Variable example

ument. This also demonstrates how to simulate for cycle in XSLT, because this construct is not defined in XSLt.

2.6 Other constructs in XSLT

It has been mentioned many elements of XSLT and their semantic. However, XSLT defines many more elements, which will not be described in this thesis in details. It can be found in [25].

It means especially elements for conditional processing and for repetition. The Figure 2.23 shown how to simulate classic *for* cycle, which is not included in XSLT. However, XSLT contain `for-each` element, which repeat its content for each node in a selected node set. The evaluation of templates can be controlled by `if` element and by `choose` element, which are very similar to classic “if” and “switch” C-like statements.

XSLT was intended to prepare XML documents for publishing, so it can sort the output by `sort` element and text can be formated by `number` element.

Chapter 3

Implementation prerequisites

In this chapter the chosen technologies and third party products used during implementation are described. The chosen relational database engine is presented in Section 3.1. Its procedural extensions to SQL, Java programming language, XML tools and integrated development environment (IDE) are presented too.

3.1 Firebird relational engine

As a relational database the Firebird ([8]) has been chosen. This is an open source project, which has been established at the end of the year 2000. In this year Borland corporation took source codes of their Interbase SQL database product to the open source community. This code became a foundation of the future Firebird project. Interbase is still developed by Borland corporation, but nowadays Firebird is completely stand-alone project and is developed in other direction.

During the implementation of this work version 2.1 of Firebird engine has been released and this version is used by this implementation. This engine is nearly SQL-92 compliant engine and it has also many SQL:1999 ([4]) and SQL:2003 features like sequences, case expressions, etc. Engine also contains procedural extension to SQL, which is used in stored procedures and triggers. Stored procedures are highly used by XSLT implementation to reduce traffic between client and SQL server.

This engine has been chosen, because it is completely open source project. It is released under Interbase Public Licence, which let to use this engine in commercial projects too and engine is not limited in any feature like other free versions of Oracle or Microsoft SQL. Firebird is a multiplatform engine, which can be run on Microsoft Windows operating system and on Linux operating system.

The documentation for this engine can be found in [9].

3.1.1 Engine connection architectures

Firebird can work in three very different scenarios:

- standard client-server
- local
- embedded

By using different scenario the Firebird can scale very well for different performance requirements and for very different size of stored data. *Client-server* scenario is similar to other world wide spread SQL databases. Client communicates over the net with dedicated stand-alone server.

In *local* architecture the client communicates with stand-alone server by special local network protocol. This is supported only in Microsoft Windows operating system. The benefit of this mode is reduced network traffic.

The third scenario of the Firebird engine usage is called *embedded*. This is very specific feature of this SQL engine. When this scenario is used, the SQL server is not stand-alone. It is compiled as a dynamically linked library and is tightly connected to the client application.

Scenarios can be chosen in any time of client application development, because it is done, mainly, by using different connection string to connect to the server. By using Firebird as a hosting relational engine XSLT implementation offers possibility to be used with the embedded server and so no server application has to be installed at all. Or it can benefit from already running Firebird database engine and use it in the client-server mode.

3.1.2 SQL procedural extensions

All contemporary relational engines offer SQL procedural extensions to support triggers and stored procedures. These features are able to move logic directly into relational engine and these programs are run directly by relational engine.

Firebird supports basic procedural extensions mainly inspired by Pascal programming language and this language is very similar to standardized SQL/PSM language introduced in SQL 1999 and extended in SQL 2003. However, used language is not fully compatible with SQL/PSM. It does not support any object oriented features. It offers simple variables, which are declared at the beginning of procedures, cycles, if statement and more. The full description of Firebird PSQL language can be found in [7].

Stored procedures in Firebird can be of two kinds: standard procedures and select procedures.

Select procedures

Select procedures return data in a table style. They are used in place of relational tables in SQL selects. They have special structure. They usually contain a cycle, which fills output variables. After filling the parameters the `suspend` statement is called, which blocks executing of procedure and returns output parameters as a new row of result set of procedure. When client asks for a new row, the procedure continue in executing directly after the suspend statement. The result set ends where procedure ends. These procedures can be used for generating data, for transforming table data, where views cannot be used, etc.

The example of select procedures is shown in Figure 3.1. This procedure loads name of all relations in database (`RDB$RELATIONS` is a name of system table in Firebird database catalog).

Standard procedures

Standard procedures can return only set of output parameters or it can perform an action and return nothing. The return parameters are marked as output parameters of select procedures. However, these procedures should not contain `suspend` statement.

3.2 IBExpert

Development of database structures and especially stored procedures has been done in *IBExpert* tool [10]. This is a fully featured integrated development environment for Firebird database engine. It offers table and view visual modelling, user friendly altering and many more. It has been also used

```
DECLARE RNAME CHAR(31);
DECLARE C CURSOR FOR ( SELECT RDB$RELATION_NAME
    FROM RDB$RELATIONS );
BEGIN
  OPEN C;
  WHILE (1 = 1) DO
  BEGIN
    FETCH C INTO :RNAME;
    IF (ROW_COUNT = 0) THEN
      LEAVE;
    SUSPEND;
  END
  CLOSE C;
END
```

Figure 3.1: Firebird Select procedure

for creating stored procedures of XSLT implementation. This software has been chosen because it can be freely used for personal developing and for education.

3.3 Java

Used relational database does not offer any facility to communicate for example with operating system, files, etc. For loading XML files into database additional programming platform has to be used.

In this XSLT implementation Java 6 SE is used, because it is complete object oriented fully XML featured platform with standardized relational database access. Java is partly open source platform maintained by Sun Microsystems. It is used for XML parsing and generating and also for complicated algorithms, which cannot be easily implemented directly in relational engine.

Java is distributed in many different packages and in this implementation Java Standard Edition is used in version 6 from Sun Microsystems [11]. Database connection to the Firebird server is done by standardized JDBC technology. Firebird project offers for this technology own JDBC driver called Jaybird [13], which is fully featured driver for Firebird server.

When JDBC is used it is actually very easy to change relational database implementation. So it is possible to use Java parts of the XSLT implementation with another SQL engine, e.g. it is possible to load XML data by this component into Oracle or MS SQL database engines.

3.3.1 XML API and Xerces

Java 6 Standard Edition offers full XML API[1]. It also contains XML parser, but this implementation does not offer all XML features, especially namespace processing. Because of this other implementation is used for XML parsing, Xerces for Java in version 2.9.0 [22].

Xerces is fully features XML parser, but in this thesis only its SAX parser ([16]) is used. However, XSLT implementation is not dependent on Xerces and any SAX implementation can be used for parsing XML documents by switching the implementation in Java XML API.

3.3.2 Spring Framework

The configuration and dependency management in object application can be done in many different ways. This XSLT implementation uses core of Spring Framework in version 2.0.2 [18]. This is an enterprise oriented framework, which is used without any application server during runtime. However, this framework offers XML based configuration and *dependency injection* mechanism. It means, that dependencies between Java classes are configured in XML files and Spring is responsible for instantiating all these classes and connecting them together during runtime.

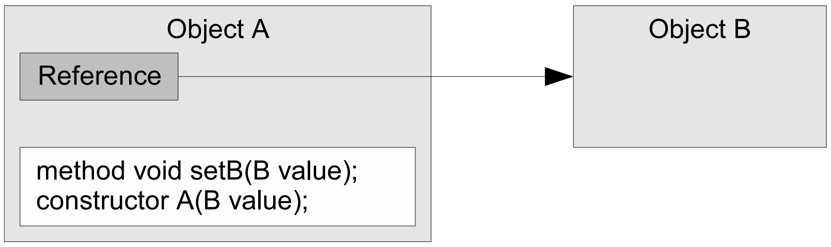


Figure 3.2: Dependency injection example

[1]Java API for XML Processing, see http://java.sun.com/xml/

```
<bean id="b" class="B"/>

<bean id="a" class="A">
  <constructor-arg ref="b"/>
</bean>
```

Figure 3.3: Spring configuration example

Example of dependency problem is shown in Figure 3.2. There are two instances in runtime, object A and object B. Object A is dependent on object B and needs a reference to the object B. This dependency is configured in Spring configuration file shown in Figure 3.3. Spring during runtime creates these two objects and object A is initialized by its constructor with the first parameter set to object B.

Details about dependency injection and entire framework can be found in [18].

3.4 Eclipse IDE

The development process has been done in Eclipse integrated development environment (IDE) version 3.2.2 [5]. This is a world wide used Java development tool with a huge amount of plugins to support entire development process. The enclosed CD mentioned in Appendix E contains Eclipse and used plugin too.

3.5 Programming tools

The other used tools are described in this section, which has been used to simplify development process, packaging process and other administration, also to ensure versioning.

Some parts of implementation has been modelled in UML[2] (e.g. object model for modelling SQL selects in XPathConvertor, see Subsection 6.2.2) and for this process the open source UML designer has been used called ArgoUML ([1]). However, UML has been used only for designing and for documenting. The generating of source code has not been used at all.

[2]Unified Modeling Language, see http://www.uml.org/

3.5.1 Maven

The building and packaging process of a Java application is possible by Eclipse IDE itself. But this is is not appropriate for projects with many third party dependencies and with several separated modules. The Maven project in the current version 2.0 [2] has been chosen for these purposes. Project configuration is stored in a Maven POM file[3], which is the central configuration point of an application and Eclipse project files are automatically generated from this file. Then it is possible to run and debug Maven based project directly from IDE. Maven is also responsible for managing dependencies of application. The dependencies are defined in POM file too and Maven downloads necessary files from central Maven repository by internet connection.

Maven community has created large central repository with huge amount of applications and components. Then any project, like this XSLT implementation, can be very easily uploaded to this repository and served to all Maven users.

3.5.2 Subversion

For central storage repository of source code files with versioning features the Subversion [20] has been used and all project files are stored there. The Eclipse plugin Subclipse [19] has been used for comfortable Subversion usage, committing changes, etc. Due to Maven usage the files of dependencies and generated files are not stored in Subversion repository. The Subversion updates, commits, etc. can be done by Maven too, but this feature has not be used, because Subclipse plugin is better integrated into Eclipse IDE.

3.6 ToXgene

The entire implementation was tested on a large XML data generated by ToXgene generator [21]. This is the template based XML documents generator. The used templates for generating XML data can be found in Appendix D. The ToXgene is able to generate complex XML content, with mixed content too, supports integrity constraints and many more. In this thesis the ToXgene has been used only for generating a large XML document to demonstrate robustness of implementation.

[3]Project Object Model, see `http://maven.apache.org/pom.html`

3.7 Tools usage

Majority of tools is not necessary for XSLT implementation in a relational database environment. Maven, Spring Framework, Subversion has been used because of purpose to find out the learning requirements to establish large enterprise project with usage of open source projects, a project with prepared team infrastructure and ensured functioning of building process, versioning process, etc.

Chapter 4

Implementation top view

The idea, how to implement XSLT with relational engine, is presented by Figure 4.1. The source XML document and the stylesheet document has to be imported into relational engine. Both these documents are parsed by XML parser and stored in tables. Then the stylesheet has to be prepared for interpreting and then the stylesheet is interpreted directly in the relational engine. The output document is stored in the relational storage as a new XML document. The output document is then exported out from the relational engine into the XML file. The XSLT implementation has a user interface and this interface controls all parts of XSLT implementation.

4.1 Implementation structure

Entire implementation can be divided into several parts. Figure 4.2 shows all these parts.

The starting point of the implementation is *XML importer*. XML importer together with XML exporter forms interface of XML storage in the relational database. Both these components are written in Java. *XML importer* parses source XML document and stores its data in relational database. *XML exporter* is responsible for exporting XML documents from relational database into XML files.

Next part written in Java is *XPathConvertor*. This component is described in Subsection 6.2.2. In simplicity, this component is responsible for converting XPath expressions used in XSLT stylesheet into SQL. Result statements are used during XSLT processing for navigating in XML documents, computing values for destination document, etc.

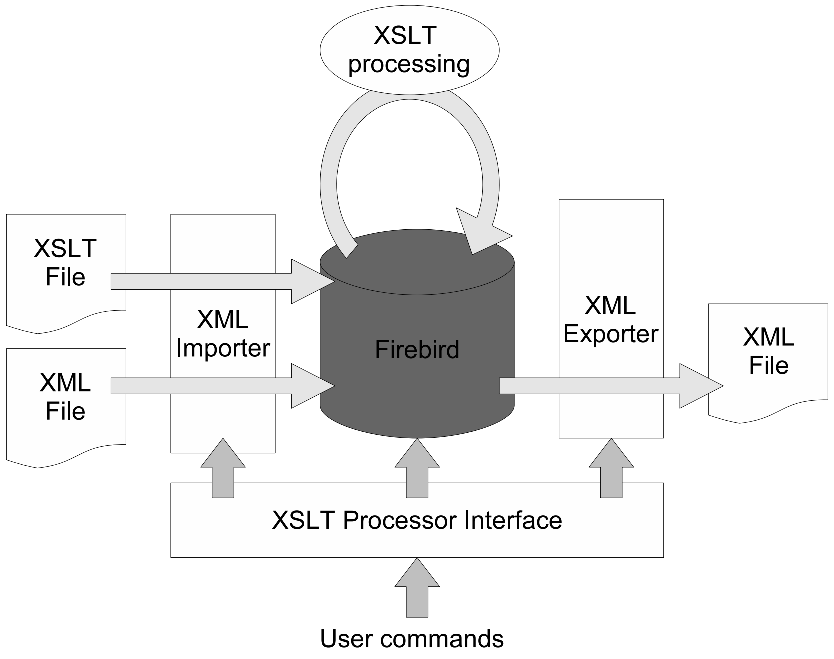


Figure 4.1: XSLT processing schema

Next parts of XSLT implementation are implemented in the relational database. This covers storage XML tables for storing decomposed XML data, tables for XSLT processing support like temporary values, XPath SQL selects, etc. Another part of implementation are *stored procedures*. By this instrument core XSLT processing is implemented directly in relational engine. Views defined in database are the last part of implementation. Views help to simplify stored procedure code by moving logic into views from stored procedures.

In the next section the main targets are discussed. Targets influenced very much decisions, how to implement XSLT processing and what technologies should be used.

4.2 The design of XSLT implementation

There are many different XSLT implementation, commercial or free and open-source too world wide. But several ideas of this XSLT implementation are not contained in any of them. Some features, which are significant for

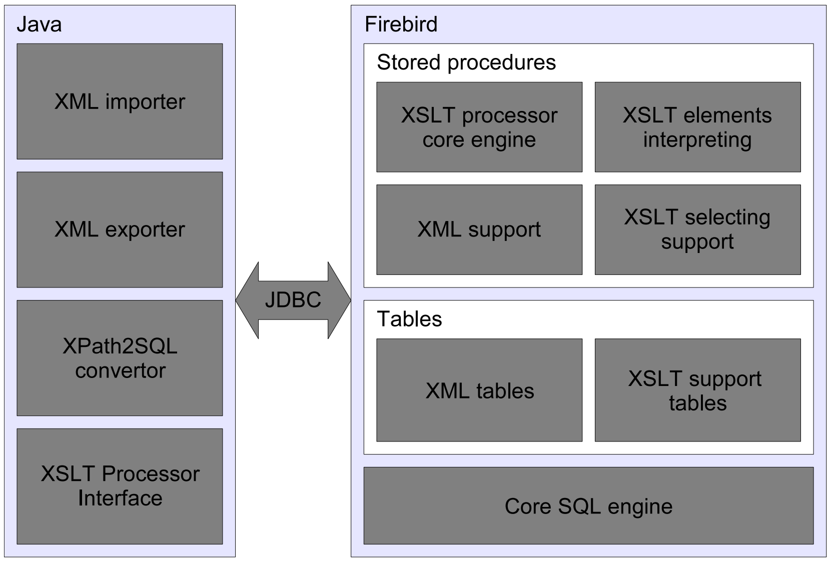


Figure 4.2: Top view of XSLT implementation

this XSLT implementation, are discussed in the following sections.

4.2.1 Large XML documents processing

The XSLT standard recommends to build representation of the source XML document in memory. It is very useful for navigating in XML data during XSLT processing. But this is impossible to be done for very large XML documents. XML is already used for storing generated data and this data can be very large, in units gigabyte and more. This implementation should not have requirements (mainly memory requirements) derived from source XML files size. This is already done by relational database usage. Relational database are very robust and are able to process huge amount of data.

4.2.2 Current relational databases usage

Relational databases are spread all around the world and are used for storing different types of data. It is natural, that there is an inclination to use these system for storing XML and for processing XSLT too and not to set up new XML databases. Moreover, relational databases and SQL processing is very well examined.

4.2.3 Hardware of current relational databases

With previous item is related usage of hardware of relational databases. This hardware is usually very huge and prepared for processing large amount of data. And XSLT processor can be next application, which can take advantage of this robustness.

4.2.4 Integration with other systems

Usage of relational database implicates integrating of systems working with relational data. This integration can be extend by XML data integration. This can be easily done by storing XML data directly in relations and this data can be easily prepared by XSLT stylesheets. Results can be used by other programs or exported to other tables used by other systems.

On the other hand, implementing XSLT in relational databases brings new problems, which dit not exist in standard implementations.

4.2.5 Work with XML stored in relations

XSLT recommendation advises to build in-memory representation of the source XML document. Generally, the widely used in-memory representation for XML data is DOM standard[1]. By this, in principal, object mapping an application creates a tree of nodes and can easily use e.g. pointer to some important node in XML tree. This cannot be efficiently implemented in relational database and application has to work with data in a batch-style. This means that application should endeavour to retrieve as much nodes as possible by each select statement and then process these nodes by a cycle.

[1]Document Object Model, see `http://www.w3.org/DOM/`

Chapter 5

XML storage implementation

Before XSLT processing XML data has to be decomposed into the relational form. This conversion can be done in many different ways. Possible mappings has been described in [23] and these possibilities are described in the following section. Section 5.2 describes chosen XML relational mapping, which is used by XSLT implementation.

5.1 Possible relational mappings

In general, the relational mappings can be divided into two groups: *generic mappings* and *schema driven mappings*.

5.1.1 Generic mappings

Generic mapping is not influenced by a structure of stored XML documents. Structure of relational tables is fixed and can store any XML document. Disadvantage of these mappings is, that structure of tables has to be very universal and usually is very similar to tree storage in relational databases. It means, that the tree is decomposed into edges and these edges are stored in tables. It leads to often join usage in used queries and database structure does not show the structure of stored data.

On the other hand, databases with generic mapping are able to store any XML document, data driven or text oriented documents, and are able to store XML documents with mixed content.

5.1.2 Schema driven mappings

All disadvantages of generic mappings are advantages of schema driven mappings and vice versa. Structure of relational tables is influenced by structure of stored XML documents. Therefore the structure of stored documents has to be known before processing these documents and the tables has to be created from this structure. If new structure appears in requirements, new tables has to be created.

Advantages of schema driven mappings are, that structure and names of tables can express structure of stored data and can be very easily recognized by humans and usually not all edges of XML tree are represented by connection between relational tables. For that reason joins in queries are not so often like in queries for generic mappings and are usually much simpler.

It has been mentioned first disadvantage, that the structure of stored documents has to be known before processing documents. The second disadvantage of these mappings is, that it is quite difficult to store text oriented documents in these mappings. Their structure is usually very complex and hierarchy is very deep. Then schema driven mapping does not have advantage, that the queries does not use many joins and queries are very similar to queries for generic mappings. The third problem is mixed content in text oriented documents. Mixed content is not usually supported by schema driven mappings.

It may seem, that the schema driven mapping has too many disadvantages, but it can be used for mapping XML to the existing relational structures, e.g. for importing and exporting data from relational databases into XML documents.

5.1.3 Indexing XML data

Tree structure is not very common for relational databases and SQL language before SQL:1999 standard it did not have support for recursive querying ([4]). For this purpose special indexing has been developed, e.g. regions pair, path indexes, etc. All these structure are described in [23]. These structures allow to evaluate recursive queries by SQL queries without recursive construct, e.g. all descendants, all followings nodes in document, etc.

5.2 Implemented generic mapping

Previous section described two possibilities how to store XML data in a relational database and this section describes chosen relational mapping, which is used by the XSLT implementation.

A XSLT processor is intended to process any XML document. That is why the generic mapping has been chosen for XML storage of the XSLT processor. Any XML document can be stored by this mapping, data driven or text oriented. Because implementation of XML storage is not target of this thesis and because generic mapping can be easily implemented, generic mapping is used.

5.2.1 Database structure

Used generic mappings uses combination of many features described in [23]. The database structure is derived from XPath XML data model, where all nodes of XML tree are of these kinds: root, element, attribute, text, namespace, processing instruction and comment node. The overview of entire database structure used by implementation can be found in Appendix B

Document table

The list of all stored documents is held by `XML_DOCUMENT` table. Each document has assigned *document ID*, which is the generated integer number. This ID is used for identifying all data in the remaining tables of XML database. This ID is also used by user interface mentioned in Section 8.3 to identify working documents. Name of stored document is derived from filename given to *XML importer*. XSLT implementation generates results of XSLT processing and these new documents have name always set to `'XSLToutput'`.

XML nodes ID

For the purpose to exactly identify a node of a XML document the generated *node ID* is used in the database. Each record of database, which represents a node, has a column for *node ID*, which is always called *XID*. The XID is unique in a XML document. The XID is an integer number and is assigned successively to nodes of a XML document and these nodes has to be ordered in document ordering (see Subsection 2.3.2). The first node of a XML document has always XID 1, following node in document ordering has XID 2 and so on.

The root node of XML document is not represented by a record of any table in database. It is only used by all nodes view (described in Section 5.2.1) to be consistent for evaluating XPath queries. In this view the root node record is added synthetically with XID set to −1 (it is consistent with parent IDs described in Section 5.2.1).

The XML nodes ID principles are demonstrated on XML example shown in Figure 2.2. The XML tree of this document is shown in Figure 5.1.

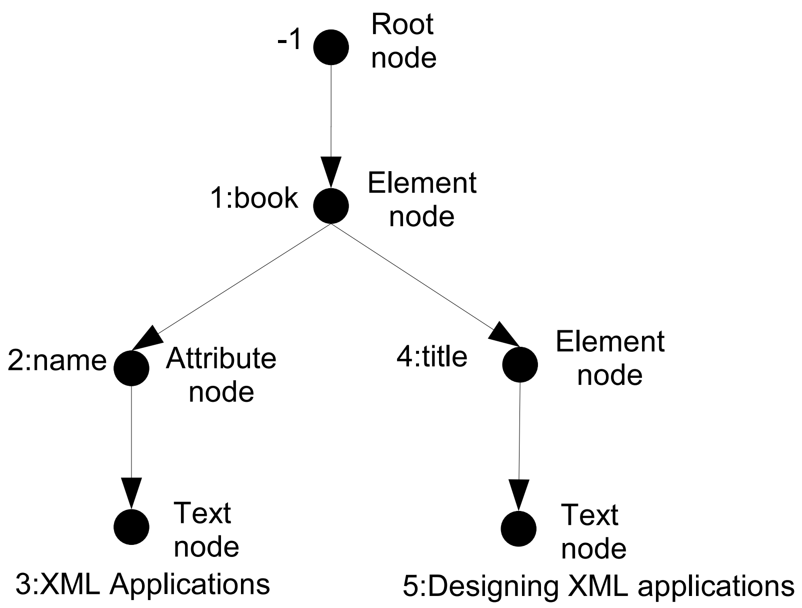


Figure 5.1: XML tree with XIDs

Tree storage

A tree of an XML document has to be decomposed into relations without loss of information. Section 5.2.1 has described, how the ordering is stored in database by XID. The parent-child relationship of tree is stored for each node by *parent ID*. The parent ID is XID of the parent node. Top-level nodes of a XML document has a parent ID set to −1, which is the XID of a root node element.

Dials

Namespace URIs and names used by elements or attributes are stored in similar tables, `XML_NAMESPACE` and `XML_NAME`, which consists of ID and value. It reduces size of database to store names and namespace URIs only once. However, the joins in queries has to be used very often.

Element node table

This table contains all element nodes of all XML documents stored in database. Table is called `XML_ELEMENT`. It contains *document ID* and a node ID, which is called `XID` (described in Section 5.2.1). Furthermore, it contains a region end XID called `ENDID` (see Section 5.2.2), a namespace ID, a name ID, a path ID for the simple path index (see Section 5.2.2) and a parent ID.

Attribute node table

This table contains all attribute nodes of stored XML documents. Its name is `XML_ATTRIBUTE` and its structure is very similar to element table. However, the simple path index is not used for attributes. Attributes do not have a *region* (see Section 5.2.2) and the content of an attribute is stored directly in the table in the column called `CONTENT`.

Text node table

All text nodes of all documents are stored in `XML_TEXT` table. The text node can be represented by multiple rows of this table because of restrictions of the char column length in the database. This table consists of a `document ID`, a *XID*, a *parent ID* and a `CHUNK` column, which can hold maximum 256 chars.

Namespace prefix definition table

Namespace prefix definitions are hold by `XML_NAMESPACE_DEF`, despite it can be stored as normal attributes in attribute table. It contain standard columns (a `document ID`, a `XID`, a `parent ID`) and a prefix column called `ALIAS` and a namespace ID, which is referenced by prefix, in `NAMESPACEID`.

XPath tables

XPathConvertor uses two tables: XPATH_COMPILATION and XPATH_COMPILATION_TEXT. First table contains one record for each converted XPath expression and the second table then contains a text of the generated select statement. This statement is divided into the rows and each row is stored as a one record of the XPATH_COMPILATION_TEXT table.

For the purpose of row numbering, the XML_COUNTING is used together with procedures CLEAR_COUNTER and NODE_COUNTER to implement the ROW_NUMBER function (see Subsection 6.2.1).

XSLT tables

XSLT processing procedures use three specific tables: XSLT_TEMPLATE, XSLT_TEMPLATE_NODE and XML_ELEMENT_STACK. The first template is used by XSLT_PROCESS template to prepare a list of all templates in a processed stylesheet. The second table, XSLT_TEMPLATE_NODE is used for *pattern matching* (see Section 6.4). The last table is used for simulating a stack (see Subsection 6.6.2).

Document all nodes view

So there are these tables in the database: XML_ELEMENT, XML_ATTRIBUTE, XML_TEXT, XML_NAMESPACE_DEF (all these tables with their structures are shown in Appendix B. These tables are called node tables in the following text. Processing instructions and comments are not supported by this implementation. XML_DOCUMENT table contains list of all stored XML documents and there is generated primary key called docId, which is generated during importing process. This key is used to distinguish nodes owned by a document in each node table, e.g. XML_ELEMENT. All IDs in database are generated integer numbers. They are generated by SQL function called generators. There is only one exception called XID, which is used to identify a node of XML document. This ID is unique in the entire XML document, but not in the entire database. First node of document has always XID 1, following has XID 2 and so on. Therefore union of nodes from node tables filtered by docId produces entire XML document stored in database and XID produces ascending sequence of numbers, which is similar to document order of nodes in source document. This is used in all queries for ordering results by this XID. Tree structure is stored by parent ID. Each node in database has its `parentId` and that is XID of parent node. Root element has parent ID set to -1 and then root node has XID -1. Root node is in fact represented by record in `XML_DOCUMENT` table, but there is no record in node tables, which represent root node of documents.

5.2.2 Indexing

XIDs and parent IDs are enough to construct entire XML tree and therefore that is enough to build XML document from relation data. But it has been mentioned in preceding chapter, that SQL is not able to evaluate recursive queries and these queries would be necessary for some queries, which are quite often in XPath language, e.g. all descendants. For this reason indexing structure has been added to the database architecture.

Regions

The main structure taken over from [23] is *region*. Each element in `XML_ELEMENT` table has its *region*, which is a pair of integer numbers, which bound entire subtree of the element. In this implementation the begin of a region is XID of an element node itself. The end of region is XID of a node, which is the last child of the element. With this structure queries, e.g. all-descendants (see Subsection 2.3.5) of an element, can be very easily evaluated by SQL — the result are nodes with XID between the start of region and the end of region. In the same way can be evaluated queries for all ancestors of node. The `XML_ELEMENT` table contains end ID column, which is end region. But it does not contain start region integer, because this is already element XID. Firebird engine does not support recursive SQL queries, so regions are only way, how to support all XPath queries, and this way is very powerful.

Disadvantage of region is necessity to recompute nearly all regions in database during inserting nodes into database or deleting nodes. This can be reduced by recomputing regions after all updates to a XML document. XSLT processor does not use inserting or deleting nodes from XML document. It appends new nodes to the end of output document. Implementation uses stack of nodes, which is represented by temporary table called `XML_ELEMENT_STACK`, and regions are directly computed during appending of nodes.

Simple path index

The second indexing structure is called *Simple path index*. For each element node and attribute node is stored a path expression, which is very similar to XPath location path expressions. There has been a mention in Subsection 2.3.3, that location paths with child axes with name tests are the most usual. Due to this, the simple path index contains all possible location paths, which consists of child axes and name tests. Each element and attribute stored in database then contain ID of corresponding path in simple path index. The simple path index is demonstrated on the XML example shown in Figure 2.2. The index is shown in Table 5.1.

XML node	Path index	Location path
element `book`	1	`/book`
element `title`	2	`/book/title`

Table 5.1: Simple path index example

Then evaluating of these paths is very simple by querying simple path index. This index can be also used for better evaluation of all paths relative to the context node. Each prefix of all location paths, which consists of child axes and name tests can be evaluated by simple path index.

The evaluation of these XPath expressions is done by SQL *like* operator. The document from example shown in Figure 2.2 would be a current document. Then the evaluation of example XPath expressions is shown in Figure 5.2

- **Expression:** `/book/title`

  ```
  select XID from XML\_PATH where XPATHEXP = '/book/title'
  ```

- **Expression:** `//title`

  ```
  select XID from XML\_PATH where XPATHEXP like 'title'
  ```

- **Expression:** `title`
  Context node: element `book`

  ```
  select XID from XML\_PATH where XPATHEXP like '%title'
          and PARENTID = 1
  ```

Figure 5.2: Example XPath evaluation with simple path index

Chapter 6

XSLT processing implementation

This chapter describes in details, how implementation of XSLT described in Section 2.4 is done in relational engine environment.

6.1 Implementation technologies

It has been discussed that there are mainly two approaches, how to implement XSLT evaluation in a relational engine. It can be done directly in relational engine, by SQL queries and by non-standard procedural extensions to SQL. The further method is to use external facilities like Java or .NET environment. This method is not very suitable for implementing XSLT processing algorithm, because all data, which should be processed by processor, has to go through the database driver and it means copying data between processes, a database server and a client application. However, this copying is not required, because the target of this XSLT implementation is to store the output document in the same relational database.

For this purpose the implementation by SQL and its procedural extensions has been chosen and the Section 6.6 describes structure of procedures and how it processes a stylesheet.

6.2 XPath evaluation

XPath has been developed together with XSLT and is very important part of XSLT, because it is used for navigating in source XML document, for selecting parts of this document and for evaluating values for output document. Implementation of XPath is principal for XSLT processor implementation. XPath is declarative query language and in this basic characteristic it is very similar to SQL.

6.2.1 Query conversion into SQL

In essence there are two approaches, how to evaluate XPath queries against XML data stored in relational database. Former is to parse XPath query and emulate going through XML document tree like it is described by XPath location paths. This emulation can be implemented on the client side of relational database engine, e.g. in Java, or, for better performance, in procedural extensions of SQL. The latter approach to evaluate XPath is to convert XPath statement into SQL select statement, which selects the same rows in relational database like XPath in source XML document.

The former one approach can be quite easily implemented and can implement all complicated features of XPath language, like numbering, context sensitiveness of XPath language, when it is used in XSLT. Successfulness of implementation of XPath to SQL conversion is not so obvious. Although SQL is declarative query language like XPath, standard SQL does not have some features, which are essential in XPath, e.g. row numbering. This problem can be solved by new features of SQL:2003 standard called window functions, e.g. `ROW_NUMBER` function. This function is able to compute sequence number for each row, or for group of rows. Unfortunately this feature is not implemented in open source relational engines and chosen Firebird engine does not support this feature at all. But row numbering is very important for evaluating XPath expressions, where XPath position function is used. In this work, the row numbering has been evaluated by set of stored procedures and a table for handling current index for groups. For ordering is used standard SQL clause `ORDER BY`. Window functions are able to filter output directly in SELECT statement by usage of QUALIFY clause. In the following example is shown SQL:2003 statement.

The statement shown in Figure 6.1 selects third row from each group of rows from the `XML_ELEMENT` table. Rows are sorted by `XID` column ascending

```
SELECT
  *
FROM
  XML_ELEMENT
QUALIFY
 ROW_NUMBER() OVER (PARTITION BY PARENTID
  ORDER BY XID ASC) = 3
```

Figure 6.1: Window function example

and grouped by PARENTID column. Then from each group, the third row is selected. In used XML mapping into relations, XID represents unique identifier, which is assigned in document order. A PARENTID is the XID of the parent element.

In this thesis a way, how to substitute this feature, has been developed. Firebird supports stored procedure, which look like normal relations. These procedures are called select procedure and can be used in select statements anywhere the normal tables or views can be. Procedural extension of SQL in Firebird does not offer any advanced way, how to store temporary values. So the normal table is used for storing current value of row counting during select statement evaluation. Example of usage of this scenario follows.

The example in Figure 6.2 is rewritten example in Figure 6.1. Selecting of elements is done in inner select. Outer select is used because of ordering of inner select. Row numbering cannot be done in inner select, because rows for NODE_COUNTER procedure are served in random ordering and the result is ordered. But this cannot be used. So outer select is used with NODE_COUNTER select procedure and row from inner select are returned in right order, they get row number from NODE_COUNTER procedure and result are filtered by standard SQL clause where. CLEAR_COUNTER procedure, which is the first table in FROM clause of outer select, is used for automatic deleting of temporary table.

It is obvious that this solution has to be less effective than ROW_NUMBER function, because optimizer cannot presume anything of a NODE_COUNTER procedure results. On the other hand, ROW_COUNT function is plain increasing function and equal result suggest that the following rows from the same group will not be in result set. Usage of this scenario is not so clear too. But it can be considered as a proof of concept, that the

```
select
  s.eXID as XID
from
  CLEAR_COUNTER, (
      select
        e.PARENTID as ePARENTID
        ,e.DOCID as eDOCID
        ,e.PATHID as ePATHID
        ,e.CONTENT as eCONTENT
        ,e.CHUNK as eCHUNK
        ,e.ENDID as eENDID
        ,e.NAMESPACEID as eNAMESPACEID
        ,e.XID as eXID
        ,e.NAMEID as eNAMEID
      from
        XML_ELEMENT as e
      order by e.XID ASC
    ) as s
    left outer join NODE_COUNTER(2, s.ePARENTID, s.eXID) as r
      on 1 = 1
where
  r.ONODEINDEX = 3
order by s.eXID ASC
```

Figure 6.2: Implementation of window function

lack of SQL:2003 features can be substituted by stored procedures in this relational engine.

6.2.2 Convertor implementation in Java

In this XSLT implementation the conversion into SQL has been chosen, because this approach should be much more effective than to interpret XPath paths. Conversion to SQL is quite complex process, so conversion is implemented in Java and module of implementation is called *XPathConvertor*. At the beginning of conversion process, parsing of XPath expressions has to be done. However, a parser implementation is not a part of this thesis, so open source implementation called Jaxen ([12]) is used for parsing. This Java implementation parses expression and returns XPath object model. Object model is quite similar to constructs defined in XPath grammar. Then the conversion of these objects into SQL can be done.

It is necessary to use any object representation of SQL select statement to be able to dynamically change the result select statement. Unfortunately there is no implementation of this model, so own simple implementation has been created. This object model represents a select statements with all necessary features to be able to represent queries like previous examples. It means especially inner selects, select procedure, all kinds of joins, ordering, etc. The UML class diagram of this model can be found in Appendix A.

Implementation has been written by hand. So far no generation has been used to convert UML class diagram into Java classes. XPathConvertor uses this object model to create select statement from XPath expression object model. Result of this process is object model of select statement. Into select object model has been implemented features to convert this model into pretty written string representation.

6.2.3 Convertor integration into XSLT implementation

The previous sections described XPathConvertor, which has been implemented in Java language, and other section mentioned, that it has been decided to implement XSLT processing algorithm directly by relational engine procedural extensions to SQL. These two worlds has to be integrated together to create functioning XSLT engine. Firebird database engine has only one possible option, how to call external routines from SQL and procedures. It is called user defined functions (UDF), but so far there is only support for binary routines with C-style calling interface. In a new version of Firebird there will be already support for Java functions called by SQL statements.

So far this function is not ready, thus XSLT implementation uses compiling of XSLT stylesheets. XSLT processor is controlled by Java application and this application before starting XSLT processing algorithm uses compiling of XSLT stylesheet. It means that all known attributes containing XPath are retrieved from databases and XPath2SQL convertor creates SQL queries. These queries are then saved to database to XPATH_COMPILATION and XPATH_COMPILATION_TEXT tables

These queries usually depend on parameters or context, which are not

```
select
  s.eXID as XID
from
  CLEAR_COUNTER, (
      select
        e.PARENTID as ePARENTID
        ,e.DOCID as eDOCID
        ,e.PATHID as ePATHID
        ,e.CONTENT as eCONTENT
        ,e.CHUNK as eCHUNK
        ,e.ENDID as eENDID
        ,e.NAMESPACEID as eNAMESPACEID
        ,e.XID as eXID
        ,e.NAMEID as eNAMEID
      from
        XML_ELEMENT as e
      order by e.XID ASC
    ) as s
    left outer join NODE_COUNTER(2, s.ePARENTID, s.eXID) as r
      on 1 = 1
where
  r.ONODEINDEX = ${PARAMETER_INDEX}
order by s.eXID ASC
```

Figure 6.3: Parameter usage in SQL

known during compiling process. This implementation does not support complex parameters, e.g. entire XPath paths, because for parameter support it uses string replace routines. It means that entire logic of XPath query has to be known during compilation. Parameters are replaced by shortcuts and these shortcuts are replace by valid values during XSLT processing. The same way the context sensitiveness is supported in these queries.

The example in Figure 6.3 shows usage of parameters in generated SQL. The query returns ordered list of element nodes identified by their XIDs. But this query is parametrized by PARAMETER_INDEX, which filters list to only one element node with specified position in document order.

6.3 Algorithm of XSLT processing

Algorithm is implemented in Firebird procedural extensions to standard SQL. Before starting of this algorithm, that is necessary to convert all XPath queries to standard SQL. This queries are stored in compilation tables in text format. It is standard SQL query, but it contains parameters and context variables in text format. These parameters are replaced by valid values during process.

XSLT algorithm has been described in Section 2.4. XSLT processor reads root node element and looks for template. If no template is found, then the built in templates are used. Templates are searched by their patterns and the pattern matching implementation is described in the next section.

6.4 Pattern matching implementation

XSLT template can have pattern attribute, which content is a subset of XPath language. This XPath expression can be evaluated into the list of nodes and these nodes can be processed by this template.

In this implementation, pattern matching is done directly by this algorithm. For each pattern the SQL select is prepared and stored in compilation tables. Before actual XSLT stylesheet processing, XSLT procedure evaluates their pattern selects and results, exactly set of nodes, are stored in table called XSLT_TEMPLATE_NODE, which has a primary key combined of XSLT stylesheet document ID, stylesheet document ID and template element XID.

During XSLT processing processor looks for correct template for an node, which is identified by its XID. So valid template can be selected from XSLT_TEMPLATE_NODE table by filtering XID of examined node, stylesheet id and source document id. It is possible, that multiple templates are selected. Then the priority attribute of templates has to be tested or default priority for each template has to be computed. This is not so far implemented and only one template is chosen randomly and interpreted.

This solution of pattern matching in relational databases can be quite exhausting for table space resources for very large tables. But it is very powerful for fast evaluation of pattern matching, because pattern matching is done by one SQL query based on indexed table.

6.5 Stylesheet programming styles

Basically XSLT provides possibilities, how to write same functions by different XSLT instructions. This section discuss two different ways, how to write XSLT stylesheets. These two approaches are usually called push and pull model.

6.5.1 Push model of stylesheets

This style of XSLT programming is based on many small templates in stylesheet. It is very usual, that each template has a pattern for some sort of source nodes, generate some output and contains one or more apply templates instruction. This style is very flexible, because templates can be overridden by other templates with other *import precedence*. Therefore users can rewrite parts of XSLT stylesheet with preserved main functionality. This possibility has be described in Section 2.4.1. Data from source document are firstly read by XSLT processor and are "pushed" to the templates.

6.5.2 Pull model of stylesheets

Pull model is very different. Stylesheet contains very small number of templates, even more it can contain only one template for root node or root element of XML document. All logic is concentrated to this template. Data from source XML document are "pulled" from source document by value-of instructions and for-each instruction is used for iterating over source document. In these XSLT stylesheets the apply-templates instructions is rarely used.

6.5.3 Different performance for different models

Push and pull model is very differently interpreted by XSLT implementation. For push model the patterns are evaluated before real XSLT processing and are evaluated only once. The value-of instructions are used with very simple queries or they are not used at all, because for getting data from source document to the output the built-in templates are used.

Pull model does not use XSLT_TEMPLATE_NODE table, so it saves table space resources. On the other hand, value-of instructions are used very often and usually contain complicated XPath expression. These expressions has to be evaluated for each value-of evaluation and it can be quite time-consuming.

6.6 XSLT stored procedures

XSLT processing are implemented by stored procedures of Firebird relational engine and are written in SQL with procedural extensions called PSQL. In fact entire processor is implemented in four procedures: `XSLT_PROCESS`, `XSLT_EXECUTE_TEMPLATE`, `XSLT_APPLY_TEMPLATES` and `XSLT_VALUEOF`. Calling graph of these procedure is shown here Figure 6.4.

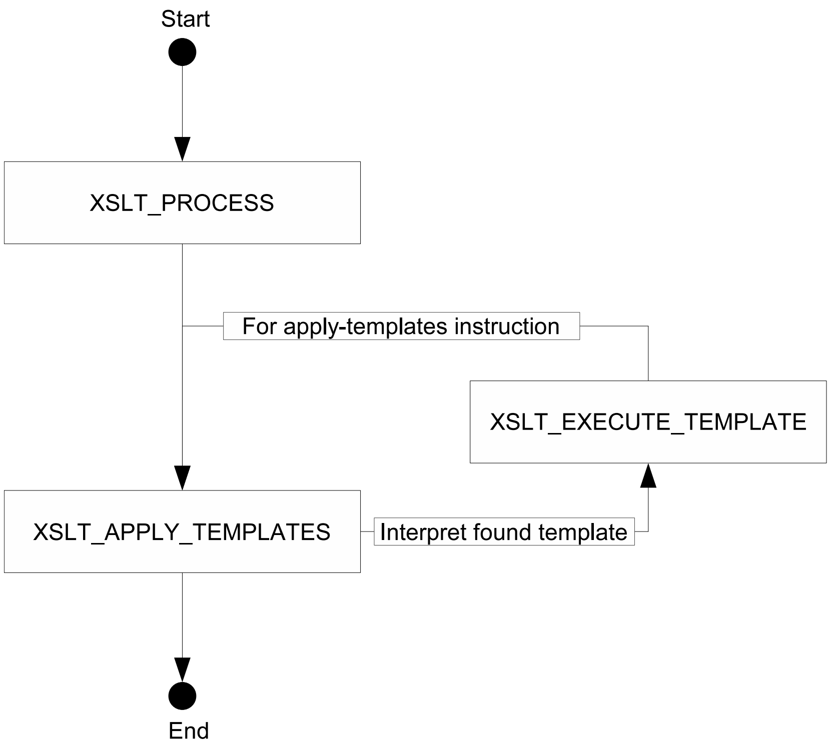


Figure 6.4: Stored procedures calling graph

6.6.1 Procedure XSLT_PROCESS

`XSLT_PROCESS` is the main procedure and an entry point to the XSLT functionalities. This procedure has this header:

```
XSLT_PROCESS(XSLTDOCID, SRCDOCID) RETURNING DSTDOCID
```

Calling of this procedure launches XSLT processing of the source document with document id `SRCDOCID`, stylesheet stored as the document with the document id `XSLTDOCID`. This procedure returns only one value, `DSTDOCID`, which is a document id of output document. This procedure is called by Java control part of XSLT implementation. It prepares output document ID and inserts new record with this ID into `XML_DOCUMENT` table. The name of output document is always set to `'XSLToutput'` constant.

This procedure is responsible for reading stylesheet element of stylesheet document. For each template element searches their match attributes and then looks for select statement, which has been prepared by XPath convertor into `XPATH_COMPILATION` and `XPATH_COMPILATION_TEXT` tables. These select statements are executed by EXECUTE STATEMENT statement, which is able to execute statements stored in a CHAR variable. Results are stored in `XSLT_TEMPLATE_NODE` table.

When patterns of all templates are evaluated, the procedure looks for root node template. XPath convertor is not used for root node patterns and root node template has to equal to `'/'` string. When the root node template is found, directly `XSLT_EXECUTE_TEMPLATE` is called for root node template. Otherwise the `XSLT_APPLY_TEMPLATES` is called for all root elements of source XML document.

6.6.2 Procedure XSLT_EXECUTE_TEMPLATE

This procedure interprets content of a template element of stylesheet. The processing can be divided into two parts: XSLT content and non-XSLT content. XSLT content means XSLT elements and attributes, which semantics has to be evaluated. Non-XSLT content is directly written to the output document. Non-XSLT content is also XML and the closing element tags has to be generated. It means that end XIDs has to be properly computed for elements in output document. For this purpose a stack of opened XML elements has to be kept during processing template. Firebird engine does not have a data structure like stack, which can be accessed by stored procedures. So the stack is simulated by table called `XML_ELEMENT_STACK` and by three stored procedures: `XML_STACK_POP`, `XML_STACK_PEEK` and `XML_STACK_PUSH`. This table holds opened elements (and their end `XID` for elements from read XSLT document) during XSLT processing. This procedure interprets certain template and after the processing the stack is leaved in the same state like when the procedure started. This procedure also evaluated XSLT element instruction and calls `XSLT_VALUEOF` procedure for XSLT value-of instructions and writes the output of `XSLT_VALUEOF` procedure to the output document.

6.6.3 Procedure XSLT_APPLY_TEMPLATES

This procedure is called for each XSLT apply-templates instruction inside stylesheet. It iterates over all children of context node, which is passed in parameter. It has to be mentioned, that children in the previous sentence are meant in XSLT style. Children of an node for apply-templates instruction means only child element nodes and child text nodes. For each child node the proper templates is searched in `XSLT_TEMPLATE_NODE` table. The priority of templates is not so far supported. If some template is found, the `XSLT_EXECUTE_TEMPLATE` is called. Otherwise built-in templates are used. It means for element nodes to recursively call apply-template instruction. Text nodes are in built-in template directly generated to the output document.

6.6.4 Procedure XSLT_VALUEOF

XSLT value-of instructions are evaluated in this procedure. Select attribute is loaded from database and its XPath value is searched in `XPATH_COMPILATION` table and then the text of select statement for this XPath expression is loaded from `XPATH_COMPILATION_TEXT` table. Context parameters are inserted into this statement by string replace function. Then this statement is evaluated by `EXECUTE STATEMENT` and for each node in result set the text value is computed. It means that for each node all descendant text nodes and all attribute values are retrieved and written to the output of this procedure. This procedure does not write directly into the output document. However, it is select procedure, which returns set of text values. These values are then written to the output document, e.g. by `XSLT_EXECUTE_TEMPLATE` procedure.

Chapter 7

Limitations

The target of this thesis is to prove, that implementation of XSLT with relational engine resources is possible and is quite efficient. It is not a target of this thesis to completely implement XSLT standard. Some limitations of this implementation are covered in the further sections.

7.1 XML database limitations

XML database implements storage for any kind of XML document. The support for missing features like storing comments and processing instructions can be easily implemented, but it is not interesting for targets of this thesis. There are not any other limitations in XML storage implementation.

7.2 XPath convertor limitations

XPath convertor to SQL has been designed to be able to properly evaluate the standard XPath expressions like relative and absolute paths. It also supports document ordering and expressions with predicates, which selects by position in document. This position cannot be prepared before querying, because the position is computed inside selected context nodes and this context is not known before processing. The architecture of the XPath convertor supports this node numbering.

Other predicates can be very easily supported by recursively adding generated selects for predicates into the outer select generated for outer expression. The joining of these selects is done in SQL select object model by proxy column objects for all possible context values.

The main limitation of XPath convertor is impossibility to easily evaluate the `last` XPath context function (description of this function can be found in Subsection 2.3.8), which returns index of the last node in current context. This is only possible by SQL count function, but this function can be used only in a separated select statement with `group by` clause. This is not supported because the select statements would become much more complicated and unreadable. This problem can be very easily solved by SQL:2003 window functions.

7.3 XSLT limitations

It has been implemented essential features of XSLT standard. This XSLT implementation is able to evaluate apply-templates, value-of and element instructions. Template patterns and select attributes of value-of instruction are limitations are derived from XPath convertor limitations. All implemented features are presented in examples shown in Appendix C.

The implementation does not evaluate the whitespace stripping as it is described in XSLT standard ([25]). It generates all found text nodes to the output document. It has been considered for the testing purpose, that two XML documents are not different, when they differ only in text nodes of whitespaces. This is important to be able to compare this implementation with other third party implementation.

Chapter 8

Implementation usage

In this chapter instructions for usage of this XSLT implementation will be described. It means installation instructions to establish entire environment and a brief overview of user interface of this XSLT implementation.

8.1 Installation

Installation process consists of installation Firebird database engine and Java environment.

Firebird engine and Java installation process instructions can be found on the websites of these projects ([8] and [11]). And the following text assumes that these parts are properly installed. However, installation files are included on the enclosed CD (its content is described in Appendix E).

Firebird engine database alias has to be configured for XSLT processor. This alias is called `xmldb` and has to point to the database file `xml.fdb`. The database alias configuration file can be found on the CD and this configuration file has to be updated according to the location of the database file. Then this alias file has to be copied into the Firebird installation directory.

Next, the rFunc UDF library ([15]) has to be installed. The implementation uses `strreplace` function for parametrizing SQL queries. The installation contains `rfunc.dll` for Windows operating system and `rfunc.so` for Linux. This dynamic library has to be copied into `%FIREBIRD_HOME%\UDF` directory.

Entire Java implementation is packed into the one archive called `xsltproc-1.0.0.zip` with all Java dependencies (it does not include Java runtime). All the files from this archive has to be extracted to a directory.

8.2 Executing implementation

It has been mentioned, that the Firebird database engine (see Section 3.1) supports several different connection architectures. Because of the performance issues, the default implementation setting uses *LOCAL* connections, which does not uses network sockets for communication and uses binary driver. For this purpose the necessary dynamic library has to be supplied to the implementation process. It has to be done by Java system property as shown in Figure 8.1. The `%JAYBIRD_HOME%` has to contain library `jaybird21.dll` for Windows operating system and `libjaybird21.so` for Linux.

If the binary driver cannot be used, the connection parameters of implementation can be reconfigured by Java system properties. All possible properties are shown in Table 8.1.

```
java -Djava.library.path=%JAYBIRD_HOME%
  -jar xsltproc-0.0.1.jar
  ...
```

Figure 8.1: Command line for executing implementation

8.3 User interface

Implementation can be very easily controlled by any Java program by using its `XSLTProcessor` class, which is an entry point into the implementation. For standard usage implementation has a user interface in a console program style.

When the Java archive is started without any argument, the console of the XSLT processor appears. It supports following commands, which can be connected together by semicolon for batch processing:

- **help** – prints list of all possible commands
- **quit** or **exit** – stops XSLT processor
- **delete(filename)** – deletes document selected by the filename from database

- **deleteId(documentId)** – deletes document selected by its document id from database
- **clear** – deletes all data from database
- **import(filename)** – imports file from file system into database; generated new document id for this file is printed to the console (if document with the same name already exists in database, this document is deleted before import)
- **export(filename,destinationFilename)** – exports document from database selected by its filename into the destination filename (destination filename is overwritten, if already exists)
- **exportId(documentId,destinationFilename** – exports document selected by its document id into the destination filename
- **process(sourceDocumentId,xsltDocumentId** – executes XSLT stylesheet selected by document id on the source document selected by document id; new generated document id for output is printed to the console and output has `XSLToutput` filename always
- **xslt(sourceFilename,xsltFilename,destinationFilename)** – standard XSLT usage command – imports source and stylesheet documents into the database and processes stylesheet on the source document; output document is exported into the destination filename; all three documents are leaved in the database

The XSLT processor can be also controlled by command line arguments. These arguments are the command, which are the same previously mentioned. Each argument is considered to be a valid statement. An example of running XSLT processor by command line arguments is shown in Figure 8.2.

```
java -jar xsltproc-0.0.1.jar import(.\\samples\\catalog.xml)
```

Figure 8.2: Command line arguments example

`jdbc.driverClassName`	The class name of the Jaybird driver. It should not be changed and the standard value is `org.firebirdsql.jdbc.FBDriver`.
`jdbc.username`	The database username.
`jdbc.password`	The plain text password for specified username.
`jdbc.rolename`	The role name used for connection. In the supplied database, the role called `XML_ADMIN` has all privileges to use the database.
`firebird.database`	The **only** database name or alias name for `LOCAL` communication style. Its default value is `xmldb`. However, it can be configured to a filename in the system, if aliases are not properly configured. For the network communication the entire network path has to be supplied (the last part can be alias): `//host:port/path/database.fdb`, e.g. `//localhost:3050/xmldb`
`firebird.connection.type`	This is the connection type of the Firebird relational engine. The default value is `LOCAL` and binary driver library has to be used. It can be changed to `TYPE4` and then the network sockets are used for communication.

Table 8.1: Implementation connection properties

Chapter 9

Experiments

This implementation has been intended to be able to process large XML documents without large resource requirements. The tests are summarized in this chapter and some possible architecture changes to improve performance of the implementation are designed.

9.1 Functionality experiment

The development of the implementation has been done with continuous testing of XSLT processing. For comparing, the thirdparty XSLT implementation has been used called Saxon ([17]). Open source version of this implementation has been used. It is widely used conformant implementation of XSLT Version 1.0.

All results generated by this implementation has been compared to the results generated by Saxon. The small example is shown in Figure 9.1. As mentioned in Chapter 7, the implementation does not do whitespace stripping and so the implementation slightly differs. In essence, this could be a problem for text oriented XML documents (see Section 2.1.2). The implementation does not support XML processing instructions, so it is not generated to the output document.

9.2 Performance and possible optimizations

The implementation has been tested on a generated XML documents. These source XML documents have been created by ToXgene (see Section 3.6). The

Saxon result (the text is wrapped because of document, however, Saxon generates the entire document on a one line, because all other whitespaces are stripped by default)

```
<?xml version="1.0" encoding="utf-8"?>
<library><member>8907319714</member><member>5610617979</member>
...<member>2897127464</member><member>2034052068</member>
</library>
```

Implementation result

```

<library>
  <member>8907319714</member>
<member>5610617979</member>
...
<member>2897127464</member>
<member>2034052068</member>

</library>
```

Figure 9.1: Comparing to other implementation

one of them has about 10 000 `book` elements. The structure of the generated document and used stylesheet can be found in Appendix D. The parameters of the test are shown in Table 9.1.

The problems with performance appeared firstly during importing phase. The communication with the server over the network sockets appeared to be a bottleneck for *importing* of the XML documents into the database. This problem has been solved by the Firebird database engine architecture possibilities. The *LOCAL* communication style has been used and the performance of the importing phase rapidly changed. The duration decreased almost twenty times. The problem with network is probably hidden in TCP/IP protocol, because neither Firebird server nor client Java application dit not used CPU to full capacity.

The performance of the *XSLT processing* phase was examined by IBExpert application (see Section 3.2). It is able to show total num-

Count of books in the source document	10 000
Hardware configuration	Laptop Pentium M 1.6 GHz
Number of elements of the source document	c. 100 000
Total count of nodes of the source document	c. 200 000
Import phase	c. 1 minute
XSLT processing phase	c. 5 minute
Export phase	c. 15 seconds

Table 9.1: Parameters of the performance test

ber of the readings and writings in the database engine. However, it is not able to show the worst queries. The debugger of this application has been very useful for development of the XSLT processing procedures.

It has been discovered, that the XSLT implementation has no memory requirements for XSLT processing, which would be dependent on the size of the source documents. However, the current implementation of the processing procedures depends on the stack simulated by a table. This is probably performance issue, which can be solved by recursive procedure calling. These two concurrent approaches has not been compared in this thesis.

9.3 Possible architecture changes

The implementation uses generic mapping for XML storage and so it cannot be easily used for processing existing relational data. It could be useful to update implementation to use schema driven mapping, which could be easily mapped on the existing data to transform relational data by XSLT. However, the generic mapping has an advantage, that all kinds of XML files can be processed, even with mixed content. Integration with existing relational schema could be done by implement conversion from an relational schema into generic mapping used by this implementation.

The used XPath to SQL conversion cannot use advanced SQL features. However, these constructs are usually common in commercial database engines and generated SQL queries would be quite simpler.

Chapter 10

Conclusion

The targets has been described in the Section 1.1. This XSLT implementation uses relational databases as a temporary storage and uses SQL and its procedural extensions to process entire XSLT algorithm directly inside relational engine. XPath convertor demonstrates, that it is possible to convert entire XPath semantics into the one SQL select statement. It can be very powerful for evaluating complicated queries against large XML documents.

Many of the problems of the conversion XPath to SQL, e.g. row numbering, has been solved and implemented in this thesis. The XPath2SQL convertor architecture is ready to be able to implement almost entire XPath language. But some approaches used to be able to implement some features would be a performance problem.

This thesis also served as a opportunity to build fully featured development environment for Java based database applications. Many tools, e.g. Maven, SVN, has been installed and configured and used during entire development process of the implementation, despite it was not necessary for the thesis. However, it brought some unique possibilities, e.g. to be able to share the XSLT implementation with Maven community very easily.

The all targets of this thesis are in principals implemented and the implementation can be considered as a base for further implementations. Despite quite small resource requirements there are some optimization issues described in Chapter 9. However, these optimizations are not a part of this proof of concept implementation.

The proof, that XSLT can be evaluated by procedural extensions of SQL can be considered as a biggest contribution of this thesis, together with XPath to SQL conversion.

Bibliography

[1] *(*ArgoUML), Tigris.org, 2007,
available at http://argouml.tigris.org/ 47

[2] *Apache Maven Project*, The Apache Software Foundation, 2007,
available at http://maven.apache.org/ 48

[3] James Clark, *Associating Style Sheets with XML documents Version 1.0*, W3C, 1999,
available at http://www.w3.org/TR/xml-stylesheet 30

[4] Jaroslav Pokorný, *Dotazovací jazyky*, Karolinum, 2002 42, 55

[5] *Eclipse SDK 3.2.2*, Eclipse contributors and others, 2007,
available at http://www.eclipse.org/ 47

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau, *Extensible Markup Language (XML) 1.1 (Second Edition)*, W3C, 2006,
available at http://www.w3.org/TR/2006/REC-xml11-20060816/. 14

[7] *Firebird 2.0 Online Manual*, Janus Software, 2007,
available at http://www.janus-software.com/fbmanual/ 44

[8] *Firebird 2.0.1*, Firebird Foundation Incorporated, 2007,
available at http://www.firebirdsql.org/ 42, 76

[9] *Firebird Documentation Set*, IBPhoenix, 2007,
available at http://www.firebirdsql.org/manual/. 43

[10] *IBExpert 2007.07.04 Personal Edition*, HK-Software, 2007,
available at http://www.ibexpert.com/ 44

[11] *Java Runtime Environment 6 Update 2*, Sun Microsystems, Inc., 2007, available at http://java.sun.com/javase/ 45, 76

[12] *Jaxen 1.1*, Codehaus, 2007, available at http://jaxen.org/ 66

[13] *Jaybird 2.2*, 2006, available at http://jaybirdwiki.firebirdsql.org/ 45

[14] Tim Bray, Dave Hollander, Andrew Layman and Richard Tobin, *Namespaces in XML 1.0 (Second Edition)*, W3C, 2006, available at http://www.w3.org/TR/REC-xml-names/ 17, 18, 36

[15] *rFunc UDF Library*, Polaris Software, 2003, available at http://rfunc.sourceforge.net/ 76

[16] David Megginson, *SAX*, SourceForge, 2000, avilable at http://www.saxproject.org/ 12, 46

[17] Michael H. Kay, *Saxon 6.5.3*, SourceForge, 2007, available at http://saxon.sourceforge.net/ 11, 80

[18] *Spring Application Framework 2.0.2*, Interface21 and community, 2007, available at http://www.springframework.org/ 46, 47

[19] *Subclipse 1.0.5*, Tigris.org, available at http://subclipse.tigris.org/ 48

[20] *Subversion 1.3.2*, Tigris.org, available at http://subversion.tigris.org/ 48

[21] Denilson Barbosa, *ToXgene 2.3*, the University of Toronto, the IBM Toronto Lab, available at http://www.cs.toronto.edu/tox/toxgene/ 48

[22] *Xerces2 Java Parser*, The Apache Software Foundation, 2006, available at http://xerces.apache.org/xerces2-j/ 11, 12, 46

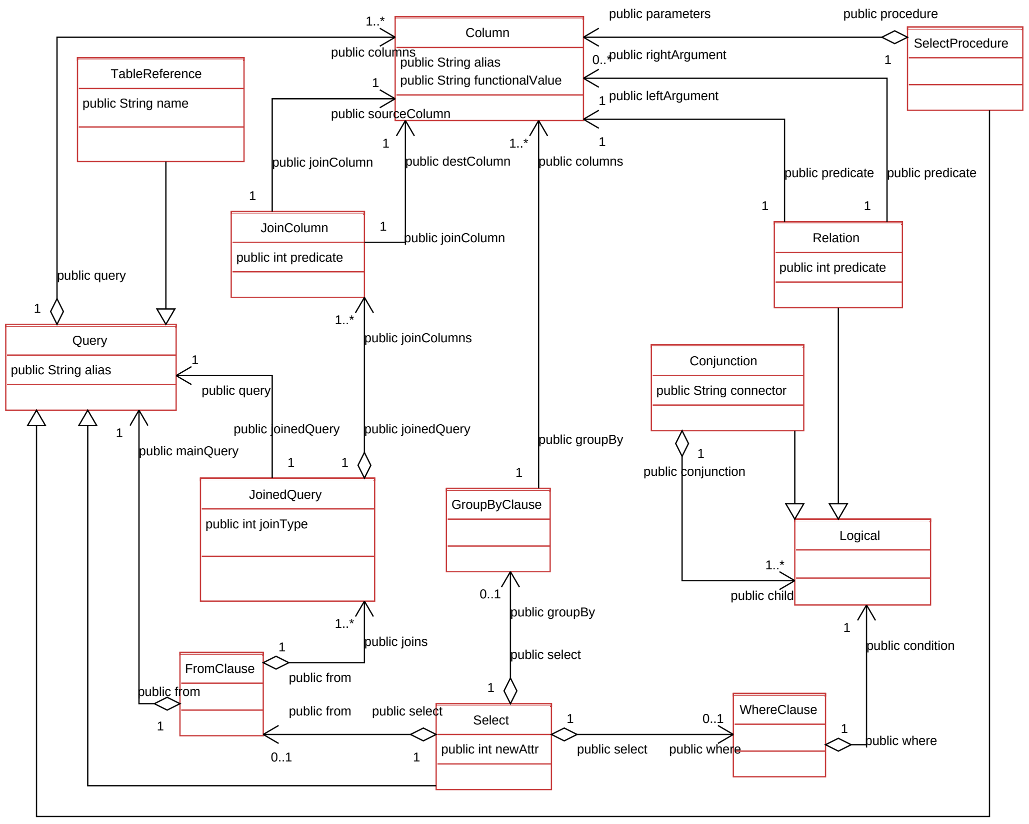
[23] Irena Mlýnková and Jaroslav Pokorný, *XML in the world of (Object-) Relational Database Systems*, Universitas Carolina Pragensis, 2003. 12, 54, 55, 56, 60

[24] James Clark and Steve DeRose, *XML Path Language (XPath) Version 1.0*, W3C, 1999,
available at http://www.w3.org/TR/xpath. 19, 20, 24, 28

[25] James Clark, *XSL Transformations (XSLT) Version 1.0*, W3C, 1999,
available at http://www.w3.org/TR/xslt. 31, 33, 38, 40, 41, 75

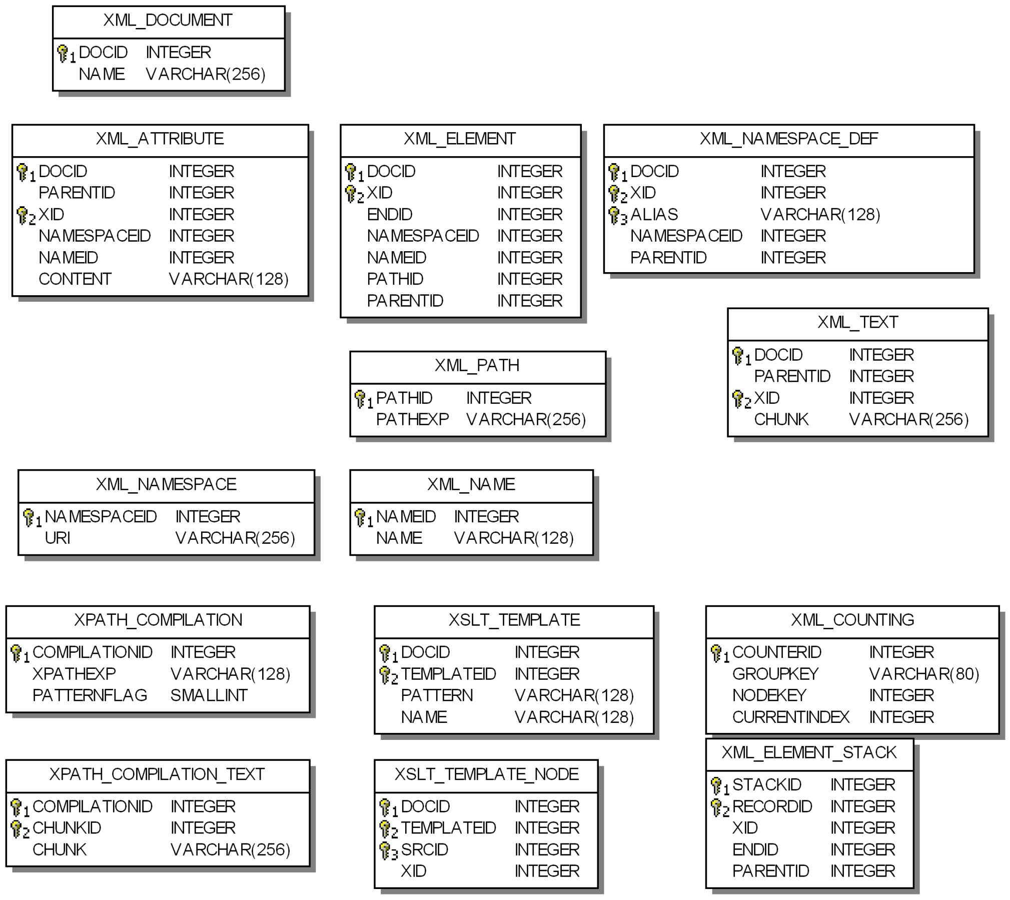
Appendix A

UML model for SQL select statement



Appendix B

XML database structure



XML_DOCUMENT		
🔑1	DOCID	INTEGER
	NAME	VARCHAR(256)

XML_ATTRIBUTE		
🔑1	DOCID	INTEGER
	PARENTID	INTEGER
🔑2	XID	INTEGER
	NAMESPACEID	INTEGER
	NAMEID	INTEGER
	CONTENT	VARCHAR(128)

XML_ELEMENT		
🔑1	DOCID	INTEGER
🔑2	XID	INTEGER
	ENDID	INTEGER
	NAMESPACEID	INTEGER
	NAMEID	INTEGER
	PATHID	INTEGER
	PARENTID	INTEGER

XML_NAMESPACE_DEF		
🔑1	DOCID	INTEGER
🔑2	XID	INTEGER
🔑3	ALIAS	VARCHAR(128)
	NAMESPACEID	INTEGER
	PARENTID	INTEGER

XML_TEXT		
🔑1	DOCID	INTEGER
	PARENTID	INTEGER
🔑2	XID	INTEGER
	CHUNK	VARCHAR(256)

XML_PATH		
🔑1	PATHID	INTEGER
	PATHEXP	VARCHAR(256)

XML_NAMESPACE		
🔑1	NAMESPACEID	INTEGER
	URI	VARCHAR(256)

XML_NAME		
🔑1	NAMEID	INTEGER
	NAME	VARCHAR(128)

XPATH_COMPILATION		
🔑1	COMPILATIONID	INTEGER
	XPATHEXP	VARCHAR(128)
	PATTERNFLAG	SMALLINT

XSLT_TEMPLATE		
🔑1	DOCID	INTEGER
🔑2	TEMPLATEID	INTEGER
	PATTERN	VARCHAR(128)
	NAME	VARCHAR(128)

XML_COUNTING		
🔑1	COUNTERID	INTEGER
	GROUPKEY	VARCHAR(80)
	NODEKEY	INTEGER
	CURRENTINDEX	INTEGER

XPATH_COMPILATION_TEXT		
🔑1	COMPILATIONID	INTEGER
🔑2	CHUNKID	INTEGER
	CHUNK	VARCHAR(256)

XSLT_TEMPLATE_NODE		
🔑1	DOCID	INTEGER
🔑2	TEMPLATEID	INTEGER
🔑3	SRCID	INTEGER
	XID	INTEGER

XML_ELEMENT_STACK		
🔑1	STACKID	INTEGER
🔑2	RECORDID	INTEGER
	XID	INTEGER
	ENDID	INTEGER
	PARENTID	INTEGER

Appendix C

Example

C.1 Source XML document

```
<?xml version="1.0" encoding="windows-1250"?>
<x:book style="textbook"
  xmlns:x='http://p3500.kolej.mff.cuni.cz/xslt'>
  <x:title>Designing XML applications</x:title>
  <x:author>
    <x:family>Nick</x:family><x:given>Marcus</x:given>
    <x:family>Bob</x:family><x:given>Pant</x:given>
  </x:author>
  <x:test>ssss</x:test>
</x:book>
```

C.2 Stylesheet document

```
<?xml version="1.0" encoding="windows-1250"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:x='http://p3500.kolej.mff.cuni.cz/xslt' version="1.0">

  <xsl:template match="/"><xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="x:book">
```

```
<chapter>
<style><xsl:value-of select="@style"/></style>
  <xsl:apply-templates />
</chapter>
  </xsl:template>

  <xsl:template match="x:author">
<test xmlns="test.test" test="test" test2="test">
</test>
      </xsl:template>

  <xsl:template match="x:title">
<title>
    <xsl:apply-templates/>
</title>
</xsl:template>

<xsl:template match="x:test">
<example>
</example>
</xsl:stylesheet>
```

Appendix D

Large data example

D.1 ToXgene generated sample document

```
<?xml version="1.0" encoding="US-ASCII"?>
<!-- generated by ToXgene Version 2.3
  on Wed Aug 01 00:27:42 CEST 2007 -->
<catalog>
  <book isbn="0957220188" genres="Mystery,Suspense">
    <title>
      sentiments dazzle final epitaphs;even,
      daring realms engage?pearls engage during
      the stealthly stealthy sheave
    </title>
    <author>Lem Martella</author>
    <author>Wessel Bladen</author>
    <author>Sujatha Nyrup</author>
    <author>Jiandong Ziavras</author>
    <author>Qinglan Mattoso</author>
    <author>Rosemarie Priestley</author>
    <author>Clarence Hansdah</author>
    <price currency="CDN">63,90</price>
  </book>
  <book isbn="7924503433" genres="Children">
    <title>
      permanently sly theodolites solve attainments?silent,
      blithe attainments beyond the ruthless, quick pains
      will have to maintain quickly
    </title>
    <author>Herkimer Messner</author>
    <author>Vidyadhar Port</author>
    <author>Zhongyang Dymetman</author>
    <author>Jarno Morihara</author>
    <author>Chaitali Henriksson</author>
    <author>Gregor Hemerik</author>
    <author>Roxanne Vedrine</author>
    <author>Russel Hanakawa</author>
    <price currency="CDN">113,00</price>
  </book>
  ...
</catalog>
```

D.2 Stylesheet document

```
<?xml version="1.0" encoding="windows-1250"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="catalog">
<library>
  <xsl:apply-templates />
</library>
</xsl:template>

<xsl:template match="book">
  <member><xsl:value-of select="@isbn" /></member>
</xsl:template>

</xsl:stylesheet>
```

Appendix E

CD-ROM Content

Entire implementation, all necessary libraries and examples are supplied on the enclosed CD-ROM. The structure of directories and the list of files follow:

cdrom.txt Contains this description.

bin/ Contains binary distribution of the implementation (without database).

database/xml.fdb Firebird database file.

document/thesis.pdf The thesis PDF document.

document/sources/ Contains all LaTeX sources and images of thesis PDF document.

install/ Contains all necessary installation files for development and running too

sources/ Contains SVN archive and Eclipse workspace with all sources of implementation. It also contains database DDL script for recreating database.