Mapping SPARQL and SQL to XQuery

Martin Kaufmann

Master’s Thesis

Systems Group
Department of Computer Science
ETH Zürich

http://www.systems.ethz.ch

15th September 2009 – 15th March 2010

Supervised by:
Prof. Dr. Donald Kossmann
Abstract

For developing an application which deals with a large amount of data, the state-of-the-art is to use a specialized query language like SQL for relational data or SPARQL for RDF. Whereas the business logic of an application is implemented in a host programming language like Java or C++, the query language provides an interface to the storage of persistent data. These languages often use different data models which cause the well-known impedance mismatch problem.

XQuery is a declarative programming language that can be applied on all application tiers and thus leverages a unified technology stack by means of the XML data model. XQuery is very well suited for querying and manipulating data that is stored in XML collections. Yet, a large number of legacy applications still exist in companies which produce SQL code as an interface to a relational database on the one hand and SPARQL code to execute queries on RDF documents on the other hand.

It is not feasible to replace all legacy databases and applications with XML databases and new programs written in XQuery at the same time. Therefore this thesis explores how legacy SQL and SPARQL code can be mapped to XQuery, which pre-conditions and limitations for an automated mapping hold and how well this transformation performs. In this way, all information can be moved to XML databases without having to change existing applications. Both old systems and applications written in XQuery can co-exist by accessing the same data. This eliminates the need for replication of different database systems and prevents inconsistencies.

Keywords: SQL, sql2xquery, SPARQL, sparql2xquery, XQuery, XML, cross-compiler, transformation
0 Overview

0.1 Contents

ABSTRACT .............................................................................................................................. 2

0 OVERVIEW ..................................................................................................................... 3
0.1 CONTENTS .................................................................................................................. 3
0.2 FIGURES ...................................................................................................................... 5
0.3 TABLES ....................................................................................................................... 5
0.4 CODE .......................................................................................................................... 6

1 INTRODUCTION ............................................................................................................ 8
1.1 BACKGROUND .......................................................................................................... 8
1.2 PROBLEM STATEMENT .............................................................................................. 8
1.3 CONTRIBUTION ......................................................................................................... 9
1.4 OUTLINE .................................................................................................................... 9

2 RELATED WORK ........................................................................................................ 10
2.1 TRANSFORMATION OF DATA .................................................................................. 10
2.2 TRANSLATING SPARQL TO SQL ........................................................................... 10
2.3 TRANSLATING SPARQL TO XQUERY ..................................................................... 10

3 FUNDAMENTALS ........................................................................................................ 12
3.1 SQL .......................................................................................................................... 12
3.2 SPARQL AND RDF .................................................................................................. 12
3.2.1 RDF .................................................................................................................... 12
3.2.2 The SPARQL Query Language .......................................................................... 13
3.3 XQUERY ................................................................................................................... 15

4 INFORMAL EXAMPLES .............................................................................................. 16
4.1 DATA MAPPING ....................................................................................................... 16
4.1.1 Relational Data .................................................................................................. 16
4.1.2 RDF ................................................................................................................... 17
4.1.3 XML Collections ............................................................................................... 17
4.2 VISUALIZATION OF DATA ...................................................................................... 18
4.3 SQL TO XQUERY ..................................................................................................... 19
4.3.1 Basic Queries ..................................................................................................... 19
4.3.2 Conditional Query .............................................................................................. 20
4.3.3 Nested Query ..................................................................................................... 20
4.3.4 Join of Two Tables ............................................................................................. 21
4.4 SPARQL TO XQUERY ............................................................................................ 22
4.4.1 Basic Query ....................................................................................................... 22
4.4.2 Alternative Pattern ......................................................................................... 24

5 DATA MAPPING ........................................................................................................ 26
5.1 IMPORT DATA TO XML COLLECTIONS ................................................................ 26
5.1.1 Relational Data .................................................................................................. 26
5.1.2 CSV Data ............................................................................................................ 27
Overview

5.1.3 RDF Documents.................................................................................................. 27
5.2 XQUERY RESULTS .......................................................................................... 28
  5.2.1 SQL Query Result .................................................................................. 28
  5.2.2 SPARQL Query Result ....................................................................... 28
6 MAPPING AND TRANSLATING LANGUAGE PATTERNS .................................. 30
  6.1 SQL TO XQUERY ......................................................................................... 30
    6.1.1 Expressions......................................................................................... 31
    6.1.2 Select Query ....................................................................................... 32
    6.1.3 Modifying the Database .................................................................... 34
  6.2 SPARQL TO XQUERY .................................................................................. 36
    6.2.1 Graph Pattern..................................................................................... 36
    6.2.2 Filter .................................................................................................. 38
    6.2.3 Expressions......................................................................................... 39
    6.2.4 Query Forms....................................................................................... 41
7 ARCHITECTURE ................................................................................................... 42
  7.1 SYSTEM ARCHITECTURE ............................................................................ 42
  7.2 SOFTWARE ARCHITECTURE ...................................................................... 43
    7.2.1 Lexer.................................................................................................. 44
    7.2.2 Parser................................................................................................ 44
    7.2.3 Error Checking ................................................................................... 45
    7.2.4 Optimizer............................................................................................ 46
    7.2.5 XQuery Code Generation ................................................................ 47
8 TESTING AND MEASUREMENTS ..................................................................... 48
  8.1 UNIT TESTS .................................................................................................. 48
  8.2 BENCHMARK .................................................................................................. 49
    8.2.1 SQL to XQuery ................................................................................... 50
    8.2.2 SPARQL to XQuery .......................................................................... 52
9 CONCLUSION ....................................................................................................... 56
  9.1 SUMMARY .................................................................................................... 56
  9.2 LIMITATIONS ............................................................................................... 56
  9.3 FUTURE WORK ............................................................................................. 56
10 ACKNOWLEDGEMENT ...................................................................................... 58
11 APPENDIX A: GLOSSARY ............................................................................... 59
  11.1 TERMS ...................................................................................................... 59
  11.2 ABBREVIATIONS ...................................................................................... 59
12 APPENDIX B: MANUAL ..................................................................................... 61
  12.1 COMMAND LINE UTILITY ....................................................................... 61
    12.1.1 Parameters ....................................................................................... 61
  12.2 WEB INTERFACE ....................................................................................... 62
    12.2.1 GUI Elements ................................................................................. 62
13 APPENDIX C: IMPLEMENTATION .................................................................. 63
14 APPENDIX D: BENCHMARKS .......................................................................... 65
  14.1 SQL .......................................................................................................... 65
Table 17: Translation of attribute-value pairs from SQL to XQuery ....................................... 34
Table 18: Translation of insert statement from SQL to XQuery .............................................. 34
Table 19: Translation of update statement from SQL to XQuery ............................................ 34
Table 20: Translation of delete statement from SQL to XQuery ............................................. 35
Table 21: Translation of basic pattern from SQL to Xquery ................................................... 37
Table 22: Definition of a SPARQL triple pattern .................................................................... 37
Table 23: Translation of an optional graph pattern from SPARQL to XQuery....................... 37
Table 24: Translation of an alternative graph pattern from SPARQL to XQuery ................... 38
Table 25: Translation of group graph pattern from SPARQL to XQuery ............................... 38
Table 26: Translation of filter condition from SPARQL to XQuery ....................................... 38
Table 27: Translation of Boolean expressions from SPARQL to XQuery .............................. 39
Table 28: Translation of Boolean operators from SQL to XQuery.......................................... 39
Table 29: Translation of comparison operators from SQL to XQuery .................................... 39
Table 30: Translation of atomic expressions from SPARQL to XQuery ................................ 39
Table 31: Translation of list of variables from SPARQL to XQuery ....................................... 40
Table 32: Translation of list of expressions for order-by from SPARQL to XQuery.............. 40
Table 33: Translation of list of expressions for order-by from SPARQL to XQuery.............. 40
Table 34: Translation of namespace definition from SPARQL to XQuery............................. 41
Table 35: Translation of SELECT query from SPARQL to XQuery ...................................... 41
Table 36: Translation of ASK query from SPARQL to XQuery............................................. 41
Table 37: Size of input data for different scaling factors ......................................................... 49
Table 38: Offsets for “heating period” of engines ................................................................... 50
Table 39: Time for translating SQL queries to XQuery........................................................... 50
Table 40: Time for translating SPARQL queries to XQuery................................................... 52
Table 41: Command line parameters ....................................................................................... 61
Table 42: Description of GUI elements ................................................................................... 62

0.4 Code

Code 1: RDF triples.................................................................................................................. 12
Code 2: SPARQL example query ............................................................................................ 13
Code 3: Basic Graph Pattern ....................................................................................................14
Code 4: Filter Function ............................................................................................................ 14
Code 5: Named Graph Pattern ................................................................................................. 14
Code 6: Optional Pattern .......................................................................................................... 15
Code 7: Optional Pattern .......................................................................................................... 15
Code 8: Group Graph Pattern ................................................................................................... 15
Code 9: RDF document representing the Periodic System of the Elements......................... 17
Code 10: XQuery code to insert data into XML collection ..................................................... 18
Code 11: SQL input – get the name of all elements................................................................. 19
Code 12: XQuery output – get the name of all elements ......................................................... 19
Code 13: SQL input – join of two tables.................................................................................. 21
Code 14: XQuery output – join of two tables .......................................................................... 21
Code 15: SQL input – return lightest element.......................................................................... 21
Code 16: XQuery output – return lightest element................................................................. 21
Code 17: SQL input – return elements conditionally and order result ................................... 20
Code 18: XQuery output – return elements conditionally and order result ............................. 20
Code 19: SQL input – get all elements that have a name and a color ....................................... 20
Code 20: XQuery output – get all elements that have a name and a color ............................. 23
Code 21: SPARQL input – get all elements with given number or color................................. 24
1 Introduction

1.1 Background
Different technologies may be involved today when data has to be stored and retrieved. A majority of information is still available in relational database systems which are based on the relational model developed by Edgar F. Codd in 1970. SQL is a declarative language that has been designed for querying and managing data stored in a relational database which can best be regarded as a set of flat tables containing lines of data. The theory behind such systems is well understood which allows a relational database server to be very stable and well optimized. Vendors of database systems offer server software (like MySQL Cluster [14]) which can distribute data among several machines to increase scalability, reliability and performance. These systems offer a very high availability of more than 99.99% which makes a relational database the first choice for mission critical data like for the banking or stock exchange business.

Information about World Wide Web resources is often stored by means of the Resource Description Framework [19]. Popular examples are personal data used in social networks or metadata about a collection of images or videos. The Semantic Web contains large RDF knowledge bases that can be exchanged between different applications by expressing the semantics of the different data sources with ontologies. Information usually is retrieved from RDF knowledge bases using the SPARQL language [25] which is often referred to as the query language for RDF.

An increasing amount of information is stored in XML databases. In recent years, XML has become the de-facto standard for information exchange in the World Wide Web, because it is very well suited for the representation of structured information. If XML is used as an interface to other software in a heterogeneous environment, it is an obvious advantage to store the information in an XML database. XQuery even goes one step further and applies to XML data model for the representation of the internal information of an application as well and thus eliminates the impedance mismatch problem completely.

1.2 Problem Statement
Relational databases, the Semantic Web and XML databases have different data models which lead to problems when data from disparate sources of information has to be integrated. Data being distributed among heterogeneous databases may lead to scattering and lost of information, because it is not possible to restore all conjunctions fast enough when it is necessary.

Different workarounds for the integration of data from different information sources are known. Mirroring fetches information from a source database, performs a conversion of the data if necessary, and imports the data to the target database where it is actually needed for a client application. In this case inconsistency problems can hardly be prevented in case of frequent updates. Therefore, another approach is to convert the data for each query by means of adapters. This helps with inconsistency, but introduces performance issues, because a lot of data has to be transferred and converted repeatedly for each query.
Switching all applications to the same programming language and databases system is not a general solution either, because often a large number of legacy applications are available, which are too expensive or time-consuming to be re-implemented. Of course, the human factor also has to be considered as well: For example, a certain domain specialist may prefer a single language. Often it is even advantageous to use different programming languages, because several problems are easier or more elegant to solve in a certain language.

In addition, if SQL or SPARQL is integrated into a host programming language like Java or C++ for the implementation of the business logic, an impedance mismatch problem occurs, because these languages use different data models.

1.3 Contribution
The xQL-to-XQuery tool has been developed to establish a bridging of heterogeneous environments by translating both SQL and SPARQL to XQuery at runtime. Thus, a way is provided to answer SQL, SPARQL and XQuery requests from the same XML database transparently. The thesis shows how an automatic translation can be implemented on the operator level, which pre-conditions have to hold and how well the generated code performs. For the transformation of information from legacy database systems to XML databases, an import function for RDF and CSV is given.

To the best of our knowledge, no other solution providing an automated SQL to XQuery translation is available so far.

1.4 Outline
The thesis is organized as follows: Section 2 shows which related work has already been done for this topic. Section 3 gives a short introduction to SQL, SPARQL and XQuery. Some informal examples for a translation to XQuery are presented in Section 4. The mapping of relational data and RDF data to XML collections is described in Section 5. A detailed and more formal description of the translation from SQL and SPARQL to XQuery is shown in Section 6. In Section 7 the system and software architecture of the xQL-to-XQuery tool is sketched. An evaluation of the performance is presented in Section 8. The thesis is concluded in Section 9 by a summary and a vision for future work.


2 Related Work

2.1 Transformation of data

The most obvious solution for providing different applications using SPARQL, SQL and XQuery with the same dataset is the transformation and replication of data instead of the translation of the query language.

[32] shows how to map RDF to XML and vice versa which enables RDF data to be imported to XML collections. [13] is the ISO standard which describes how to represent relational data in XML. The standard defines how data types in SQL can be mapped to XML Schema types, how to convert names like for tables and columns to XML tag names and attributes.

As expected, the transformation of data brings about series consistency problems and is not a suitable solution for data that is updated frequently.

2.2 Translating SPARQL to SQL

Several proposals have already been made for translating SPARQL to SQL. The motivation is to make the data stored in relational databases available to Semantic Web applications. [8] proposes a translation of SPARQL which produces not nested SQL code for more efficient processing by relational database servers. The authors propose model-based algorithms in order to implement each SPARQL operator by means of SQL query augmentation. Benchmarks show that the performance of this approach is quite good compared to other methods which produce nested SQL code.

A similar approach is taken by [10] which introduces a relational algebra based on the semantics of SPARQL. Like [8], this paper produces SQL code that is semantically equivalent to the SPARQL input. After having provided a translation to SQL for SPARQL triples, basic graph patterns and optional patterns, the authors present a number of optimizations in order to produce simpler and more efficient SQL output. Again, benchmarks show that performance if quite good compared to native RDF storage systems.

The main problem with the translation to SQL is that this approach does not solve the impedance mismatch problem. Additional difficulties occur by mapping the SPARQL type system to SQL types.

2.3 Translating SPARQL to XQuery

The SPARQL2XQuery Framework [6] introduces a formal mapping from OWL ontology to XML Schema and a methodology for the translation from SPARQL to semantically equivalent XQuery code in order to answer SPARQL queries from XML databases. The mapping between XML Schema and RDF is either generated automatically or provided manually by a domain expert.

Although the authors claim that their implementation is complete, they do not show how to translate named graphs and variable graph names. According to the W3c standard [25] it is
possible to combine data from different data sources (i.e. RDF documents) and it is even allowed to leave the name of the source open as a variable. The application developed for this thesis supports multiple data sources and named graphs.

In addition, the SPARQL2XQuery framework provides a way to combine Semantic Web and XML environments only. It does not cover the integration of information from relational database systems.
3 Fundamentals

This section gives a short introduction to the languages which are relevant for this thesis. Good references are [16] for SQL and [9] or [31] for XQuery. The best tutorial for SPARQL is the W3C standard [25] itself.

3.1 SQL

The Structured Query Language (SQL) is the most popular language used as an interface to a relational database management system (DBMS). SQL originally has been designed by IBM in the early 1970s for their System R and has been standardized by the American National Standards Institute the first time in 1986 as SQL-86. The most recent revision is SQL:2008 which is standardized in ISO/IEC 9075:2008.

SQL consists of several subsets:

- The **Data Definition Language (DDL)** defines the schema of a database. The most important keywords in this context are CREATE, ALTER, DROP and RENAME.
- The **Data Manipulation Language (DML)** provides statements to manage objects in existing instances of the schema. Well-known examples of the DML are SELECT, INSERT, UPDATE and DELETE.
- Access privileges can be defined by the **Data Control Language (DCL)**. Frequently used keywords are GRANT to give a user permission to a database or table and REVOKE to withdraw the permission again.
- The **Transaction Control (TCL)** provides statements to group statements together to be executed atomically.

For this thesis, only the DML of SQL is relevant, because the schema of the database (with respect to the DDL) is defined by means of an XML configuration file of the XQL-to-XQuery tool (see Section 7.2). The full SQL standard is very complex and comprehensive, so only a subset can be implemented for this work.

3.2 SPARQL and RDF

As SPARQL is closely related to RDF, an introduction to RDF is given first.

3.2.1 RDF

The Resource Description Framework (RDF) is W3C Recommendation [19] for the specification of metadata models for the Semantic Web. An RDF document expresses information about a resource of the web by a list of subject-predicate-object triples. The following example shows 2 RDF triples which make a statement about the chemical element Hydrogen:

```
<http://www.xql2xquery.org/chemistry#H>
  <http://www.xql2xquery.org/chemistry/name> "hydrogen" .
<http://www.xql2xquery.org/chemistry#H>
  <http://www.xql2xquery.org/chemistry/number> 1 .
```

Code 1: RDF triples
Structure of a Document
The subject (“H” in the example) identifies a resource on the web by a Uniform Resource
Identifier (URI), the predicate represents the name of a property (e.g. “name” or “number”),
whereas the object is the value (“Hydrogen” or “1”). An RDF document may contain a large
number of triples which can be visualized by a directed graph.

Serialization
There are different formats for the serialization of RDF data. The most common format is
RDF/XML [20] which stores data in XML syntax. Additional formats are Notation 3 [21], N-
Triples [22] and Turtle [23] which are more suitable for human readability than RDF/XML.
The formats are semantically equivalent and can be converted into one another using tools
like CWM [18] by W3C.

3.2.2 The SPARQL Query Language
SPARQL is available as a W3C Recommendation [25] since 2008 and is known as the query
language for RDF.

Query Forms
In SPARQL, different query forms are available:
• SELECT: return the value of variables which may be bound by a matching query pattern
• ASK: return true if a given query matches and false if not
• CONSTRUCT: return an RDF graph by substituting the values in given templates
• DESCRIBE: return an RDF graph which defines the matching resource
(CONSTRUCT and DESCRIBE are not used frequently and are not covered by this thesis)

This introduction focuses on the SELECT form, because this is the most commonly used
query type. The following example shows a SELECT query which fetches the color of the
element “silicon” from the database:

```
PREFIX chemistry: <http://www.xql2xquery.org/chemistry#>
SELECT ?col
WHERE
{
?element chemistry:name "silicon".
}
ORDER BY ?col
LIMIT 10
OFFSET 4
```

Code 2: SPARQL example query

Structure of a Select Query
The PREFIX section defines all namespaces which are used in the query. The SELECT clause
lists one or many variables to return as a result. A variable in SPARQL is denoted
equivalently by either a “?” or “$” character. If a triple containing a variable matches, the
variable is “bound” to that value. If several variables are contained in the WHERE clause or if
multiple matches of a variable occur, all possible combinations are returned in the result.
More complex conditions in the WHERE clause can be constructed with the building blocks basic graph patterns and group graph patterns, which are explained subsequently.

**Basic Graph Pattern**

A basic graph pattern contains a set of triple patterns. One triple pattern consists of a whitespace-separated list of subject, predicate and object which may contain variables on each position. If several triple patterns are contained in one basic pattern, these triples are combined by a conjunction, i.e. a basic graph pattern only matches if all triples match.

```sparql
{
    ?element chemistry:name "silicon".
    ?element chemistry:atomicNumber ?number.
}
```

*Code 3: Basic Graph Pattern*

**Constraints**

A SPARQL FILTER function can be added to a basic graph pattern in order to restrict the result according to Boolean conditions. The following pattern only retrieves elements whose name end in “ium”. This is achieved by a function that defines a regular expression which is equivalent to the Regex syntax used for XQuery 1.0 and XPath 2.0 [24].

```sparql
{
    ?element chemistry:name ?name.
    FILTER regex(?name, "ium")
}
```

*Code 4: Filter Function*

**Named Graph Pattern**

Additional datasets (i.e. RDF documents) may be added to the query as so called named graph which is indicated by the GRAPH keyword. The patterns within that section are matched against the named graph which has been selected. By this construct, information from several RDF sources may be combined. The name of the graph can either be specified explicitly by an IRI reference or left open as a variable (as shown in the following example). Each query must define a default graph which is active, whenever no named graph has been declared by a GRAPH keyword.

```sparql
GRAPH ?src
{
    ?compound comp:name ?compoundName.
}
```

*Code 5: Named Graph Pattern*

In the following examples, a *pattern* may be replaced by a basic graph pattern, a group graph pattern, an optional graph pattern, an alternative graph pattern or a named graph pattern.
Optional Patterns
The purpose of an optional pattern is to supplement the solution with additional information. If the pattern within an OPTIONAL clause matches, the variables defined within that pattern are bound to one or many solutions. If the pattern does not match, the solution remains unchanged.

```
pattern OPTIONAL { pattern } OPTIONAL { ... }
```

Code 6: Optional Pattern

Alternative Patterns
The semantics of the UNION keyword is the combination of all solutions. The total pattern matches if one or several pattern matches. If more than one alternative has been found, all possible solutions are added to the result.

```
{ pattern } UNION { pattern } UNION { ... }
```

Code 7: Optional Pattern

Group Graph Patterns
In a group graph pattern all patterns must match.

```
{ pattern } { pattern } { ... }
```

Code 8: Group Graph Pattern

The set of solutions (which has been retrieved by a matching WHERE clause) may be influenced by a solution modifier.

Solution Modifiers
The ORDER BY, LIMIT and OFFSET expressions are called “solution modifiers” as they influence the current solution. The ORDER BY clause lists one or many variables which determine the order of the result. The LIMIT clause restricts the number of results to the given value. For navigating through the result, an OFFSET can be set to start the result at a certain number.

3.3 XQuery
XQuery is a declarative and Turing complete programming language which has originally been designed to extract information and perform transformations on XML data. XQuery 1.0 is a W3C recommendation [31] since 2007. It includes XPath 2.0 [30] as a sublanguage. The powerful features for retrieving and transforming data of XQuery 1.0 have been supplemented with extensions to the language in order to support updates [29], scripting [28] and full-text search [27].

From the Turing completeness of XQuery it follows that all programs written in either language can be expressed equivalently by a program written in XQuery. Yet, theory gives no clue how efficient these programs are and how concisely they can be expressed. This translation is the topic of the remaining sections of this thesis.
4 Informal Examples

As motivated in Section 1, the goal of this thesis is to provide a translation from SQL and SPARQL to XQuery. In this way, queries from all 3 languages can be answered from the same XML database transparently. This section demonstrates the translation by means of some informal examples first. A more formal description of the translation is shown in Section 6 later.

4.1 Data Mapping

The queries are demonstrated by means of data representing the Periodic Table of the Elements. The example scenario supposes that the same data shall be stored both in a relational database and as an RDF document. This introductory section shows informally, how this data can be mapped to an XML collection.

4.1.1 Relational Data

In a relational database the information is stored in a set of tables. Each table has a specified set of columns which are identified by a name unique within a table. A horizontal row corresponds to one element of data. For each table a set of columns may be chosen which identifies a tuple uniquely. This set of column names is called a primary key and is marked by underlining.

The following tables show an example database containing the Periodic System of Elements in the table “element” and an additional table “gas” which represents some gases. Each element has a unique ID, a name and a number. A gas has a unique ID, a reference to an element (only one in this simple example) and a weight. The column “element” is called a foreign key to the “element” table.

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>number</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>hydrogen</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>He</td>
<td>helium</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Li</td>
<td>lithium</td>
<td>3</td>
<td>silvery white/grey</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 1: Relational database representing the Periodic System of the Elements

<table>
<thead>
<tr>
<th>ID</th>
<th>element</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2</td>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>He</td>
<td>He</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 2: Relational database representing gases
4.1.2 RDF

The following code shows an RDF tree which stores the same data as described in Section 4.1.1. The RDF document is a flat list of entries. The name of each entry below the `<rdf:RDF>` tag corresponds to one tuple of a relational table with the name of the table used as tag name. The columns and the value per tuple are added as nested tags. Each entry has an identifier, indicated by `<rdf:ID>`, which must be unique for the RDF document. A reference to another entry is represented by the attribute `<rdf:resource>`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [ ... ]>
<rdf:RDF>
  <Element rdf:ID="H">
    <name>hydrogen</name>
    <number>1</number>
  </Element>
  <Element rdf:ID="He">
    <name>helium</name>
    <number>2</number>
  </Element>
  ... 
  <Gas rdf:ID="H2">
    <element rdf:resource="#H"/>
    <weight>2</weight>
  </Gas>
  ...
</rdf:RDF>
```

Code 9: RDF document representing the Periodic System of the Elements

4.1.3 XML Collections

The data from both the relational table in 4.1.1 and the RDF document in 4.1.2 can be imported in an XML collection by means of XQuery insert functions which are currently not standardized. For the remaining thesis, the functions from the Zorba [35] collection module are used. The insert function `coll:insert-nodes-last()` takes the name of the collection as first parameter and the sequence of nodes to be inserted as a second one. The name of the database and the name of the RDF document correspond to the collection name in this work.

The structure of the RDF document has been adopted for the XML collection which reduces the import of RDF to an XML collection to calling an insert function which contains the RDF entries within the `<rdf:RDF>` tag. It is necessary to declare the RDF namespace in the XQuery document which imports the data.

The following example shows the XQuery code which inserts some element data into a collection with the name “chemistry”.

```xml
......
18 Informal Examples

```
import module namespace coll="http://www.zorba-xquery.com/...functions";
declare namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
coll:insert-nodes-last(
  "chemistry",
  
  (<Element rdf:ID="H">
    <name>hydrogen</name>
    <number>1</number>
  </Element>
  (<Element rdf:ID="He">
    <name>helium</name>
    <number>2</number>
  </Element>
  ...
  );
```

Code 10: XQuery code to insert data into XML collection

4.2 Visualization of Data

In order to simplify the presentation of the examples in the next sections, the following notation is introduced to visualize data in an XML document:

- **<tagname>** XML tag with a name `<tagname>`
- **<attr.name> [<attr.value>]** XML attribute with a name `<attr.name>` and value `<attr.value>`
- **<body content>** Content `<body content>` of an XML tag

Figure 1: Notation for visualization of XML data
Applying this notation to the sample data, the following graph can be generated:

![Graph for Periodic System of the Elements](image)

Figure 2: Graph for Periodic System of the Elements

### 4.3 SQL to XQuery

This section shows some informative examples for the conversion of SQL code to XQuery. A database schema according to Section 4.1 is assumed.

#### 4.3.1 Basic Queries

This query returns the name of all elements which are stored in the database. I.e. only the column “name” is included in the result:

#### 4.3.1.1 Visualization of the Query

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>number</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>hydrogen</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>He</td>
<td>helium</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Li</td>
<td>lithium</td>
<td>3</td>
<td>silvery white/grey</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3: SQL query – get the name of all elements

#### 4.3.1.2 SQL

```sql
SELECT name from element;
```

Code 11: SQL input – get the name of all elements

#### 4.3.1.3 XQuery

In the XQuery representation of this query the collection has to be loaded first. The next line implements a loop over all elements. The XPath expression is required to limit the result to elements and to exclude the gases, which are stored in the same collection. A result tag is generated for each matching element containing the selected attributes as content.
20 Informal Examples

let $doc_default := fn:collection("chemistry")
for $element in $doc_default[name(.) = "element"]
return
  <result>
    <name>{data($element/name)}</name>
  </result>

Code 12: XQuery output – get the name of all elements

4.3.2 Conditional Query

This query returns the name of all elements which is renamed to title and the number renamed as weight. The result is restricted to elements of which the atomic number is greater than 5 and the name is not “Oxygen”.

4.3.2.1 SQL

```
SELECT name AS title, number AS weight
FROM element e
WHERE number > 5 AND NOT name = 'oxygen'
ORDER BY name;
```

Code 13: SQL input – return elements conditionally and order result

4.3.2.2 XQuery

For the translation to XQuery, again, the collection is loaded and a loop over the elements is generated. As the result shall be ordered by name, an order by clause is added. The order in SQL is case-insensitive, therefore the value is transformed to lower-case. As the fn:lower-case() function offered by XQuery returns an error for numbers, non-string values have to be excluded from the transformation which is encapsulated by the xqllib:ignoreCase() function from the xqllib module, which has been developed for this work. The implementation is given in Appendix C. Similarly to SQL, the restriction is added as a where clause.

```
let $doc_chemistry := fn:collection("chemistry")
for $e in $doc_chemistry[name(.) = "element"]
order by xqllib:ignoreCase($e/name)
where ($e/num > 5) and not($e/name = "oxygen")
return
  <result>
    <title>{data($e/name)}</title>
    <weight>{data($e/num)}</weight>
  </result>
```

Code 14: XQuery output – return elements conditionally and order result

4.3.3 Nested Query

The next query returns the name and the number of the lightest element. A nested query selects the minimum atomic number of all elements. The outer query shows only those elements for which the number is equivalent to the minimal one.
4.3.3.1 SQL

```sql
SELECT name, num
FROM element
WHERE number = (
    SELECT min(e.number)
    FROM element e
);
```

Code 15: SQL input – return lightest element

4.3.3.2 XQuery

The translation to XQuery is straight-forward and quite similar to SQL. A nested query retrieves the minimal value which is returned by the inner query as a single number.

```xml
let $doc_chemistry := fn:collection("chemistry")
for $element in $doc_chemistry[name(.) = "element"]
where data($element/num) = (
    let $e := $doc_chemistry[name(.) = "element"]
    return
    min($e/num)
)
return
<result>
    <name>{data($element/name)}</name>
    <num>{data($element/num)}</num>
</result>
```

Code 16: XQuery output – return lightest element

4.3.4 Join of Two Tables

The query returns all gases for the element “hydrogen”. In this example an inner join of two tables is shown.

4.3.4.1 SQL

```sql
SELECT c.compoundName
FROM compound c JOIN elem e ON c.elemName = e.name
WHERE e.name = "hydrogen";
```

Code 17: SQL input – join of two tables

4.3.4.2 XQuery

The XQuery representation implements the join by an additional loop with an XPath expression for the join condition.
let $doc_chemistry := fn:collection("chemistry")
for $g in $doc_chemistry[name(.) = "gas"]
for $e in $doc_chemistry[
    name(.) = "elem" and ($g/element/@rdf:resource = @rdf:ID)
] where ($e/name = "hydrogen")
return
  <result>
    <compoundName>{data($c/compoundName)}</compoundName>
  </result>

Code 18: XQuery output – join of two tables

4.4 SPARQL to XQuery

In this section several different examples for the conversion of SPARQL to XQuery are given. Again, the schema of Section 4.1 is assumed for the storage.

4.4.1 Basic Query

The next query returns the name and the color of elements. Only elements are included in the result which have both a name and a color (i.e. the variables have to be bound to a value).

4.4.1.1 Visualization of the Query

This query can be visualized by the following figure:

![Figure 3: Element name and color](image-url)
4.4.1.2 **SPARQL**

The SPARQL query defines the prefix “chemistry” for the namespace of the chemistry example. The SELECT clause adds the variables ?name and ?color to the result. The WHERE clause contains 2 triples which bind both the variable ?name to the name of the element and ?color to its color.

```xml
PREFIX chemistry: <http://www.xql2xquery.org/chemistry#>
SELECT ?name ?color
WHERE
{
 ?element chemistry:name ?name.
}
```

Code 19: SPARQL input – get all elements that have a name and a color

4.4.1.3 **XQuery**

Again, in the translation to XQuery the “chemistry” collection is loaded and the elements are selected. 3 nested loops are required, because 3 variables occur in the query and one entry is added to the result for each possible combination (binding) of the variables. In the most general case there might be elements which have more than one color and several names.

It is important to note that the WHERE clause restricts the result to elements for which the element, the color and the name are not empty. The name of the variables always corresponds to the tag name, as there is no way in SPARQL to assign the variable to a different name in the result.

```xml
declare namespace pse "http://www.xql2xquery.org/chemistry ";
let $doc_pse := fn:collection("pse")
for $node_element in $doc_pse[@rdf:ID]
for $value_color in xqllib:getData("pse",$node_element/color)
for $value_name in xqllib:getData("pse",$node_element/name)
order by $value_name
where fn:exists($node_element) and fn:exists($value_color)
    and fn:exists($value_name)
return
    <result>
        <name>{data($value_name)}</name>
        <color>{data($value_color)}</color>
    </result>
```

Code 20: XQuery output – get all elements that have a name and a color

Even for a very simple SPARQL query, the corresponding XQuery code becomes quite verbose. This does not mean that the XQuery representation is less efficient, but it is much more concise and elegant to express such queries in SPARQL.
4.4.2 Alternative Pattern

An alternative pattern expresses a disjunction of different conditions. This example fetches all elements from the database which have either an atomic number “13” or a green color.

4.4.2.1 Visualization of the Query

This query can be visualized by the following figure:

![Figure 4: Alternative conditions](image)

4.4.2.2 SPARQL

The SPARQL query defines two basic graph patterns which are combined by the `UNION` keyword. The variable `?name` can be bound by both sub patterns and thus returns a value if either graph matches.

```
PREFIX chemistry: <http://www.xql2xquery.org/chemistry#>
SELECT ?name
WHERE
{
  { ?element chemistry:name ?name;
    chemistry:number 13.
  }
  UNION
  { ?element chemistry:name ?name;
    chemistry:color "green".
  }
}
```

*Code 21: SPARQL input – get all elements with given number or color*
4.4.2.3 XQuery

The alternative pattern is implemented by two sub graphs in XQuery. The result of the two queries is combined by creating a sequence of both return values.

```
declare namespace chemistry "http://www.xql2xquery.org/chemistry";
let $doc_chemistry := fn:collection("chemistry")

let $GRAPH_1 :=
  for $node_element in $doc_chemistry[@rdf:ID and number = 13]
  for $value_name in xqllib:getData("chemistry",$node_element/name)
  where fn:exists($node_element) and fn:exists($value_name)
  return
    <result>
      <name>{$value_name}</name>
    </result>

let $GRAPH_2 :=
  for $node_element in $doc_pse[@rdf:ID and color = "green"]
  for $value_name in xqllib:getData("chemistry",$node_element/name)
  where fn:exists($node_element) and fn:exists($value_name)
  return
    <result>
      <name>{$value_name}</name>
    </result>

return ($GRAPH_1,$GRAPH_2)
```

Code 22: XQuery output – get all elements with given number or color

A nested loop for each of the variables is not a good translation of the alternative pattern, because it leads to very large intermediate results. The advantage of this solution is that both graphs can be calculated independent of each other which make that query parallelizable.
5 Data Mapping

To answer a query in SQL and SPARQL equivalently on a XML database [33], both a translation of the native code to XQuery and a mapping of the data are required. Section 5.1 shows how data from different sources can be imported to XML collections. The format chosen for the output which is returned by the XQuery code generated by xQL-to-XQuery is described in Section 5.2.

5.1 Import Data to XML Collections

The XML collection can be populated with data from different sources. The information is added to the database by means of an XQuery insert function from the Zorba [35] collection module.

5.1.1 Relational Data

With the current version of Zorba, no DDL support is available. As the translation tool needs to have knowledge about the schema of the data, this information is provided by means of a mapping file (mapping.xml), which is passed to the xQL-to-XQuery tool as a parameter. This XML file maps the tables and attributes of the relational database to the according XML collection and dataset. A dataset is the term used for different classes of nodes which are stored as a sequence in a collection.

The mapping of SQL data types to XML Schema types is described by the ISO 9075-1 [13] standard. The following table shows the correlation for the most important SQL types that can be mapped to a XML Schema Type.

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>XML Schema Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT, INTEGER, SMALLINT, BIGINT</td>
<td>xs:integer</td>
</tr>
<tr>
<td>NUMERIC, DECIMAL</td>
<td>xs:decimal</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>xs:boolean</td>
</tr>
<tr>
<td>TEXT, VARCHAR</td>
<td>xs:string</td>
</tr>
<tr>
<td>DATE</td>
<td>xs:date</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>xs:datetime</td>
</tr>
</tbody>
</table>

Table 4: Mapping SQL types to XML types

A large number of rules are given by the ISO standard to map the types more precisely. For example a SQL UNSIGNED TINYINT can be further restricted in XML Schema by adding <xs:minInclusive value="0"/> and <xs:maxInclusive value="255"/>. The length $\ell$ of a SQL VARCHAR is represented by <xs:length value="\ell"/>. If the value of a SQL attribute may be NULL, this is indicated in the XML Schema by nillable="true".

The ISO standard defines exact specifications for the translation of character sets, encoding of binary data, table and attribute names.

For this work, the mapping of the attributes is done manually by a domain expert by specifying the corresponding column name in mapping.xml. If no mapping descriptor is available, the same attribute name in SQL and XML is assumed by the translation tool.
5.1.2 CSV Data

The xQL-to-XQuery tool includes a parser of CSV data which generates an XQuery insert function call for each line of the CSV file. The assignment of the attribute name to a column number is implemented by a XML configuration file.

The following code shows an example for a definition of the CSV columns. There may be different CSV formats for the same collection and dataset.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<import>
  <collection name="chemistry">
    <format name="elem_data">
      <dataSet>element</dataSet>
      <attributes>
        <attribute name="name"/>
        <attribute name="number"/>
        <attribute name="color"/>
      </attributes>
    </format>
  </collection>
</import>
```

Code 23: CSV configuration file

A CSV file in the format defined by the previous configuration file may look like this:

```
1,hydrogen,
3,lithium,silvery white/grey
11,sodium,silvery white
```

Code 24: CSV data

As the comma is a separation character, this character must be removed or replaced for each value of a column before the CSV file is created.

5.1.3 RDF Documents

As the format for the XML collections has been derived from the RDF/XML format, data from RDF/XML documents can be imported natively. RDF and XML collections use the same XML Schema type system. A sequence of nodes is directly generated out of the children of the `<rdf:RDF>` tag.

Both RDF data in N3 and T-Triple format can be inserted to XML collections using the translation tool xQL-to-XQuery by converting the document to RDF/XML first using Cwm - The general-purpose data processor for the semantic web [18], which is maintained by W3C.
5.2 XQuery Results

5.2.1 SQL Query Result

The result of a SQL query which has been translated to XQuery is returned in a simple XML format which can be mapped to a table easily. The next example shows such an output:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<output>
  <result>
    <title>Hydrogen</title>
    <weight>1</weight>
  </result>
  <result>
    <title>Helium</title>
    <weight>2</weight>
  </result>
</output>
```

Code 25: SQL output in a simple XML format

A `<result>` tag is added for each matching tuple. For each attribute a child tag is added to the result which contains the value of this attribute in the body. In a result of a SQL query, all tuples have the same set of attributes.

5.2.2 SPARQL Query Result

The output format of a SPARQL query is standardized by W3C [26] and is available as a W3C recommendation from 15 January 2008. For this reason the result which is returned by the generated XQuery code is generated in a way such that this standard is satisfied. The output is formatted by a function of the xqlLib module. The function takes the node sequence of the result as first parameter and the sequence of variables as second parameter.

```xml
xqlLib:formatSparqlXml ($input as node()*, $variables as xs:string+)
  as node()*
```

The following example shows an output of a SPARQL query, which is structured according to the SPARQL Query Results XML Format.
<?xml version="1.0" encoding="UTF-8"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="elementName"/>
    <variable name="color"/>
  </head>
  <results>
    <result>
      <binding name="elementName">
        <literal>hydrogen</literal>
      </binding>
      <binding name="color">
        <literal>colourless</literal>
      </binding>
    </result>
    <result>
      <binding name="elementName">
        <literal>helium</literal>
      </binding>
      <binding name="color">
        <literal>colourless</literal>
      </binding>
    </result>
  </results>
</sparql>

Code 26: SPARQL output satisfying the W3C standard

The header contains a list of all variables that may be bound in a result entry. Within the <results> a sequence of all possible combinations of bound variables is returned. Each <result> tag corresponds to one solution and includes one child element for each variable which is bound in this combination of values.

The xQL-to-XQuery tool supports both the SPARQL Query Results XML Format and a flat format which is used for the benchmarks because of performance. The output format can be chosen with the --output_format option as described in Appendix B.
6 Mapping and Translating Language Patterns

In Section 4 the translation from SQL and SPARQL has been demonstrated by means of some practical examples. This section will focus on a more formal description of the translation process.

6.1 SQL to XQuery

As the complete SQL-92 specification is defined on more than thousand pages, only a basic subset of the language can be shown here.

Translation function

Let \( \text{sql2xquery()} \) be a function, which takes a SQL query \( \text{IN}_{\text{SQL}} \) as an argument and returns the corresponding XQuery representation \( \text{OUT}_{\text{XQu}} \) as a result. This function is implemented by the xQL-to-XQuery tool. The following sections show the result of the mapping function for different input values. In each case it holds:

\[
\text{OUT}_{\text{XQu}} = \text{sql2xquery(\text{IN}_{\text{SQL}})}
\]

Mapping function

Let \( \text{mapping()} \) be a function, which takes a SQL table name or attribute name as an argument and returns the corresponding XQuery dataset name or attribute name as a result. The function \( \text{mapping()} \) is implemented by the mapping configuration file (\( \text{mapping.xml} \)) passed to the xQL-to-XQuery tool as an input parameter as described in Appendix B. The translation has to respect the mapping from SQL types to XML types which is described in Section 5.1.1.

\[
\text{name}_{\text{XQu}} = \text{mapping(\text{name}_{\text{SQL}})} \quad \text{if mapping file is available}
\]

\[
\text{name}_{\text{XQu}} = \text{name}_{\text{SQL}} \quad \text{otherwise}
\]

The following translation tables show the SQL code \( \text{IN}_{\text{SQL}} \) in the left column and the corresponding XQuery code \( \text{OUT}_{\text{XQu}} \) in the right. The function \( \text{sql2xquery()} \) takes a cell of the left column as an input and returns the corresponding right column. Different lines of the translation tables represent different possibilities. Variables can by replaced by a possible definition from a translation table. The completeness and correctness of the function has not been proofed formally but evaluated by comparing results with a MySQL server [15] for standard use-cases like [5] and examples taken from [16] and [12] which are described in Section 8.1.
6.1.1 Expressions

6.1.1.1 Names
For the following definitions it holds:
- \textit{attributeName} a attribute which is defined in the schema

\[
\begin{array}{|l|l|}
\hline
\text{expr}_{\text{SQL}} := & \text{expr}_{\text{XQu}} := \\
\text{attributeName} & \text{mapping}(\text{attributeName}) \\
\text{funct}_{\text{SQL}}(\text{expr}_{\text{SQL}}) & \text{funct}_{\text{XQu}}(\text{expr}_{\text{XQu}}) \\
\text{boolExpr}_{\text{SQL}} & \text{boolExpr}_{\text{XQu}} \\
\hline
\end{array}
\]

Table 5: Translation of expressions from SQL to XQuery

6.1.1.2 List of expressions
The following table implements a list of expressions with arbitrary length.

\[
\begin{array}{|l|l|}
\hline
\text{exprList}_{\text{SQL}} := & \text{exprList}_{\text{XQu}} := \\
\text{expr}_{\text{SQL}} & \text{expr}_{\text{XQu}} \\
\text{exprList}_{\text{SQL}}, \text{expr}_{\text{SQL}} & \text{exprList}_{\text{XQuery}}, \text{expr}_{\text{XQu}} \\
\hline
\end{array}
\]

Table 6: Translation of expression lists from SQL to XQuery

6.1.1.3 Atomic-Type Functions
The following table shows some examples for functions which take an atomic type as a parameter.

\[
\begin{array}{|l|l|}
\hline
\text{funct}_{\text{SQL}} := & \text{funct}_{\text{XQu}} := \\
\text{UCASE}(\text{expr}_{\text{SQL}}) & \text{fn:upper-case}(\text{expr}_{\text{XQu}}) \\
\text{LCASE}(\text{expr}_{\text{SQL}}) & \text{fn:lower-case}(\text{expr}_{\text{XQu}}) \\
\text{LEN}(\text{expr}_{\text{SQL}}) & \text{fn:string-length}(\text{expr}_{\text{XQu}}) \\
\text{ABS}(\text{expr}_{\text{SQL}}) & \text{fn:abs}(\text{expr}_{\text{XQu}}) \\
\text{ROUND}(\text{expr}_{\text{SQL}}) & \text{fn:round}(\text{expr}_{\text{XQu}}) \\
\hline
\end{array}
\]

Table 7: Translation of some atomic-type functions from SQL to XQuery

6.1.1.4 Group-by and Aggregate Functions
All attributes that are contained in \text{expr}_{\text{SQL}} must occur in \text{groupBy}_{\text{SQL}}.

\[
\begin{array}{|l|l|}
\hline
\text{expr}_{\text{SQL}} := & \text{expr}_{\text{XQu}} := \\
\text{AVG}(\text{expr}_{\text{SQL}}) & \text{fn:avg}(\text{expr}_{\text{XQu}}) \\
\text{COUNT}(\text{expr}_{\text{SQL}}) & \text{fn:count}(\text{expr}_{\text{XQu}}) \\
\text{MAX}(\text{expr}_{\text{SQL}}) & \text{fn:max}(\text{expr}_{\text{XQu}}) \\
\text{MIN}(\text{expr}_{\text{SQL}}) & \text{fn:min}(\text{expr}_{\text{XQu}}) \\
\text{SUM}(\text{expr}_{\text{SQL}}) & \text{fn:sum}(\text{expr}_{\text{XQu}}) \\
\hline
\end{array}
\]

Table 8: Translation of aggregate functions from SQL to XQuery
6.1.1.5 Boolean Expressions

<table>
<thead>
<tr>
<th>boolExprSQL :=</th>
<th>boolExprXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>fn:true()</td>
</tr>
<tr>
<td>FALSE</td>
<td>fn:false()</td>
</tr>
<tr>
<td>! boolOpSQL</td>
<td>not(boolExprXQu)</td>
</tr>
<tr>
<td>not boolOpSQL</td>
<td>not(boolExprXQu)</td>
</tr>
<tr>
<td>boolExprSQL boolOpSQL boolExprSQL</td>
<td>boolExprXQu boolOpXQu boolExprXQu</td>
</tr>
<tr>
<td>exprSQL compSQL exprSQL</td>
<td>boolExprXQu boolOpXQu boolExprXQu</td>
</tr>
</tbody>
</table>

Table 9: Translation of Boolean expressions from SQL to XQuery

6.1.1.6 Boolean Operators

<table>
<thead>
<tr>
<th>boolOPSQL :=</th>
<th>boolOPXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>and</td>
</tr>
<tr>
<td>OR</td>
<td>or</td>
</tr>
</tbody>
</table>

Table 10: Translation of Boolean operators from SQL to XQuery

6.1.1.7 Comparison Operators

<table>
<thead>
<tr>
<th>compSQL :=</th>
<th>compXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

Table 11: Translation of comparison operators from SQL to XQuery

6.1.2 Select Query

6.1.2.1 Query Structure
For the following definitions it holds:

• \( N \in \mathbb{N} \)

<table>
<thead>
<tr>
<th>tableSQL :=</th>
<th>nodesXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT exprListSQL FROM source1SQL, ..., sourceNSQL ( WHERE boolExprSQL )? ( GROUP BY groupBySQL )? ( ORDER BY orderListSQL )?</td>
<td>for $source1 := source1XQu ... for $sourceN := sourceNXQu ( group by groupByXQu )? ( where boolExprXQu )? ( order by orderListXQu )? return nodesXQu</td>
</tr>
</tbody>
</table>

Table 12: Translation of query structure from SQL to XQuery
For the following definitions it holds:

- $collName_{XQu}$ is the collection in which the data is stored
- $tableName_{SQL}$ a table which is defined in the schema
- $dataSet_{XQu} = mapping(tableName_{SQL})$
- $K \in \{1..N\}$

<table>
<thead>
<tr>
<th>$sourceK_{SQL}$</th>
<th>$sourceK_{XQu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tableK_{SQL}$</td>
<td>$nodes_{XQu}$</td>
</tr>
<tr>
<td>$tableNameK_{SQL}$</td>
<td>fn:collection(&quot;$collName_{XQu}$&quot;) /$dataSetK_{XQu}$</td>
</tr>
</tbody>
</table>

Table 13: Translation of names from SQL to XQuery

**6.1.2.2 Group-by**

The following table implements a list of variables for the aggregation with arbitrary length.

<table>
<thead>
<tr>
<th>$groupBy_{SQL}$</th>
<th>$groupBy_{XQu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$attributeName_{SQL}$</td>
<td>mapping($attributeName_{SQL}$)</td>
</tr>
<tr>
<td>$groupBy_{XQu}$ , $attributeName_{SQL}$</td>
<td>$groupBy_{XQu}$ , mapping($attributeName_{SQL}$)</td>
</tr>
</tbody>
</table>

Table 14: Translation of list of group-by from SQL to XQuery

**6.1.2.3 Ordering the Result**

The following table implements a list of expressions with arbitrary length. For each expression the sort order may be ascending or descending.

<table>
<thead>
<tr>
<th>$orderList_{SQL}$</th>
<th>$orderList_{XQu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$expr_{SQL}$</td>
<td>$expr_{XQu}$</td>
</tr>
<tr>
<td>$expr_{SQL}$ ASC</td>
<td>$expr_{XQu}$</td>
</tr>
<tr>
<td>$expr_{SQL}$ DESC</td>
<td>$expr_{XQu}$ descending</td>
</tr>
<tr>
<td>$orderList_{SQL}$ , $expr_{SQL}$</td>
<td>$orderList_{XQu}$ , $expr_{XQu}$</td>
</tr>
<tr>
<td>$orderList_{SQL}$ , $expr_{SQL}$ ASC</td>
<td>$orderList_{XQu}$ , $expr_{XQu}$</td>
</tr>
<tr>
<td>$orderList_{SQL}$ , $expr_{SQL}$ DESC</td>
<td>$orderList_{XQu}$ , $expr_{XQu}$ descending</td>
</tr>
</tbody>
</table>

Table 15: Translation of list of expressions for order-by from SQL to XQuery

**6.1.2.4 Formatting the Result**

The following table formats the result as sequence of tuples returned as a result.

<table>
<thead>
<tr>
<th>$nodes_{XQu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;result&gt;</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>for $node in exprList_{XQu}</td>
</tr>
<tr>
<td>&lt;varName&gt;</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>data($node)</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>&lt;/varName&gt;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>&lt;/result&gt;</td>
</tr>
</tbody>
</table>

Table 16: Translation of list of expressions from SQL to XQuery
6.1.3 Modifying the Database

The modified data can both be given explicitly or as the result of a nested expression. For the following definitions it holds:

- $K \in \{1..N\}$
- $\text{attributeName}_K$ an attribute which is defined in the schema
- $\text{value}_K$ a literal expression (e.g. a number or a string)

<table>
<thead>
<tr>
<th>$\text{updateData}_{\text{SQL}}$</th>
<th>$\text{updateData}_{\text{XQuery}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{table}_{\text{SQL}}$</td>
<td>$\text{nodes}_{\text{XQuery}}$</td>
</tr>
<tr>
<td>$\text{attributeName}_1 = \text{value}_1,$</td>
<td>$&lt;\text{result}&gt;$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$&lt;\text{attributeName}_1&gt;$</td>
</tr>
<tr>
<td>$\text{attributeName}_K = \text{value}_K,$</td>
<td>$\text{value}_1$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$&lt;/\text{attributeName}_1&gt;$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$&lt;\text{attributeName}_K&gt;$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\text{value}_K$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$&lt;/\text{attributeName}_K&gt;$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$&lt;/\text{result}&gt;$</td>
</tr>
</tbody>
</table>

Table 17: Translation of attribute-value pairs from SQL to XQuery

6.1.3.1 Insert

For the following definitions it holds:

- $\text{collName}_{\text{XQuery}}$ is the collection in which the data is stored
- $\text{tableName}_{\text{SQL}}$ a table which is defined in the schema
- $\text{dataSet}_{\text{XQuery}} = \text{mapping}(\text{tableName}_{\text{SQL}})$

<table>
<thead>
<tr>
<th>$\text{insertResult}_{\text{SQL}}$</th>
<th>$\text{insertResult}_{\text{XQuery}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{INSERT INTO}$ $\text{tableName}_{\text{SQL}}$</td>
<td>$\text{coll:insert-nodes-last}$</td>
</tr>
<tr>
<td>$\text{SET}$ $\text{updateList}_{\text{SQL}}$</td>
<td>$&quot;\text{collName}<em>{\text{XQuery}}&quot;,&quot;\text{updateList}</em>{\text{XQuery}}$</td>
</tr>
</tbody>
</table>

Table 18: Translation of insert statement from SQL to XQuery

6.1.3.2 Update

For the following definitions it holds:

- $\text{collName}_{\text{XQuery}}$ is the collection in which the data is stored
- $\text{tableName}_{\text{SQL}}$ a table which is defined in the schema
- $\text{dataSet}_{\text{XQuery}} = \text{mapping}(\text{tableName}_{\text{SQL}})$

<table>
<thead>
<tr>
<th>$\text{updateResult}_{\text{SQL}}$</th>
<th>$\text{updateResult}_{\text{XQuery}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{UPDATE}$ $\text{tableName}_{\text{SQL}}$</td>
<td>$\text{for }$ $\text{node in}$ $\text{updateList}_{\text{XQuery}}$</td>
</tr>
<tr>
<td>$\text{SET}$ $\text{updateList}_{\text{SQL}}$</td>
<td>$/*$</td>
</tr>
<tr>
<td>$\text{WHERE}$ $\text{boolExpr}_{\text{SQL}}$</td>
<td>$\text{return }$ $\text{replace}$</td>
</tr>
<tr>
<td>$\text{fn:collection}(&quot;\text{collName}_{\text{XQuery}}&quot;)$</td>
<td>$\text{fn:collection}(&quot;\text{collName}_{\text{XQuery}}&quot;)$</td>
</tr>
<tr>
<td>$/\text{dataSet}<em>{\text{XQuery}}[\text{boolExpr}</em>{\text{XQuery}}$</td>
<td>$/\text{dataSet}<em>{\text{XQuery}}[\text{boolExpr}</em>{\text{XQuery}}$</td>
</tr>
<tr>
<td>$\text{and}$</td>
<td>$\text{and}$</td>
</tr>
<tr>
<td>$\text{name}(.) = \text{name}($node$)$</td>
<td>$\text{name}(.) = \text{name}($node$)$</td>
</tr>
<tr>
<td>$\text{with}$</td>
<td>$\text{with}$</td>
</tr>
<tr>
<td>$\text{$node$}$</td>
<td>$\text{$node$}$</td>
</tr>
</tbody>
</table>

Table 19: Translation of update statement from SQL to XQuery
6.1.3.3 Delete

For the following definitions it holds:

- $collName_{XQu}$ is the collection in which the data is stored
- $tableName_{SQL}$ a table which is defined in the schema
- $dataSet_{XQu} = \text{mapping}(tableName_{SQL})$

<table>
<thead>
<tr>
<th>$deleteResult_{SQL}$ :=</th>
<th>$deleteResult_{XQu}$ :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{DELETE FROM} $tableName_{SQL}$ \text{WHERE} $boolExpr_{SQL}$</td>
<td>delete node $\text{fn:collection(&quot;collName}<em>{XQu}\text{&quot;)}$ $/dataSet</em>{XQu}[boolExpr_{XQu}]$</td>
</tr>
</tbody>
</table>

Table 20: Translation of delete statement from SQL to XQuery
6.2 SPARQL to XQuery

The rules for the translation which are presented in this section are derived from the SPARQL W3C standard [25].

Translation function

Let \( \text{sparql2xquery}() \) be a function, which takes a SPARQL query \( \text{IN}_{\text{SPA}} \) as an argument and returns the corresponding XQuery representation \( \text{OUT}_{\text{XQu}} \) as a result. This function is implemented by the xQL-to-XQuery tool. The following sections show the result of the mapping function for different input values. In each case it holds:

\[
\text{OUT}_{\text{XQu}} = \text{sparql2xquery}(\text{IN}_{\text{SPA}})
\]

The following translation tables show the SPARQL code \( \text{IN}_{\text{SPA}} \) in the left column and the corresponding XQuery code \( \text{OUT}_{\text{XQu}} \) in the right. The function \( \text{sparql2xquery}() \) takes a cell of the left column as an input and returns the corresponding right column. Different lines of the translation tables represent different possibilities. Variables can by replaced by a possible definition from a translation table. The completeness and correctness of the function has not been proofed formally but evaluated by comparing results to a ARQ SPARQL engine [3] for standard use-cases like [5] and examples taken from the W3C standard [25] as described in Section 8.1.

6.2.1 Graph Pattern

6.2.1.1 Basic Pattern

In a basic graph pattern a set of triple patterns must all match. For the following definitions it holds:

- Let \( \text{vars}(\text{pattern}_{\text{SPA}}) \) be a set of all variables which occur in the pattern.
- Let \( \text{subjVars}(\text{pattern}_{\text{SPA}}) \) be a set of all variables which occur in at least one subject. The library function \( \text{xqllib:getSubj()} \) fetches all subjects from the database.
- Let \( \text{predVars}(\text{pattern}_{\text{SPA}}) \) be a set of all variables which occur in at least one predicate but in no subject. The library function \( \text{xqllib: Pred($subj)} \) fetches all predicates from the database for a given subject.
- Let \( \text{objVars}(\text{pattern}_{\text{SPA}}) \) be a set of all variables which occur in at least one object but in no predicate and no subject. The library function \( \text{xqllib: getObj($pred)} \) fetches all objects from the database for a given predicate.
- Let \( \text{constants}(\text{pattern}_{\text{SPA}}, \text{varList}) \) be a set of all constant values which occur in a triple for a given combination of variables.
### Table 21: Translation of basic pattern from SQL to XQuery

In a triple pattern `subjectSPA, predicateSPA` and `objectSPA` all can either be a constant value or a variable.

<table>
<thead>
<tr>
<th><strong>Pattern</strong></th>
<th><strong>pattern</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{triplePattern SPA} := )</td>
<td>( \forall \text{subjName} \text{(subjVars(patternSPA))} )</td>
</tr>
<tr>
<td>( \text{...} )</td>
<td>( \text{for } $\text{subjName in xqllib:getSubj()} )</td>
</tr>
<tr>
<td>( \text{triplePattern SPA (filterSPA)*} )</td>
<td>( \forall \text{predName} \text{(predVars(patternSPA))} )</td>
</tr>
<tr>
<td>( \text{for } $\text{predName in xqllib:getPred(}$\text{subjName} )</td>
<td></td>
</tr>
<tr>
<td>( \text{forall objName (objVars(patternSPA))} )</td>
<td>( \text{for } $\text{objName in xqllib:getObj(}$\text{predName} )</td>
</tr>
<tr>
<td>( \text{where} )</td>
<td>( \forall \text{constant (constants(patternSPA, subjName, predName, objName))} )</td>
</tr>
<tr>
<td>( $\text{subjName} = \text{constant} )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( $\text{predName} = \text{constant} )</td>
<td>( \text{patternLSPA , patternRSPA} )</td>
</tr>
<tr>
<td>( $\text{objName} = \text{constant} )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( \forall \text{filterCondition (filterXQu)} )</td>
<td>( \text{patternLSPA , patternRSPA}) )</td>
</tr>
<tr>
<td>( \text{(and)? filterCondition} )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( \text{)}? )</td>
<td>( \text{patternLSPA , patternRSPA}) )</td>
</tr>
<tr>
<td>( \text{return} )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( \langle \text{result}&gt; )</td>
<td>( \text{patternLSPA , patternRSPA}) )</td>
</tr>
<tr>
<td>( \forall \text{varName (vars(patternSPA))} )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( &lt;\text{varName}{\text{data(}$\text{varName})} &lt;/\text{varName}&gt; )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( \langle \text{result}&gt; )</td>
<td>( \text{patternLSPA , patternRSPA}) )</td>
</tr>
</tbody>
</table>

### Table 22: Definition of a SPARQL triple pattern

### 6.2.1.2 Optional Graph Pattern

An optional pattern adds supplementary information to a given result. If the optional pattern does not match, the result remains unchanged. The optional pattern is implemented in XQuery by a function from the xqlLib module. The implementation is given in Appendix C.

<table>
<thead>
<tr>
<th><strong>pattern</strong></th>
<th><strong>pattern</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{patternSPA :=} )</td>
<td>( \text{xqllib:optional(} )</td>
</tr>
<tr>
<td>( \text{subjectSPA predicateSPA objectSPA .} )</td>
<td>( \text{patternLSPA , patternRSPA}) )</td>
</tr>
</tbody>
</table>

### Table 23: Translation of an optional graph pattern from SPARQL to XQuery

More than 2 consecutive optional patterns may be defined. Two of them at a time can be translated using the function `xqllib:optional()`. The result again can be taken as an input value for a further function call of `xqllib:optional()`. This works, because the `OPTIONAL` keyword is **left-associative**. This is the key-idea for the translation of graph patterns from SPARQL to XQuery.
According to [25] it holds:

\[
\text{pattern OPTIONAL \{ pattern \} OPTIONAL \{ pattern \}}
\]

is equivalent to

\[
\{ \text{pattern OPTIONAL \{ pattern \}} \} \text{ OPTIONAL \{ pattern \}}
\]

### 6.2.1.3 Alternative Graph Pattern

In an alternative graph pattern two possible patterns are evaluated and the union of both taken as a result. The alternative pattern can be expressed in XQuery by a sequence of the results of both patterns.

<table>
<thead>
<tr>
<th>patternSPA :=</th>
<th>patternXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ patternLSPA } \union { patternRSPA }</td>
<td>(patternLSPA, patternRSPA)</td>
</tr>
</tbody>
</table>

Table 24: Translation of an alternative graph pattern from SPARQL to XQuery

### 6.2.1.4 Group Graph Pattern

In a group graph pattern the set of graph patterns must all match. The pattern is implemented in XQuery by a function from the xqlLib module which is given in Appendix C.

<table>
<thead>
<tr>
<th>patternSPA :=</th>
<th>patternXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ patternLSPA } { patternRSPA }</td>
<td>xqllib:and(patternLSPA, patternRSPA)</td>
</tr>
</tbody>
</table>

Table 25: Translation of group graph pattern from SPARQL to XQuery

### 6.2.2 Filter

A filter restricts the solutions to those expressions for which the filter condition is true.

#### 6.2.2.1 Condition

<table>
<thead>
<tr>
<th>filterSPA :=</th>
<th>filterXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>\FILTER boolExprSPA</td>
<td>boolExprXQu</td>
</tr>
</tbody>
</table>

Table 26: Translation of filter condition from SPARQL to XQuery
6.2.2.2 Boolean Expressions

For the following definitions it holds:

- Let flagSPA be the flags for the REGEX function (e.g. “i” for case-insensitive).

<table>
<thead>
<tr>
<th>boolExprSPA :=</th>
<th>boolExprXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>fn:true()</td>
</tr>
<tr>
<td>FALSE</td>
<td>fn:false()</td>
</tr>
<tr>
<td>! boolExprSPA</td>
<td>not(boolExprXQu)</td>
</tr>
<tr>
<td>boolExprSPA and boolOpSPA boolExprSPA</td>
<td>boolExprXQu boolOpXQu boolExprXQu</td>
</tr>
<tr>
<td>exprSPA compSPA exprSPA</td>
<td>exprXQu compOpXQu exprXQu</td>
</tr>
<tr>
<td>REGEX(exprTSPA, exprPSPA)</td>
<td>fn:matches(exprTQu, exprPQu)</td>
</tr>
<tr>
<td>REGEX(exprTSPA, exprPSPA, flagSPA)</td>
<td>fn:matches(exprTQu, exprPQu, flagSPA)</td>
</tr>
</tbody>
</table>

| Table 27: Translation of Boolean expressions from SPARQL to XQuery |

6.2.2.3 Boolean Operators

<table>
<thead>
<tr>
<th>boolOPSPA :=</th>
<th>boolOPXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Table 28: Translation of Boolean operators from SQL to XQuery |

6.2.2.4 Comparison Operators

<table>
<thead>
<tr>
<th>compSPA :=</th>
<th>compXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(general)</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>!=</td>
<td>!=</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

| Table 29: Translation of comparison operators from SQL to XQuery |

6.2.3 Expressions

6.2.3.1 Atomic Expression

An atomic expression can either be a constant or a variable.

- varName is a name of a variable
- const is a constant value

<table>
<thead>
<tr>
<th>exprSPA :=</th>
<th>exprXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>[?$]varNameSPA</td>
<td>varNameXQu</td>
</tr>
</tbody>
</table>

| Table 30: Translation of atomic expressions from SPARQL to XQuery |
6.2.3.2 List of Variables

The following table implements a list of variables with arbitrary length. For the following definitions it holds:

- varName is a name of a variable

<table>
<thead>
<tr>
<th>varListSPA :=</th>
<th>varListXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>[?$]varNameSPA</td>
<td>varNameXQu</td>
</tr>
<tr>
<td>varListSPA , [?$]varNameSPA</td>
<td>varListSPA , varNameXQu</td>
</tr>
</tbody>
</table>

Table 31: Translation of list of variables from SPARQL to XQuery

6.2.3.3 Ordering the Result

The following table implements a list of variables with arbitrary length. For each variable the sort order may be ascending or descending. For the following definitions it holds:

- varName is a name of a variable

<table>
<thead>
<tr>
<th>orderListSPA :=</th>
<th>orderListXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>[?$]varNameSPA</td>
<td>varNameXQu</td>
</tr>
<tr>
<td>ASC([?$]varNameSPA)</td>
<td>varNameXQu</td>
</tr>
<tr>
<td>DESC([?$]varNameSPA)</td>
<td>varNameXQu descending</td>
</tr>
<tr>
<td>orderListSPA , [?$]varNameSPA</td>
<td>orderListXQu , varNameXQu</td>
</tr>
<tr>
<td>orderListSPA , ASC([?$]varNameSPA)</td>
<td>orderListXQu , varNameXQu</td>
</tr>
<tr>
<td>orderListSPA , DESC([?$]varNameSPA)</td>
<td>orderListXQu , varNameXQu descending</td>
</tr>
</tbody>
</table>

Table 32: Translation of list of expressions for order-by from SPARQL to XQuery

6.2.3.4 Limiting the Result

For the following definitions it holds:

- limitSPA is an integer value which gives the maximum number of results to display
- offsetSPA is an integer value which gives the result number to start with
- maxXQu is an integer value which gives the maximum nodes in the sequence of results $result which can be calculated by fn:count($result) in XQuery

<table>
<thead>
<tr>
<th>limitOffsetSPA :=</th>
<th>positionXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIMIT limitSPA</td>
<td>position() = (1 to limitSPA)</td>
</tr>
<tr>
<td>OFFSET offsetSPA</td>
<td>position() = (offsetSPA+1 to maxXQu)</td>
</tr>
<tr>
<td>LIMIT limitSPA</td>
<td>position() = (offsetSPA+1 to offsetSPA+limitSPA)</td>
</tr>
<tr>
<td>OFFSET offsetSPA</td>
<td></td>
</tr>
</tbody>
</table>

Table 33: Translation of list of expressions for order-by from SPARQL to XQuery
6.2.3.5 Namespace Definition

The following table implements a list of namespace definitions with arbitrary length. For the following definitions it holds:

- **NsUri** is a URI which identifies a namespace
- **prefixName** an arbitrary local alias for the namespace with a given URI

<table>
<thead>
<tr>
<th>nsListSPA</th>
<th>nsListXQu</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsListSPA</td>
<td>nsListXQu</td>
</tr>
<tr>
<td>PREFIX prefName: &lt;NsUri&gt;</td>
<td>declare namespace prefixName = &quot;NsUri&quot;</td>
</tr>
<tr>
<td>(empty)</td>
<td>(empty)</td>
</tr>
</tbody>
</table>

Table 34: Translation of namespace definition from SPARQL to XQuery

6.2.4 Query Forms

6.2.4.1 Select Query

The result of a select query is a list of combinations of values which are bound to the variables defined in varListSPA. If a variable is not bound in a certain combination, the value may be empty.

<table>
<thead>
<tr>
<th>resultSPA</th>
<th>resultXQu</th>
</tr>
</thead>
<tbody>
<tr>
<td>resultSPA</td>
<td>resultXQu</td>
</tr>
<tr>
<td>SELECT varListSPA WHERE { patternSPA } ( ORDER BY orderListSPA )? ( limitOffsetSPA )?</td>
<td>let $result := patternXQu ( order by orderListXQu )? return $result([positionXQu])?</td>
</tr>
</tbody>
</table>

Table 35: Translation of SELECT query from SPARQL to XQuery

6.2.4.2 Ask Query

An Ask query returns true if at least one possible solution for patternSPA can be found.

<table>
<thead>
<tr>
<th>resultSPA</th>
<th>resultXQu</th>
</tr>
</thead>
<tbody>
<tr>
<td>resultSPA</td>
<td>resultXQu</td>
</tr>
<tr>
<td>ASK { patternSPA }</td>
<td>let $result := patternXQu return &lt;result&gt; &lt;ask&gt; { fn:exists($result) } &lt;/ask&gt; &lt;/result&gt;</td>
</tr>
</tbody>
</table>

Table 36: Translation of ASK query from SPARQL to XQuery
7 Architecture

The xQL-to-XQuery software is written in C++ and can be operated by means of a simple command-line interface. The behaviour of the application can be influenced by of a number of parameters which is described in Appendix B.

In order to improve the usability of the system, a web-interface has been developed which enables the user to set all necessary parameters and enter the input queries by means of a standard web-browser. For the different use-cases, the input form is adjusted automatically such that only the relevant parameters are enabled.

Section 7.1 shows the system architecture and the environment of both the command-line tool and the web-interface. In Section 7.2 the focus is on the software architecture of the xQL-to-XQuery command-line tool.

7.1 System Architecture

The web-application is a CGI script written in Perl 5.6, which is running on an Apache web server. Perl has been chosen because of its capabilities with regular expressions and forwarding requests to the command line interface.

Users may enter queries in SQL or SPARQL into the input form of the web-interface. After having selected the necessary supplementary information like the language and the collection, the web application executes the xQL-to-XQuery command line tool and passes the data using the according parameter settings. The query itself is written to the file system and the filename passed to the translation tool, because a query usually may contain several lines and special characters.

If the transformation has been performed successfully, the corresponding XQuery code is returned as a result. Otherwise, an error message is given back which is marked by a defined prefix string that is known to the web-application. In this way, an error can be recognized and shown to the user. The generated XQuery code is given to the XQuery engine either via the command-line interface (Zorba) or by means of a HTTP-Request (Sausalito). For Sausalito, a special handler module has been written which is able to take the XQuery code as a parameter and use the XQuery \texttt{fn:eval{}} function to execute the code. The result is given back to the web-application in terms of XML data. If the execution of the XQuery code has been successful is, again, determined by means of regular expressions in the Perl program. The XML data is formatted in case of a successful translation and executed or an error message is shown otherwise.

Sausalito is able to store persistent data in XML collections which can be accessed and modified by XQuery functions which are available from a collection module. The collections may be initialized and reset to a default state by calling a \texttt{init()} function which clears all data and writes some examples to the database. This state may be modified by XQuery code containing functions from the collection module. Sausalito is able to keep persistent data and writes updates back to the file system regularly. Unlike Sausalito, Zorba discards all updates, because it does not write changes back to hard disk. As the Zorba command-line program is restarted for each query, the updates of the previous queries are not visible.
The web-application offers the possibility to execute SQL or SPARQL queries on a MySQL server or an ARQ SPARQL processor, respectively. The result of the native input can be shown to the user on the web-interface. In addition, a feature is available which is able to parse the result of the processed code and compare it automatically. Thus, the results can be verified for equivalence easily.

As suggested by the motivation in Section 1, there may be more client applications which use an API to simulate a SPARQL engine or a SQL server. The legacy code produced by the client is then forwarded to the xQL-to-XQuery tool and translated to XQuery. This is shown by the dashed lines in Figure 5. The result of the executed code is given back to the application in the correct format (e.g. as a table for SQL). Thus, the application code does not have to be modified.

### 7.2 Software Architecture

An overview of the software architecture of the xQL-to-XQuery application is shown in Figure 6 which visualizes the process of generating XQuery code out for a given input. The first step of the translation is analyzing and parsing the input which is implemented using Flex [11] and Bison [7]. After having passed several error checks and optimization steps the processed information is passed to the XQuery renderer as an object tree in order to generate the XQuery output.

The transformation is described in more detail on the following pages.
7.2.1 Lexer

In the first step of the translation process, the lexer performs the lexical analysis which converts the stream of input characters into a sequence of tokens. The behaviour of the lexer is defined by a set of rules making use of regular expressions which define the set of characters for each token.

7.2.2 Parser

The stream of tokens generated by the lexer is passed to the parser in order to perform the syntactic analysis which determines a relationship between the input tokens. A parser can decide which sentences are valid with respect to a given grammar. The relationship can be visualized by means of a parse tree. The following example shows a parse tree for a simple SQL SELECT query:

```
SELECT symbol, num FROM elements WHERE num > 100
```

Figure 7: Parse tree for SQL SELECT query
For this project, a LARL (Look-ahead LR) parser has been implemented which is based on a finite-state-automaton and therefore is more efficient than a more general GLR parser. An LR parser for context-free grammars reduces the input from left to right for calculating a Rightmost-derivation. The SQL, SPARQL, CSV and RDF parsers which have been written for this work, are defined by a LARL(1) grammar which only computes one single look-ahead. This grammar type is more efficient than a LARL(n) grammar with \( n > 1 \), but requires the introduction of some additional parser rules when implementing the EBNF grammar defined in a W3C standard like [25] for SPARQL.

For each parse-node an object is created which passes the information from bottom-up using transfer classes. In each parse node all information about the nodes below is available. The business logic is called for the parser root node which is implementing the Visitor Pattern.

### 7.2.3 Error Checking

To ensure a translation which is correct both syntactically and semantically, error checking has to take place at different stages of the translation process.

**Lexer**

The only error the lexer can throw is the “mystery character” problem. This error occurs, when a token of the input stream does not match any lexer rule.

**Parser**

The parser expects a certain relationship between the tokens in the input stream, which is defined by the parser rules. As the LARL parser uses a deterministic and non-ambiguous grammar, the type of the next possible and expected token is always well-defined. If a token type occurs, which does not match any grammar rule, a parser error is thrown. In order to track the problem, a location object is generated by the parser which contains the error position in terms of a character range in the input stream. By means of this information, the web-application can reconstruct which token has caused the error and mark the token with red color. In addition, a red arrow is added in front of the error. If the error is not caused by a certain token but a position instead, only an arrow is inserted. As for instance, this is the case for missing quotes or missing left parenthesis. The line number is calculated by counting the number of return characters (\(\backslash n\)).

Figure 8 shows an example for an error which is generated by a missing right parenthesis.
Variable Binding
For SQL code, an error is thrown if in a condition or a select clause an attribute is used that has neither been defined by a nested expression nor occurs in a table which is in scope. If the mapping file is disabled, schema check is disabled implicitly. In this case it must be possible for each attribute to be assigned to a table uniquely. From this it follows anonymous variables with disabled mapping may only be used if only one table is in scope. If this rule is violated, an error is thrown.

In SPARQL, no error can occur if a variable is used that is not assigned or bound. The value of a not bound variable is just left empty.

7.2.4 Optimizer
The optimizer improves the efficiency of the generated XQuery code. Optimizing the code makes sense for the translation of languages only, not for the generation of code from imported data. Two types of optimizers can be distinguished for this application.

The input optimizer is dependent on an input language like SQL and performs optimizations which are executed on the object tree directly. One example is the logical simplification of Boolean expressions like the reduction of tautologies. Yet, a lot of these optimizations can be performed by the XQuery engine as well. For this reason it has been decided to keep this optimizer rather simple and focus on the simplification steps which are not performed by the engine.

The output optimizer is integrated into the XQuery renderer and affects the code generation directly. One very important optimization is the upward moving of Boolean conditions for nested loops which is heavily used for the SPARQL translation.

This is demonstrated by the following example which returns all orders with a priority greater than 3 of the customer 'Marc'.

```xquery
for $customer in $customers
for $order in $orders
where $order/priority > 3 and $customer/name = 'Marc'
return
  <order>
    { $order/id, $order/priority }
  </order>
```

Code 27: Naïve XQuery code

```xquery
for $customer in $customers[name = 'Marc']
for $order in $orders[priority > 3]
return
  <order>
    { $order/id, $order/priority }
  </order>
```

Code 28: Optimized XQuery code
After the optimization, the condition \((\text{priority} > 3)\) has to be evaluated for the matching customer 'Marc' only in contrast to all combinations of customers and orders in the not optimized version. Experiments have shown that the Zorba engine does not perform this simplification which is probably due to the fact that the most general solution requires logical transformations. As this optimization has a huge impact on the transformation of SPARQL to XQuery and the Boolean conditions are quite simple (conjunction of all triples), this optimization is performed by the xQL-to-XQuery tool.

### 7.2.5 XQuery Code Generation

The XQuery renderer produces the XQuery code out of the data which has been processed by the previous steps. Whenever possible, the code generation is kept independent of the source language and is moved to an abstract class from which all language dependent XQuery renderers inherit.
8 Testing and Measurements

At a very early state of the development phase, an extension for the web application has been implemented which allows to define a number of queries in SQL and SPARQL which are given to the xQL-to-XQuery translation process on the one hand and executed natively on the SQL server or SPARQL engine on the other hand. The results are compared automatically and the execution time is measured. This allows for ensuring correctness and comparing performance regularly whenever the application is modified. Thus, the consequences of every change on the application code during the development can be made visible very quickly and efficiently. The early discovery of a problem reduces time spent on debugging and increased productivity during the work.

![Figure 9: Comparing xQL to XQuery translation to native execution](image)

This section is divided into 2 topics. The unit tests in 8.1 refer to ensuring correctness automatically whereas 8.2 measures and evaluates performance.

8.1 Unit Tests

The unit tests are supposed to represent the most frequent use cases which have been identified and completed continuously during the development phase. For SQL, most test cases are chosen to support the TPC-H benchmarks [17] and examples from a SQL reference [16] and a database textbook [12].

For SPARQL a different approach to choose the test cases is necessary, because most examples and references like [4] only reflect very basic examples. For this reason the best tutorial for SPARQL is the W3c standard [25] itself which explains all combinations of language constructs that are possible.

Both for SQL and SPARQL a separate configuration file with unit tests is stored for the web application. For each unit test there is a short description, the query string and the name of the collection it refers to. For SPARQL, optionally, there may be a list of named graphs that are used in the query.

When clicking on the “Auto Test” button, all unit tests for the current language (i.e. SQL or SPARQL) are executed. Each single unit test is executed both with the additional mapping
information and without it. A test may be skipped if a certain combination of test case and mapping is not possible and therefore disabled. For each skipped test case the reason is given.

As executing all unit tests takes a considerable amount of time, all results are cached in a database and executed again only if the timestamp of the SQL-to-XQuery binary changes or if the cache has been reset.

8.2 Benchmark

For the benchmarks a standard test suite has been chosen. The Berlin SPARQL Benchmark [5] provides both queries for SPARQL and SQL running on the same schema of the data source, which perfectly fits the needs for this thesis.

The tests have been executed for Berlin scaling factors from 10 to 100 only, because the Sausalito application server currently is not able to handle a larger collection size. The following table shows the number of tuples for different scaling factors.

<table>
<thead>
<tr>
<th>scaling</th>
<th>products</th>
<th>features</th>
<th>offers</th>
<th>vendors</th>
<th>producers</th>
<th>persons</th>
<th>reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>289</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>999</td>
<td>2000</td>
<td>1</td>
<td>3</td>
<td>54</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 37: Size of input data for different scaling factors

The data is generated in a way to grow approximately linearly with the scaling factor.
For the measurement of the execution time, each query is run 10 times for both the translation to XQuery and the native execution. The repetition lets the distribution of the result converge to a Gaussian Normal Distribution and makes the measurement more stable.

The ARQ SPARQL engine [3] has a constant additional execution time of about one second which is probably due to the fact that the Java virtual machine has to be initialized first. As for performance critical applications this overhead would probably be eliminated by keeping a running process of the engine in memory, an offset for each engine has been subtracted from the results shown in the next sections. The values have been determined by calculating the mean time of executing 50 queries which fetch one tuple from a data source that contains one entry only.

<table>
<thead>
<tr>
<th>Engine</th>
<th>Time / sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARQ SPARQL engine</td>
<td>1.0934934</td>
</tr>
<tr>
<td>MySQL server</td>
<td>0.00005088</td>
</tr>
<tr>
<td>XQuery processor</td>
<td>0.07859156</td>
</tr>
</tbody>
</table>

Table 38: Offsets for “heating period” of engines

### 8.2.1 SQL to XQuery

For the SQL measurements the Berlin Benchmark has been applied instead of TPC-H [17], because it can be used both for SQL and SPARQL. In addition, because of the limitation of maximum tuples caused by Sausalito, only a very small TPC-H maximum scaling factor 0.01 could be used. For very small scaling factors < 0.01 the generated TPC-H test data does not grow linearly with the scaling factor. Thus, the Berlin Benchmark seems to be the better choice for this work. In the thesis, only query 6 and 8 are presented, because they show the most important results. A more complete list of test cases can be selected on the web interface which is available at [34].

The time which is necessary for the translation from SQL to XQuery is independent of the data size and dependent on the length and complexity of the query only:

<table>
<thead>
<tr>
<th>Query</th>
<th>Time / sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin SQL query 6</td>
<td>0.011</td>
</tr>
<tr>
<td>Berlin SQL query 8</td>
<td>0.021</td>
</tr>
</tbody>
</table>

Table 39: Time for translating SQL queries to XQuery

**Performance of a simple SQL query**

Berlin SQL Query 6, which is shown in 14.1.1, returns the number and the label of all products which contain the string “manner” in their label. The condition is given by means of a regular expression.

Figure 11 shows the execution time of the SQL query on a MySQL server compared to the time of the generated XQuery executed on a Sausalito application server in a logarithmic scale.
The execution speed of the XQuery code on Sausalito is about 2 orders of magnitudes slower than the SQL code being natively executed on a MySQL server. In addition, the performance of the MySQL server is almost constant for the given scaling factors, whereas the performance of XQuery decreases remarkably for higher scaling factors. The time for translating the SQL input to XQuery is a constant value of 0.011 seconds, which is not shown in the figure.

**Performance of a SQL query joining two tables**
The following SQL query, which is number 8 from the Berlin benchmark shown in 14.1.2, performs a join of two tables. The query asks for reviews of a given product number fetching the name of the reviewer from the Person table.

Figure 12 reveals that the performance of joins in XQuery gets even worse than the simple query and decreases another order of magnitude compared to MySQL. The translation time of 0.022 seconds is negligible for this query compared to the execution time.

The measurements show that the execution time of the XQuery code increases very fast with the scaling factor. This is due to the fact that the join has to be implemented by means of an additional nested loop for each table in a join. Compared to a SQL database server, the performance is so bad, because the execution time is in $O(n \cdot m)$ with $n$ and $m$ being the size of the two tables to be joined. Sausalito has to execute the join as an inefficient nested loop join whereas the MySQL server probably uses additional information like indices or hash values to replace the nested loops with an index nested loop join or a hash join.
8.2.2 SPARQL to XQuery

The Berlin SPARQL Benchmark [5] is the most popular metric for the performance of SPARQL engines. The test suite contains a Java application for generating test data and a set of 12 SPARQL queries. For the presentation in this thesis, 4 queries have been selected, which evaluate the most important constructs of the SPARQL language.

Again, the time which is required for the translation from SPARQL to XQuery is independent of the data size and dependent on the length and complexity of the query only:

<table>
<thead>
<tr>
<th>Query</th>
<th>Time / sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin SPARQL query 1</td>
<td>0.013</td>
</tr>
<tr>
<td>Berlin SPARQL query 3</td>
<td>0.016</td>
</tr>
<tr>
<td>Berlin SPARQL query 4</td>
<td>0.016</td>
</tr>
<tr>
<td>Berlin SPARQL query 8</td>
<td>0.013</td>
</tr>
</tbody>
</table>

Table 40: Time for translating SPARQL queries to XQuery

Performance of a simple SPARQL query
The Berlin SPARQL Query 1 shown in 14.2.1 returns the ID and the label of products which have a certain type and two given product features. In addition, a filter condition restricts the results to have a numeric property with a value greater than 100.
The measurement results in Figure 13 show that the execution time of the generated XQuery code is significantly faster than executing the corresponding SPARQL input on ARQ, although the offset of about one second of ARQ has already been subtracted from the values above. For larger scaling factors and size of data this difference even becomes larger. Again, the constant time for translation of 0.013 seconds is neglected.

Obviously, the execution of a semantically equivalent simple query is much more efficient on Sausalito than on ARQ. The difference can not only result from the fact that Sausalito keeps all data in memory and therefore behaves like an in-memory database system, because the disregarded “heating period” could have been used by ARQ to cache the required data in memory to show a similar rate of increase in time like Sausalito. Therefore, the bad performance has to be caused by inefficient algorithms and data structures used for ARQ in order to fetch data from a single data source.

Performance of a SPARQL query with optional data
The query 3 from 14.2.2 fetches products which have a given type and feature and adds two restrictions to the numeric property value. In addition to the structure seen in query 1, an OPTIONAL section is added which binds the label of the current product to the variable ?testVar, whenever the product has a feature with number “46”. By means of a filter condition !bound(?testVar) a negotiation is achieved which actually retrieves all products without a feature number “46”.

In the translation to XQuery, the OPTIONAL section is implemented by means of a helper function xqlib:optional() which performs an action similar to a LEFT OUTER JOIN in SQL. This is done using two sub queries, which may lead to a large intermediate result which becomes ineffective for an increasing scaling factor. For this reason, as depicted in Figure 14, the XQuery version is faster than the SPARQL query for small values taking advantage of the fast execution of simple queries, but becomes slower for larger scaling factors due to increasing size of the intermediate result. Obviously, ARQ uses a more efficient algorithm here.
Performance of a SPARQL query with union of two data sources

A query which defines two sub-queries and performs the union of the result is given as query 4 in 14.2.3. Each sub-query selects products for a given type and two features, binds a variable to a numeric property, which is restricted by a filter condition. In the result, products are shown which satisfy any of the conditions defined for the two sub-queries.

As shown in Figure 15, again, the execution time of the generated XQuery code is faster than the native SPARQL code on ARQ, although the offset for a trivial query has already been subtracted from the numbers. Yet, performance of both versions decreases in the same way for larger scaling factors. The execution time of query 4 is considerable higher than query 1, although the results from the two sub-queries have to be concatenated only. The explanation for this strange behaviour is that query 1 asks for a single product by giving its
product number, whereas query 4 returns a much larger set of matching products which is approximately linear to the scaling factor.

**Performance of a SPARQL query which joins two data sources**

Query 8, which is given in 14.2.4, performs a join of two different classes of data. First, all reviews for a given product number are selected. A number of variables are bound to properties of reviews like “title”, “date” and “text”. The variable \(?\text{reviewer}\) is bound to the identity of the reviewer by adding a SPARQL triple with \(?\text{review}\) as a subject, the predicate “reviewer” and \(?\text{reviewer}\) as an object. The variable \(?\text{reviewer}\) can now be used to retrieve information from the other data source.

![Figure 16: Performance of a SPARQL query with join of two data sources](image)

As shown in Figure 16, the performance for combining two data sources is faster in XQuery with Sausalito than SPARQL being executed on ARQ. The execution time is in the same order of magnitude and increases approximately in the same way. Both Sausalito and ARQ have no additional information and data structures available, like indices or hashes, which could be used to improve the performance of the join, which leads to a nested loop in both languages.
9 Conclusion

9.1 Summary
In this thesis it has been shown that it is actually possible to map a (in theory arbitrary) SQL or SPARQL input to XQuery. In the most general case, additional information has to be provided to the system for the translation process by means of a mapping file.

As documented in Section 8.2 the performance of the XQuery code generated from SQL performs rather poorly compared to a MySQL [15] server. The performance of SPARQL being translated to XQuery is in the same order of magnitude than a native SPARQL processor like ARQ [3].

So far, no fundamental reason could be identified which prevents XQuery from reaching the same performance like a mature relational database server. Given that an optimized and stable XQuery application server is available it might be convenient for companies to store all business data in XML collections. New applications developed in XQuery access data directly whereas SQL or SPARQL queries from legacy applications are mapped to XQuery first. The main advantage of this scenario is that all applications can work on the same data.

9.2 Limitations
The SQL-92 standard is very long and complex. The BNF grammar alone is defined on more than 200 pages. For this reason, only a subset of SQL-92 could be implemented for this thesis. This subset has been chosen in order to support the most frequent use cases and benchmarks. The software architecture is chosen in way which makes it possible to add more language constructs to the system seamlessly.

For the translation from SQL to XQuery, more optimizations are possible. For example if there is a nested-loop and an inner \texttt{WHERE}-condition, experiments have shown that performance can be improved by splitting the \texttt{WHERE}-clause and moving those conditions into outer loops which depend only on the variables defined so far. Yet, a most general implementation requires complex logical transformation rules which is out of scope for the transformation step. Not all possible optimizations have been implemented, because they can be done by the XQuery engine in the same way.

9.3 Future Work
Version 0.9.8 of the Zorba XQuery processor and version 0.9.6 of the Sausalito XQuery application server does not provide support for typed data. As soon as the type system is implemented and available on the XQuery application server, the type information provided by RDF documents can be mapped to the data stored in XML collections during the translation process. This type information can be exploited when mapping SQL and SPARQL to XQuery and thus prevents type conflicts. With type information available, XML collections can be stored and queried more efficiently. As an example, for a Boolean typed value, only 1 bit of storage is required as opposed to 8 bits for a string character (or even more, if the string “false” is stored).
The most severe bottleneck that occurs when automatically generated XQuery code is executed is joining information from two datasets. Currently, this can be done with a nested loop join only which results in a runtime which is quadratic to the size of the collection. Both for SQL and SPARQL fast and efficient join algorithms are an important precondition for good performance. The runtime of the current system could therefore be improved by providing an index data structure which allows for more efficient join algorithms like an index nested loop join.

Two days before the end of this thesis, a new release of Zorba and Sausalito has been released. Unfortunately, the time is too late to study the new release as part of this work and examine the influence of the improvements on the benchmarks. With the new version of Zorba, integrity constraints have been introduced. This feature may be suitable to model SQL constraints with an XML database in order to ensure referential integrity and consistency of the stored data. The most important improvement is the support of indices which allows for a significant improvement of the performance of joins, which have been indentified as the main bottleneck by the benchmarks in 8.2. Adding integrity constraints and index support to the xQL-to-XQuery translation tool will provide a more complete alternative to a relational database and a much better performance of the XQuery code generated both from SQL and SPARQL.
10 Acknowledgement

I would like to thank Prof. Dr. Donald Kossmann for the possibility to work on this interesting topic in his group. The regularly scheduled meetings were important milestones on my project plan and a very efficient way to discuss questions about the system architecture.

The excellent support offered by the team of 28msec [1] was very helpful. Most of all, I am very grateful to Matthias Brantner, David Graf and Dennis Knochenwefel for their fast and competent answers to all my questions regarding the Sausalito application server [2]. For my thesis Dennis Knochenwefel kindly provided a special version of Sausalito which does not write updates back to harddisk automatically and thus simplified debugging and benchmarks.
11 Appendix A: Glossary

11.1 Terms

**Compiler**  A compiler is a software component that transforms source code from one language into another.

**Lexer / Scanner**  A lexer is a software component which produces a sequence of tokens out of an input stream of characters. This action is called lexical analysis.

**Parser**  The task of a parser is to determine a grammatical structure of a sequence of tokens. This action is called syntactic analysis.

11.2 Abbreviations

**GLR parser**  Generalized Left-to-Right Rightmost derivation parser

**GUI**  Graphical User Interface

**HTML**  Hypertext Markup Language

**HTTP**  Hypertext Transfer Protocol

**SPARQL**  SPARQL Protocol and RDF Query Language

**SQL**  Structured Query Language

**TCP/IP**  Transmission Control Protocol/Internet Protocol

**LARL parser**  Lookahead LR-parser

**LR parser**  Parser type which reads input from left to right

**URL**  Uniform Resource Locator
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
12 Appendix B: Manual

This appendix section explains how the xQL-to-XQuery tool can be used.

12.1 Command Line Utility

12.1.1 Parameters

The xQL-to-XQuery command line utility uses the following parameters:

<table>
<thead>
<tr>
<th>parameter name</th>
<th>value</th>
<th>description</th>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>name of collection</td>
<td>storage of data</td>
<td>4</td>
</tr>
<tr>
<td>debug</td>
<td>boolean (true</td>
<td>false]</td>
<td>enable or disable debugging mode</td>
</tr>
<tr>
<td>file</td>
<td>filename</td>
<td>input filename</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>attribute</td>
<td>format of imported data</td>
<td></td>
</tr>
<tr>
<td>graph_files</td>
<td>comma separated list of filenames</td>
<td>RDF files for named graphs</td>
<td>5</td>
</tr>
<tr>
<td>help</td>
<td>boolean (true</td>
<td>false)</td>
<td>show a help screen</td>
</tr>
<tr>
<td>import_mapping</td>
<td>filename</td>
<td>XML mapping for imported data</td>
<td></td>
</tr>
<tr>
<td>output_format</td>
<td>(sparql-xml</td>
<td>default)</td>
<td>format of query output</td>
</tr>
<tr>
<td>schema</td>
<td>filename</td>
<td>XML schema for collections</td>
<td>6</td>
</tr>
<tr>
<td>storage</td>
<td>(filesystem</td>
<td>collection)</td>
<td>storage type</td>
</tr>
<tr>
<td>timer</td>
<td>boolean (true</td>
<td>false)</td>
<td>enable measurement of execution time</td>
</tr>
<tr>
<td>type</td>
<td>(sql</td>
<td>sparql</td>
<td>csv</td>
</tr>
</tbody>
</table>

Table 41: Command line parameters
12.2 Web Interface

12.2.1 GUI Elements

The web-interface is shown exemplarily for the SPARQL language.

![Screenshot input form of the web-interface](image)

1. selection of the input language
2. input of the expression in the language chosen in (1)
3. selection of a pre-defined example dependent on the language in (1)
4. selection of the collection
5. selection of named graphs to be loaded
6. add additional information for the translation (mapping file)
7. selection of the XQuery engine
8. show additional information for debugging
9. start translation of expression entered in (2)
10. execute expression (2) natively in language (1)
11. do translation and compare result with native execution
12. run unit tests
13. reset database to default values

Table 42: Description of GUI elements
13 Appendix C: Implementation

The function xqllib:matches takes two nodes and a list of variables as an input and returns true if both nodes have identical values for all variables.

```
declare function xqllib:matches ($result1 as node(), $result2 as node(), $variables as xs:string*) as xs:Boolean {
    if(not(fn:empty($variables))) then
        let $result :=
            for $var in $variables
                where not($result1/var/@name = $var and $result2/var/@name and data($result1/var[@name = $var]) = data($result2/var[@name = $var]))
            return fn:true()
        return not(fn:exists($result))
    else fn:true()
};
```

Code 29: XQuery function which checks if two graphs match

The function xqllib:and takes two list of nodes and a list of variables as an input and returns the union of each pair of nodes which have identical values for all variables. This operation corresponds to an INNER JOIN in SQL.

```
declare function xqllib:and ($result1 as node()*, $result2 as node()*, $variables as xs:string*) as node()* {
    for $element1 in $result1
    for $element2 in $result2
        where xqllib:matches($element1, $element2, $variables)
        return
            <result>{ xqllib:distinct-deep(($element1/*, $element2/*)) }</result>
};
```

Code 30: XQuery function which returns conjunction of two graphs
The function `xqllib:optional` takes two lists of nodes and a list of variables as an input and returns the union of each pair of nodes which have identical values for all variables and the left node only if there is no match. This operation corresponds to a LEFT OUTER JOIN in SQL.

```xml
declare function xqllib:optional ($result1 as node()* , $result2 as node() *, $variables as xs:string*) as node() * {
  for $element1 in $result1
  let $join :=
    for $element2 in $result2
      where xqllib:matches($element1, $element2, $variables)
      return
        <result>
        {
          xqllib:distinct-deep(($element1/*, $element2/*))
        }
      </result>
      if($join) then
        $join
      else
        <result>{ $element1/* }</result>
  };
}
```

**Code 31: XQuery function which adds the information of one graph to another**

The next function implements a lower-case for order-by. It does not return an error (like `fn:lower-case()`) when the input value is a number.

```xml
declare function xqllib:ignoreCase ($input)
{
  if(string(number($input)) = 'NaN') then
    fn:lower-case($input)
  else
    $input
};
```

**Code 32: XQuery function which ignores case without returning an error for numbers**
14 Appendix D: Benchmarks

In this appendix session the queries are listed which are executed to evaluate the execution time in benchmark tests for SQL and SPARQL shown in Section 8.2. The queries are taken from the Berlin Benchmark [5].

14.1 SQL

14.1.1 Berlin SQL Query 6

Find products having a label that contains a specific string.

```
SELECT nr, label
FROM bsbm_product
WHERE label like '%manner%';
```

Code 33: Berlin SQL Query 6, SQL input

```
let $doc_bsbm := fn:collection("bsbm")
for $bsbm_product in $doc_bsbm[name(.) = "Product"]
where fn:matches($bsbm_product/label, "manner", "i")
return
  <result>
    <nr>{data($bsbm_product/nr)}</nr>
    <label>{data($bsbm_product/label)}</label>
  </result>
```

Code 34: Berlin SQL Query 6, generated XQuery code

14.1.2 Berlin SQL Query 8

Get recent reviews for a given product.

```
SELECT r.title, r.txt, r.reviewdate, p.nr, p.name
FROM bsbm_review r, bsbm_person p
WHERE r.reviewfor=1 AND r.publisher=p.nr
ORDER BY r.reviewdate desc
LIMIT 20;
```

Code 35: Berlin SQL Query 8, SQL input
let $doc_bsbm := fn:collection("bsbm")
let $result :=
  for $r in $doc_bsbm[name(.) = "Review"]
  for $p in $doc_bsbm[name(.) = "Person"]
  order by xqlib:ignoreCase($r/reviewDate) descending
  where (($r/reviewfor = 1) and ($r/publisher = $p/nr))
  return
    <result>
      <nr>{data($p/nr)}</nr>
      <name>{data($p/name)}</name>
      <title>{data($r/title)}</title>
      <txt>{data($r/text)}</txt>
      <reviewdate>{data($r/reviewDate)}</reviewdate>
    </result>
return
$result[position() = (1 to 20)]

Code 36: Berlin SQL Query 8, generated XQuery code
14.2 SPARQL

14.2.1 Berlin SPARQL Query 1

Find products for a given set of generic features.

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/.../instances#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/.../vocabulary#>

SELECT DISTINCT ?product ?label
WHERE {
    ?product bsbm:type "5" .
    FILTER (?value1 > 100)
}
ORDER BY ?label
LIMIT 10;
```

**Code 37: Berlin SPARQL Query 1, SPARQL input**

```
declare namespace bsbm = "http://www4.wiwiss.fu-berlin/.../vocabulary";
declare namespace bsbm-inst = "http://www4.wiwiss.fu-berlin/.../instances";
let $doc_bsbm := fn:collection("bsbm")
let $result :=
    let $GRAPH_0 :=
        for $node_product in $doc_bsbm[@rdf:ID and ProductFeature = "151"
            and ProductFeature = "150" and type = "5"]
        let $value_product := xqllib:getSubj("bsbm",$node_product)
        for $value_label in xqllib:getData("bsbm",$node_product/label)
        for $value_value1 in
            xqllib:getData("bsbm",$node_product/ProductPropertyNumeric)
            where fn:exists($value_label) and fn:exists($node_product)
            and fn:exists($value_value1)
        return
            <result>
                <var name="label">{$value_label}</var>
                <var name="product">{$value_product}</var>
                <var name="value1">{$value_value1}</var>
            </result>
    let $GRAPH_1 := $GRAPH_0[(data(var[@name="value1"]) > 100)]
    for $node in $GRAPH_1
        order by $node/var[@name="label"]
    return xqllib:formatOutput($node,("product","label"))
    return xqllib:distinct-deep($result)[position() = (1 to 10)]
```

**Code 38: Berlin SPARQL Query 1, generated XQuery code**
### 14.2.2 Berlin SPARQL Query 3

Find products for a given set of generic features and not having one given feature.

```sparql
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/.../instances#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/.../vocabulary#>

SELECT ?product ?label
WHERE {
  ?product bsbm:type "1" .
  FILTER ( ?p1 > 483 )
  FILTER (?p3 < 348 )
  OPTIONAL {
    ?product bsbm:label ?testVar
  }
  FILTER (!bound(?testVar))
}
ORDER BY ?label
LIMIT 10;
```

**Code 39: Berlin SPARQL Query 3, SPARQL input**

The translation of this SPARQL input to XQuery can be found at [34]. Because of the length of the generated code, the output is not shown here.
14.2.3 Berlin SPARQL Query 4

Find products matching two different set of features.

```sparql
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/.../instances#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/.../vocabulary#>

SELECT ?product ?label
WHERE
{
  {
    ?product bsbm:type "1" .
    FILTER ( ?p1 > 100 )
  }
  UNION
  {
    ?product bsbm:type "8" .
    ?product bsbm:ProductFeature "1" .
    FILTER ( ?p2 > 500 )
  }
}
ORDER BY ?label
LIMIT 10;
```

Code 40: Berlin SPARQL Query 4, SPARQL input

The translation of this SPARQL input to XQuery can be found at [34]. Because of the length of the generated code, the output is not shown here.
14.2.4 Berlin SPARQL Query 8

Get recent reviews for a given product. See SQL Query 8 [14.1.2].

```sparql
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary#>

SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName
WHERE
{
    ?review bsbm:reviewfor "1" .
    ?review bsbm:text ?text .
    ?review bsbm:reviewDate ?reviewDate .
    ?reviewer bsbm:name ?reviewerName .
}
ORDER BY DESC(?reviewDate)
LIMIT 20;
```

Code 41: Berlin SPARQL Query 8, SPARQL input

The translation of this SPARQL input to XQuery can be found at [34]. Because of the length of the generated code, the output is not shown here.
Bibliography


[34] Martin Kaufmann. Web-interface for the SPARQL and SQL to XQuery translator. http://www.xql2xquery.org/ Website is protected. Please enter username “ethz” and password “xquery”.