FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Implementation and Optimisation of Queries in XSPARQL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Information & Knowledge Management

eingereicht von

Stefan Bischof
Matrikelnummer 0327033

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr. Thomas Eiter
Mitwirkung: Dipl.-Ing. Thomas Krennwallner
Externe Mitwirkung: Dipl.-Ing. Dr. Axel Polleres

Wien, 2.11.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung

Stefan Bischof, Wolfgang-Schmälzl-Gasse 5/34, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. November 2010

Kurzfassung

XSPARQL ist eine Sprache zur Transformation von Daten zwischen XML und RDF. XML ist ein weit verbreitetes Format zum Austausch von Daten. RDF ist ein Datenformat basierend auf gerichteten Graphen, das primär zur Repräsentation von Daten im Semantic Web verwendet wird. XSPARQL kombiniert die Stärken der beiden zugehörigen Anfragesprachen XQuery für XML und SPARQL für RDF. In dieser Diplomarbeit präsentieren wir zwei Verbesserungen der XSPARQL-Sprache, die Constructed Dataset und Dataset Scoping genannt werden, die XDEP Dependent Join Optimierung sowie eine neue Implementierung von XSPARQL. Constructed Dataset erlaubt das Erstellen und Abfragen temporärer RDF-Graphen. Durch Dataset Scoping können unerwartete Ergebnisse, die beim Auswerten einer komplexen XSPARQL-Anfrage mit verschachtelten SPARQL-Anfrageteilen auftreten können, vermieden werden. Die XSPARQL-Implementierung schreibt eine XSPARQL-Anfrage in eine XQuery-Anfrage um, die, zur Verarbeitung von RDF Daten, eingeschobene Aufrufe einer SPARQL-Engine enthält. Die resultierende Anfrage wird dann von einem XQuery-Prozessor und einer SPARQL-Engine gemeinsam ausgewertet. Die Dependent Join Optimierung XDEP zielt auf eine Reduktion der Auswertungsdauer für Anfragen ab, die eingebettete SPARQL-Anfrageteile wiederholt auswerten müssen. XDEP minimiert die Anzahl von Interaktionen zwischen dem XQuery-Prozessor und der SPARQL-Engine, indem ähnliche SPARQL-Anfragen zusammengefasst und das Auswählen der relevanten Daten dem XQuery-Prozessor überlassen wird. Anhand einer adaptierten Version des XQuery-Benchmarks XMark haben wir eine experimentelle Evaluation unseres Ansatzes durchgeführt. Wir werden zeigen, dass die XDEP-Optimierung die Auswertungsdauer von allen kompatiblen Anfragen reduzieren konnte. Durch die Optimierung konnten wir bestimmte Anfragen um zwei Größenordnungen schneller auswerten als in der unoptimierten Version.

Abstract

XSPARQL is a language for transforming data between XML and RDF. XML is a widely used format for data exchange. RDF is a data format based on directed graphs, primarily used to represent Semantic Web data. XSPARQL is built by combining the strengths of the two corresponding query languages XQuery for XML, and SPARQL for RDF. In this thesis we present two XSPARQL enhancements called Constructed Dataset and Dataset Scoping, the XDEP dependent join optimisation, and a new XSPARQL implementation. Constructed Dataset allows to create and query intermediary RDF graphs. The Dataset Scoping enhancement provides an optional fix for unintended results which may occur when evaluating complex XSPARQL queries containing nested SPARQL query parts. The XSPARQL implementation works by first rewriting an XSPARQL query to XQuery expressions containing interleaved calls to a SPARQL engine for processing RDF data. The resulting query is then evaluated by standard XQuery and SPARQL engines. The dependent join optimisation XDEP is designed to reduce query evaluation time for queries demanding repeated evaluation of embedded SPARQL query parts. XDEP minimises the number of interactions between the XQuery and SPARQL engines by bundling similar queries and let the XQuery engine select relevant data on its own. We did an experimental evaluation of our approach using an adapted version of the XQuery benchmark suite XMark. We will show that the XDEP optimisation reduces the evaluation time of all compatible benchmark queries. Using this optimisation we could evaluate certain XSPARQL queries by two orders of magnitude faster than with unoptimised XSPARQL.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

Finishing a thesis and a whole computer science study takes a lot of time and effort. I am very grateful that I had many people around me who helped me achieving this goal.

For the work of this master's thesis I had the pleasure to do an internship at the Digital Enterprise Research Institute (DERI) in Galway, Ireland. I want to thank Prof. Thomas Eiter, my supervisor at the Vienna University of Technology (TUW), and Axel Polleres, my supervisor at DERI, for always encouraging me and make this inter-university thesis work, about a very interesting topic, possible. Thanks to Nuno Lopes, my colleague at DERI, for answering tons of questions during my time in Ireland and later. Thanks to Thomas Krennwallner TUW for his help with university bureaucracy and for creating the epilog poster. Thanks to all four of you, for proofreading, many insights and feedback—your work helped me greatly.

Special thanks go to my girlfriend Herlinde, for being there for me all the time and helping me in many ways to work on, write, and finally finish this thesis. Thanks to Lea Zalto for distracting and motivational break conversations during the last few months of work at the library. Of course I want to thank all my fellow students and friends for the great time during all the years of study. Finally I want to thank my parents Herta and Josef for their continuous support, for their believe in me and for making my studies initially possible.

This work has been partially funded by a thesis scholarship of the Faculty of Informatics at the TUW and my internship in DERI has been funded by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

Chapter 1

Introduction

Nowadays lots of data are stored or transferred in the XML [Bray et al., 2008] format. The Semantic Web uses another format, the Resource Description Framework (RDF) [Klyne and Carroll, 2004]. As the use of both, XML and RDF, increases, the need for transforming data between the two formats grows similarly. Although the default serialisation syntax for RDF is an XML language, transforming data remains a complicated task because of fundamental differences between the two data formats and syntactical ambiguities of RDF/XML.

XSPARQL [Polleres et al., 2009] integrates the two query languages for XML and RDF, XQuery and SPARQL, in order to remedy theses drawbacks. XSPARQL provides an intuitive and concise syntax to simplify data transformation and therefore brings the worlds of XML and RDF closer together. Nevertheless XSPARQL shows problems when evaluating complex queries: Query evaluation results can be intuitively wrong and query evaluation times can grow fast with increasing data source size.

The main contributions of this thesis are the following:

- A new semantics for the XSPARQL transformation language;
- The new features Constructed Dataset and Dataset Scoping to make query authoring easier;
- A new implementation of the XSPARQL rewriter used in a similar architecture as the former implementation;
- The dependent join optimisation Xdep to improve query evaluation time of complex XSPARQL queries;
- A practical performance evaluation comparing XQuery evaluation times with XSPARQL evaluation times and unoptimised XSPARQL with Xdep optimised XSPARQL.

The goal of the present thesis was to further develop XSPARQL and to build a stable and flexible implementation of the language. To achieve this goal standard compiler construction techniques, such as lexer and parser generators, are used. Two extensions to XSPARQL are introduced. The First, *Dataset Scoping*, fixes a known issue of nested queries returning unexpected results when certain variables are bound to blank nodes. The second extension, *Constructed Datasets*, facilitates new use cases by allowing one to create intermediary RDF graphs and to query these graphs in the same query. Furthermore we will introduce an approach for optimising XSPARQL query evaluation. Eventually the effectiveness of our optimisations is demonstrated in an experimental evaluation.

The following sections present the big picture of this thesis. We will give an overview of the XML and RDF data formats, their corresponding query languages and the XSPARQL transformation language. Furthermore we will discuss issues of the current XSPARQL specification which we will address as the main part of this thesis.

1.1 Data on the Web

XML The most important format to transfer data over the Internet and other networks of the last years is surely XML. One example use of XML are SOAP web services [Mitra and Lafon, 2007], where all parts of the communication depend heavily on XML. XHTML [Pemberton, 2002], another example, is a new definition of the Hypertext Markup Language (HTML) in XML terms. The *Extensible Markup Language* (XML) [Bray et al., 2008] is a specification language which enables us to create domain specific markup languages. XML implements the semi-structured data model [Abiteboul, 1997, Buneman, 1997]. A semi-structured data format such as XML can be used with or without a schema. It is therefore a flexible data model capable of representing loosely structured documents (e. g. XHTML web pages) and highly structured data (e. g. export of data of relational databases).

Example 1.1. Listing 1.1 shows a simple XML file describing three persons and their relations to each other [Polleres et al., 2009].

XML allows only one single *root element*. In Listing 1.1 the root element is `relations`. The three *child elements* represent three persons. The names of the individuals are given as *attributes*. The order of elements in a document (*document order*) is relevant, but not that of attributes for an element. Persons are alphabetically ordered in our example. Each person can know another person (in this example such a relation is unidirectional), represented as a child element containing the name of that second person as *text* node.

```
<relations>
  <person name="Alice">
    <knows>Bob</knows>
    <knows>Charles</knows>
  </person>
  <person name="Bob">
    <knows>Charles</knows>
  </person>
  <person name="Charles"/>
</relations>
```

Listing 1.1: XML example data

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/" xmlns:rdf="http://www.
    w3.org/1999/02/22-rdf-syntax-ns#">
  <foaf:Person rdf:about="alice/me">
    <foaf:knows>
      <foaf:Person foaf:name="Charles"/>
    </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

Listing 1.2: Example for XML namespaces

An important concept used in XML are *XML Namespaces* [Layman et al., 1999]. They allow XML to uniquely identify elements and therefore to distinguish for example between elements for a *drawer* being part of a desk and a *drawer* being a person who draws (signs) a cheque, by using a globally unique identifier. In XML *Uniform Resource Identifiers* (URI) [Berners-Lee et al., 1998] or *Internationalized Resource Identifiers* (IRI) [Duerst and Signard, 2005] are used as such identifiers. An abbreviated form of a URI is a *qualified name* (QName). Qualified names are built by appending a colon and the local part after a namespace prefix. A namespace prefix is declared in any enclosing element (like the root element). When defining an element, the URI `http://xmlns.com/foaf/0.1/name` can be written as the QName `foaf:name`, given that the namespace `http://xmlns.com/foaf/0.1/` was assigned to the prefix `foaf` earlier.

XML namespaces

The root element `rdf` of the XML document in Listing 1.2 declares two different namespaces: The RDF namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#` is bound to the `rdf` prefix, while the FOAF namespace `http://xmlns.com/foaf/0.1/` is bound to the `foaf` prefix. Thus every element starting with the `rdf` prefix is associated with the RDF namespace, likewise the `foaf` prefix associates element and attribute names with the FOAF namespace.

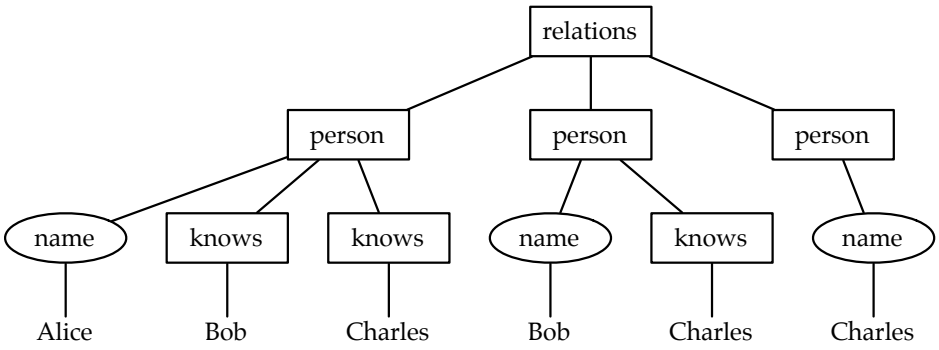

Figure 1.1: Example XML data

Schema language

XML is called *semistructured*, meaning a schema is optional and unlike relational database management systems, a schema is not needed to access or process XML data. The W3C Recommendation *XML Schema* [Walmsley and Fallside, 2004] provides a complex type system together with key and cardinality constraints and thus gives fine grained control over valid XML document instances. XML Schema contains 44 pre-defined simple types for numbers, strings, dates, durations and more as well as the capability to create user defined types.

Data models

Over time multiple *data models* have been created for XML. They hide syntactic issues like encoding or entity declarations from XML processing applications and provide a formal way to look at XML documents. The XML Infoset [Cowan and Tobin, 2004] was the first XML data model providing an abstract view of an XML document as a tree. The XQuery data model (XDM) [Fernández et al., 2007] extends the Infoset and it is the data model that XQuery (see Section 2.3) shares with XPath 2.0 [Berglund et al., 2007] and XSLT 2.0 [Kay, 2007]. It includes support for sequences of nodes (as the result of an XPath expression), XML Schema typing and ordered heterogeneous sequences. XQuery (as well as XPath and XSLT) takes XML documents under XDM as input and returns a value in XDM again.

When a XML document is interpreted under such a data model it can be represented as a tree.

Example 1.2. See Figure 1.1 for the tree representation of the XML document in Listing 1.1. Elements are represented as rectangles, attributes as circles and text without borders.

RDF

On the other hand more and more services offer access to data via the Semantic Web's *Resource Description Framework* (RDF) [Manola and Miller, 2004]. More and more organisations make data available in RDF as Linked

Data [Berners-Lee, 2006, Linked Data] using RDF. RDF is a way to describe resources, which can be documents, but also real world objects or abstract concepts; basically anything which can be labelled with an URI. RDF describes these resources and the relations between them by a data model based on directed graphs, i. e., resources are represented as nodes with directed links, called predicates, pointing from a source, called subject, to a target, called object.

1.2 Querying XML and RDF Data

XQuery [Boag et al., 2007] is the W3C Recommendation for an XML query language. It is specified together with XSLT 2.0 and XPath 2.0 [Boag et al., 2007, Melton and Muralidhar, 2007, Kay, 2007, Fernández et al., 2007, Malhotra et al., 2007, Draper et al., 2007, Fernández et al., 2007]. XQuery can be used to query XML documents or XML databases and extract, transform and aggregate XML data.

According to the introduction of the XQuery specification [Boag et al., 2007], there are two important design aspects: XQuery is a strongly typed and functional language. XQuery is a *functional* language, which means that the basic constructs are *expressions* (in contrast to *statements* like in imperative or object oriented languages). An XQuery query is therefore a list of possibly nested expressions. XQuery is a *strongly-typed* language, that operates on the XPath/XQuery data model (XDM) [Fernández et al., 2007] which uses XML Schema for data types. With optional *static typing* the types of expressions are determined and checked after parsing but before query evaluation, which allows static error detection. During query evaluation, when not only the query, but also the data is known, *dynamic type checking* is performed. As long as only standard extensions are used, XQuery is free of side effects which makes development of optimisations easier. An important aspect that XQuery shares with XSLT 2.0 and XPath 2.0 is a library of built-in functions [Malhotra et al., 2007] for manipulating XML and text content. The standard data type in XQuery is the sequence: Most functions take sequences as arguments, and eventually return sequences. Single values are interpreted as sequences of the length one.

The SPARQL query language is designed for querying different RDF data sources and is, like XQuery, also a W3C Recommendation [Prud'hommeaux and Seaborne, 2008]. It features a powerful graph pattern matching facility. Graph patterns are a way to match and extract parts of RDF graphs. SPARQL can return the result in several formats: In a row based format such as relational database systems, as another RDF graph again or as a binary "yes/no" answer. Similar to query languages for other data formats SPARQL

SPARQL

provides slicing and ordering of results.

Limitations of SPARQL

While being useful for querying RDF data, SPARQL has some limitations making extensions or complex queries hard. First of all, the built-in functions are available for FILTER expressions only, and there exists no mechanism to define other functions. Whenever named graphs are used, they have to be declared statically. This is a problem if a graph's IRI is stored in the RDF graph itself. In that case firstly the IRI of this graph has to be obtained and after that the "real" query can be built manually. Unlike SQL it is not possible to formulate subqueries or aggregates in SPARQL. Such features would have to be implemented in whatever context SPARQL is called (e. g. some generic programming or scripting language). Although the upcoming SPARQL 1.1 [Harris and Seaborne, 2010] is addressing subqueries and aggregates.

1.3 Transforming Data between XML and RDF

It is important for the use cases of *data integration* and for *interoperability* of tools and whole systems, to have a reliable way to transform data between or within different data formats. To make such transformations manageable and maintainable, it is desirable to use a tool which is not only able to perform this transformation in a robust and reliable manner, but which makes adaptions of the concrete transformation process easily achievable. In this section we discuss ways of transforming data between XML and RDF.

One way to accomplish the tasks of data integration is to use a generic programming or scripting language like Java, but for most transformations this approach is more complicated, longer and hence less maintainable than a tailored approach provided by query languages, some of which we discuss briefly in the following.

XML to XML The standard way of transforming data from one XML format to another XML format is to use XSLT [Kay, 2007] or XQuery [Boag et al., 2007] depending on the specific needs. XSLT (XSL Transformations) is the older language of the two, intended exactly for this use case: transforming XML data to XML (other output formats such as text or binary are also possible). XSL stands for *The Extensible Stylesheet Language* and consists of XSLT, XPath (also used in XQuery) and XSL Formatting Objects (XSL-FO). XQuery is slightly more declarative than the template based XSLT language but serves the same purpose. We will describe XQuery in more detail in Section 2.3.

RDF to RDF SPARQL can be used for converting RDF data again to RDF. But SPARQL provides no functions to manipulate text or numbers during that

process. The simple syntax makes complex queries impossible to express.

XML to RDF Since there is also an XML serialisation of RDF called RDF/XML, it is possible to use XSLT or XQuery to transform data from some XML format to RDF/XML. Nevertheless this approach is tedious because RDF is used the represent graphs and XML is limited to trees.

RDF to XML One way to go would be to use XSLT or XQuery again. Apparently that is only possible if the data is available in RDF/XML. If the data is given in some other RDF serialisation format, it first has to be transformed to RDF/XML. The biggest problem with RDF/XML as data source is the great syntactical variety one RDF graph can be represented in using XML. This variety makes it hard to come up with a versatile and working solution. Even if such a solution is found it is all but concise and poorly maintainable.

An alternative would be to first create a SPARQL query and then apply a XSLT or XQuery transformation on the SPARQLXML result set. Using this approach one might have to create several SPARQL queries and combine the results of all those queries in a single XQuery query. This approach is error prone, requires a lot of testing and distributes the transformation process over several queries, thus reducing maintainability of the data transformation process.

Akhtar et al. [2007] give a more detailed discussion of the problems of these transformations. In summary transforming data between XML and RDF has serious issues. With XSPARQL we provide an integrated query and transformation language, capable of simplifying these transformations.

1.4 The XSPARQL Transformation Language

The XSPARQL transformation language combines the strengths of XQuery and SPARQL to make data transformation between XML and RDF simpler. By merging these two query languages, XSPARQL can be seen as feature enabler for both base languages: It brings SPARQL's graph pattern matching facility to XQuery, thus allowing to access and process RDF data on a data model level, i. e., independent of the concrete RDF serialisation syntax. XSPARQL brings the large XQuery function library to SPARQL, making up for the fact that SPARQL lacks even basic functionality such as simple string manipulation or value aggregation. Furthermore it allows one to formulate subqueries in a syntax very similar to plain SPARQL, thus making more complex transformations and queries possible.

Besides, XSPARQL can be used as a SPARQL scripting framework making queries possible which can not be achieved that way in SPARQL alone for

```
1 construct {
2    $person :name { fn:concat($firstname, " ", $lastname) } .
3 }
4 from <relations.rdf>
5 where {
6    $person :firstname $firstname .
7    $person :lastname  $lastname  .
8 }
```

Listing 1.3: Simple XSPARQL string manipulation

RDF to RDF transformations. Since XSPARQL provides access to dynamically specified RDF data sources, it allows querying a dynamic set of named graphs (e. g., merge multiple RDF graphs in a single one as specified by a master RDF graph). SPARQL allows no extension by using functions. By allowing this via the XQuery extension mechanisms, it becomes possible in XSPARQL to return computed values or aggregated values. By using implementation specific XQuery functions, as found in most implementations, XSPARQL provides access to specific parts of XML chunks stored in RDF graphs.

Nevertheless the main use case for XSPARQL is data transformation and data integration. XSPARQL can be used to access both XML and RDF data and to emit data in the same two formats. XSPARQL is suited to make Semantic Web clients work together with XML based web services.

Listing 1.3 shows a simple XSPARQL query very similar to SPARQL. The query concatenates the first and last name of a person to a single string containing both. SPARQL can not provide this kind of transformation because of its lack of even simple string manipulation functions such as string concatenation. By using the XQuery function library, XSPARQL provides a simple and intuitive solution for this problem.

Issues of XSPARQL Nevertheless XSPARQL reveals several issues when working with complex queries featuring nested SPARQL parts. Even when processing only medium sized RDF graphs, XSPARQL query evaluation performance decreases fast. This performance decrease makes XSPARQL for several use cases practically useless.

While common relational database systems (DBMS) allow the user to create and query temporary tables, and XQuery too, allows one to create and query temporary XML trees, SPARQL lacks a measure to create and query temporary RDF graphs. XSPARQL inherits this feature lack from SPARQL and thus rules certain use cases out.

Another problem are unintended results which may arise when writing XSPARQL queries containing nested SPARQL parts. Depending on the query

and the concrete data source, XSPARQL can show unexpected behaviour. The author of such a query may comes to the conclusion that a variable "forgot its value", i. e., a variable occurs to be free although a value was assigned to it earlier. To make it worse this behaviour seems even inconsistent over one single data source.

1.5 Other Work Related to XSPARQL

Most of the ongoing work in the domain of XML-RDF data integration can be assigned to one of two classes: Data Translation or Query Language Integration.

Data Translation These approaches integrate XML and RDF by translating the data directly to a format suited better for further processing. The TriX [Carroll and Stickler, 2004] format is an alternative XML syntax for RDF data. It allows defining syntactic extensions and macros by using XSLT. Based on the XML Schema definition Gloze [Battle, 2006] aims at interpreting XML data under the RDF data model by providing a custom mapping between specific XML and RDF data. By translating XML to RDF and annotating the resulting RDF data with meta data needed to evaluate XPath expressions Droop et al. [2007] evaluate XPath expressions by evaluating it in the form of an converted SPARQL query. By converting concrete XML Schemas to specified ontologies Deursen et al. [2008] translate schema conformant XML data to RDF of the specified target ontology.

Query Language Integration As XSPARQL these approaches aim at integrating languages made for processing XML or RDF data. RDF Twig [Walsh, 2003] encoding (sub)trees occurring in RDF to abbrrdf/xml by XSLT using an extension function. These subtrees are then easily processable in plain XSLT. [Diego et al., 2008] extends XSLT by additional constructs allowing to query SPARQL endpoints while processing the results given in an XML format. SPARQL2XQuery [Bikakis et al., 2009] translates a SPARQL query to XQuery by an OWL to XML Schema mapping. [Groppe et al., 2008] follows a combined approach. First RDF data is translated in a custom XML format, then, on top of this data, queries of a new language, embedding SPARQL in XSLT/XQuery, are evaluated.

1.6 Thesis Structure

In Chapter 2 we describe some topics needed to understand the following chapters. First we present an overview over different data formats, particu-

larly XML and RDF, and corresponding query languages XQuery and SPARQL. Additionally the original XSPARQL language is introduced. Next in Chapter 3 we present our extensions to XSPARQL. These two extensions are embedded in the formal XSPARQL semantics (relying on the XQuery semantics). The new XSPARQL prototype implementation is described in Chapter 4. Chapter 5 explains some optimisations we developed to make XSPARQL query execution more efficient. To measure the effects of these optimisations we conducted a practical evaluation, the results of which are given in Chapter 6.

In this work information about people and their relations is used as running example. We will present, query, filter and transform data from this domain in both XML and RDF.

Chapter 2

Preliminaries

The various data representation mechanisms used nowadays are capable of making data of different sources accessible to computer programs. In this chapter we describe two data representation formats: a tree based approach (XML) in Section 2.1 and another one based on directed graphs (RDF) in Section 2.2. Furthermore we present the corresponding query languages: XQuery for XML in Section 2.3 and SPARQL for RDF in Section 2.4. In Section 2.5 will introduce XSPARQL, a language built by combining XQuery and SPARQL to make transformations between XML and RDF easier.

2.1 Data Representation with XML

An XML document is a hierarchically structured document (thus tree based), containing nested objects of different kinds. The most important ones are *elements*, allowing the nested structure, *attributes*, being element annotations and *text*, e. g., paragraphs of a document. Listing 2.1 shows the most important production rules of the XML grammar given in Appendix Section A.3. An XML document consists of a *root element* prefixed by a prolog (see Rule [1] in Listing 2.1), specifying the XML version and possibly the text encoding. This *root element* can contain character data, other elements, unparsed character data, processing instructions and comments (see Rules [39] and [43]). *Attributes* are contained in the XML element start tag (see Rules [40] and [41]).

The meaning of the nesting is not defined by XML but by the specification of the concrete XML format used. It could mean for example composition, aggregation, abstraction or some kind of inheritance. The strict hierarchical structure of XML documents makes representation of data with cyclic associations hard.

```
1 [1] document ::=  prolog element Misc*
2 [39] element ::=  EmptyElemTag | STag content ETag
3 [40] STag ::= '<' Name (S Attribute)* S? '>'
4 [41] Attribute ::=  Name Eq AttValue
5 [42] ETag ::= '</' Name S? '>'
6 [43] content ::=  CharData? ((element | Reference | CDSect | PI |
      Comment) CharData?)*
```

Listing 2.1: Excerpt of the XML grammar [Bray et al., 2008] in Appendix A.3

2.2 Data Representation with RDF

The Resource Description Framework (RDF) is a framework to describe resources of different kinds. Theses resources are addressed by URIs. RDF uses directed graphs as base data model. By adding relations, RDF provides a versatile way to represent data as graphs.

URIs can also be written as QNames similar as in XML. RDF triples are defined by three parts: subject, predicate and object. A set of RDF triples can be represented as a directed graph (or RDF graph) with labelled nodes and edges where the edge, or predicate, points from the subject to the object.

Definition 2.1 (RDF Triple, RDF Graph [Pérez et al., 2006])**.** Given the pairwise disjoint sets of URI references $\mathcal{U}$, blank nodes $\mathcal{B}$, and literals $\mathcal{L}$ a triple $(s, p, o) \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is called *RDF triple*. In such a triple s is the *subject*, p is the *predicate* and o the *object*. A set of RDF triples is called *RDF graph*.

URI references are used as names to uniquely define things, documents, persons, etc.RDF literals are used to encode simple strings, possibly tagged with a language annotation, or values of simple user-defined datatypes. Blank nodes are unnamed or anonymous nodes in an RDF graph used to reference to things without names or with unknown names (names meaning IRIs). But they can be addressed within the RDF graph. It depends on the concrete syntax how this behaviour can be achieved.

Unlike XML, RDF is defined in an abstract and syntax agnostic way to represent data, based on the mathematical notation of the directed graph. Therefore example RDF graphs can be drawn as graphs straightforwardly.

Example 2.1. Figure 2.1 shows the same data as Listing 1.1 in RDF. The three persons are represented by three different blank nodes. The predicate `a` is a shorthand for `rdf:type` which again means, that the subject is an instance of the class at object position, `foaf:Person` in this example. Each person has a name, given as a string literal, and relations to other persons. The example also shows how strings can be tagged with language tags (`"Charlie"@en`

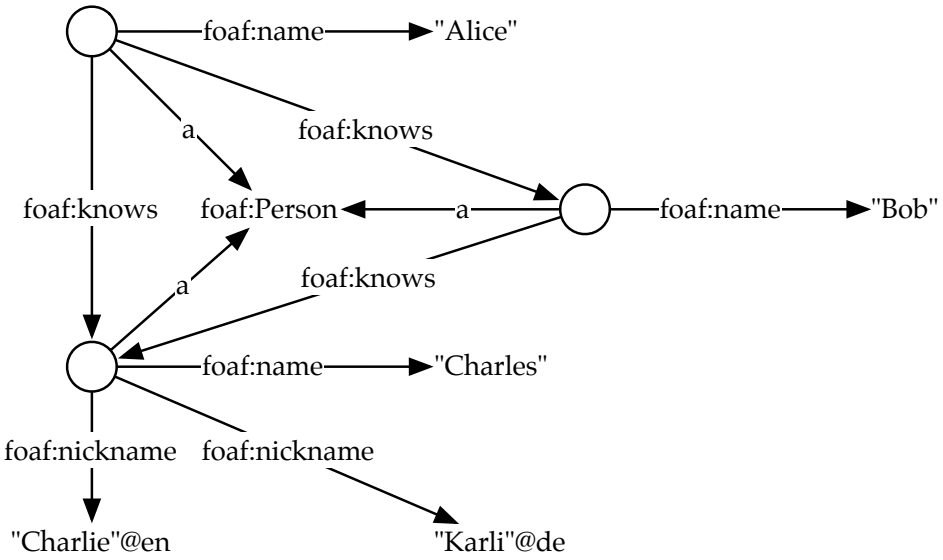

Figure 2.1: RDF example data

being the English nickname and `"Karli"@de` being a nickname of the same person in German).

The namespace prefix `rdf` resolves to the URL `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. The Friend-Of-A-Friend (FOAF) vocabulary [FOAF] used here was created exactly for the domain of people and their relations to each other. The namespace prefix `foaf` resolves to `http://xmlns.com/foaf/0.1/`.

2.2.1 RDF Syntax

There exist different serialisation syntaxes for RDF namely RDF/XML [Beckett, 2004], RDFa [Adida and Birbeck, 2008] for embedding RDF in XHTML pages, Notation 3 [Berners-Lee, 1998], N-Triples and Turtle [Beckett and Berners-Lee, 2008], where the latter three are related closely. In this work RDF examples are given in the *Terse* RDF *Triple Language*, shortly called *Turtle*.

Turtle is a subset of Notation 3 and allows to represent RDF data in a concise and intuitive way. Turtle is also the basis for the syntax of SPARQL basic graph patterns, used for graph pattern matching, and in SPARQL `construct` definitions [Prud'hommeaux and Seaborne, 2008]. Triples are represented as three consecutive strings—subject, predicate and object—separated with whitespace and terminated by a dot. *Blank nodes* can be represented as a pair of square brackets (called *anonymous blank node*) or as a node label starting

```
1  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
2  _:b1 a foaf:Person;
3       foaf:name "Alice";
4       foaf:knows _:b2;
5       foaf:knows _:b3.
6  _:b2 a foaf:Person;
7       foaf:name "Bob";
8       foaf:knows _:b3.
9  _:b3 a foaf:Person;
10      foaf:name "Charles";
11      foaf:nickname "Charlie"@en, "Karli"@de.
```

Listing 2.2: RDF example data in Turtle

with an underscore and a colon (*labelled blank node*). There exist different abbreviations to reduce repetition, such as the semicolon character to group predicate-object lists for the same subject, or the comma character to group lists of objects for the same subject-predicate pair.

Example 2.2. Listing 2.2 shows a Turtle serialisation of the RDF graph of Figure 2.1. Since the first four triples share the same subject, the abbreviation syntax using the semicolon is used. The last two triples, shown on line 11, use the abbreviation syntax for triples sharing both, subject and predicate. The blank node `_:b3` is subject of two triples with the same predicate, `foaf:nickname`. The person has nicknames in two different languages, one, `"Charlie"@en` annotated with an English language tag and another one, `"Karli"@de` annotated with a German language tag.

RDF/XML [Beckett, 2004] in contrast is a more verbose RDF representation using XML.

Example 2.3. Listing 2.3 shows the same example as Listing 2.2 in RDF/XML syntax.

One problem of RDF/XML when using XML tools is the ambiguity of the syntax. One single RDF graph can represented in different ways. Listing 2.2, 2.4, and 2.5 contain different serialisations of the relation of Alice and Charles. All three encode the same RDF graph.

A structured way to access and manipulate these expressive data formats is needed. The next two sections explain the most important query languages for XML and RDF.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/" xmlns:rdf="http://www.
    w3.org/1999/02/22-rdf-syntax-ns#">
  <foaf:Person>
    <foaf:name>Alice</foaf:name>
    <foaf:knows>
      <foaf:Person>
        <foaf:name>Bob</foaf:name>
        <foaf:knows rdf:nodeID="b3" />
      </foaf:Person>
    </foaf:knows>
    <foaf:knows>
      <foaf:Person rdf:nodeID="b3">
        <foaf:name>Charles</foaf:name>
      </foaf:Person>
    </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

Listing 2.3: RDF example data in RDF/XML

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/" xmlns:rdf="http://www.
    w3.org/1999/02/22-rdf-syntax-ns#">
  <foaf:Person rdf:about="alice/me">
    <foaf:knows>
      <foaf:Person foaf:name="Charles"/>
    </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

Listing 2.4: Alternative RDF/XML serialisation of Listing 2.3

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/" xmlns:rdf="http://www.
    w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <foaf:name>Charles</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about="alice/me">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <foaf:knows rdf:nodeID="x"/>
  </rdf:Description>
</rdf:RDF>
```

Listing 2.5: Another alternative RDF/XML serialisation of Listing 2.3

```
[33] FLWORExpr ::= (ForClause | LetClause)+ WhereClause? OrderByClause?
    "return" ExprSingle
[34] ForClause ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in
    " ExprSingle ("," "$" VarName TypeDeclaration? PositionalVar? "in"
    ExprSingle)*
[35] PositionalVar ::= "at" "$" VarName
[36] LetClause ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle
    ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*
[37] WhereClause ::= "where" ExprSingle
[38] OrderByClause ::= (("order" "by") | ("stable" "order" "by"))
    OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest
    " | "least"))? ("collation" URILiteral)?
[45] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[118] TypeDeclaration ::= "as" SequenceType
```

Listing 2.6: XQuery grammar [Boag et al., 2007]

2.3 Querying XML Data with XQuery

XQuery is an XML query languaged specified by a W3C recommendation [Boag et al., 2007]. In this section we will introduce the syntax, the processing model, and the semantics of XQuery.

2.3.1 Syntax of XQuery

An XQuery query starts with a prolog (containing namespace-, function- and variable declarations), followed by a sequence of FLWOR expressions. Listing 2.6 lists the most important grammar production rules of the FLWOR expression.

FL The expression starts with a list of `for` and `let` statements (see Rule [33] in Listing 2.6). `for` expressions (Rule [34]) iterate over values of a sequence. `let` expressions (Rule [36]) assign a sequence to a variable. Sequences can be generated by a nested FLWOR expression again.

W `where` expressions (Rule [37]) are used to filter values of the processed sequence.

O Ordering of sequences is accomplished by `order by` (Rules 38]–[41]).

R The `return` expression (Rule [33]) is responsible for building the result sequence. This expression consists of a template using the variables of

```
<relations>
{ for $person in /relations/person
  return
    <person name="{$person/@name}">
    { let $friends := $person/knows
      return <friends>{count($friends)}</friends>
    }
    </person>
  }
</relations>
```

Listing 2.7: Simple XQuery query

the former `for` and `let` expressions. It can also contain nested FLWOR expressions.

Additionally to FLWOR expressions, XQuery allows *conditional expressions* also known from imperative programming languages. Such an *IfExpr*, as given by Rule [45], is given by an *Expr* as condition, an *ExprSingle* used if the condition evaluates to *fn:true*, and a second *ExprSingle* used otherwise.

Example 2.4. The example query in Listing 2.7 creates a document containing a list of persons, the only child being an element `friends`, containing the number of persons known by the person. First the XML root element is constructed. The `for` expression iterates over all `person` elements which are children of the root element `relations` of the source XML tree. For each of these `persons`, the `name` attribute, is emitted as an attribute again. While still iterating over the persons, all the `knows` children are extracted and stored in the new `$friends` variable. Finally a new child element `friends`, containing the number of elements, i. e., `knows` relations, is created. Listing 1.1 shows the XML source document used and Listing 2.8 shows the corresponding XML result document.

The result in Listing 2.8 shows that Alice knows two people, Bob knows one person and Charles knows nobody.

2.3.2 Processing Model of XQuery

The XQuery processing model shown in Figure 2.2 shows core notions of the following XQuery semantics.

Generally XQuery queries are processed in two phases: Static analysis phase and dynamic evaluation phase. Before entering the static analysis phase, schema definitions may have to be imported, before the dynamic evaluation phase, the data (e. g. an XML document) has to be parsed.

```
<relations>
  <person name="Alice">
    <friends>2</friends>
  </person>
  <person name="Bob">
    <friends>1</friends>
  </person>
  <person name="Charles">
    <friends>0</friends>
  </person>
</relations>
```

Listing 2.8: Result of Listing 2.7

2.3.2.1 Static Analysis Phase

First the XQuery query is parsed and, given no errors occurred, represented in an Op-Tree (SQ1). Next, parts of the *static context* is initialised from the environment (SQ2). The static context contains information needed later in the static analysis phase but also in the dynamic evaluation phase. The semantics refers to the static context with the symbol *statEnv*. Parts of the static context are initialised with information from the query itself, i. e., namespace declarations (SQ3). An example for the usage of static context information is expanding QNames afterwards (SQ4). The central step in this phase is *normalisation* (SQ5): An XQuery query is reformulated into a simpler sublanguage *XQuery Core*. Since this step minimises the number of different syntactic objects, the semantics definition is simpler. When enabled, static type checking is performed as the last step in this phase (SQ6). It determines type information for all the expressions (as far as this is possible) and throws an error if computed and declared types of an expression are incompatible.

2.3.2.2 Dynamic Evaluation Phase

Again the environment, in this case the *dynamic context*, has to be initialised (DQ2, DQ3). The dynamic context contains runtime information and provides access to static context information. When the query is eventually evaluated the XQuery semantics uses information of the dynamic environment, referred to as *dynEnv*, as well as an XDM instance, i. e., a parsed XML document (DQ4, DQ5).

In summary the semantics of most syntactic objects is given by rules in three different stages: normalisation, static (type) analysis and dynamic evaluation.

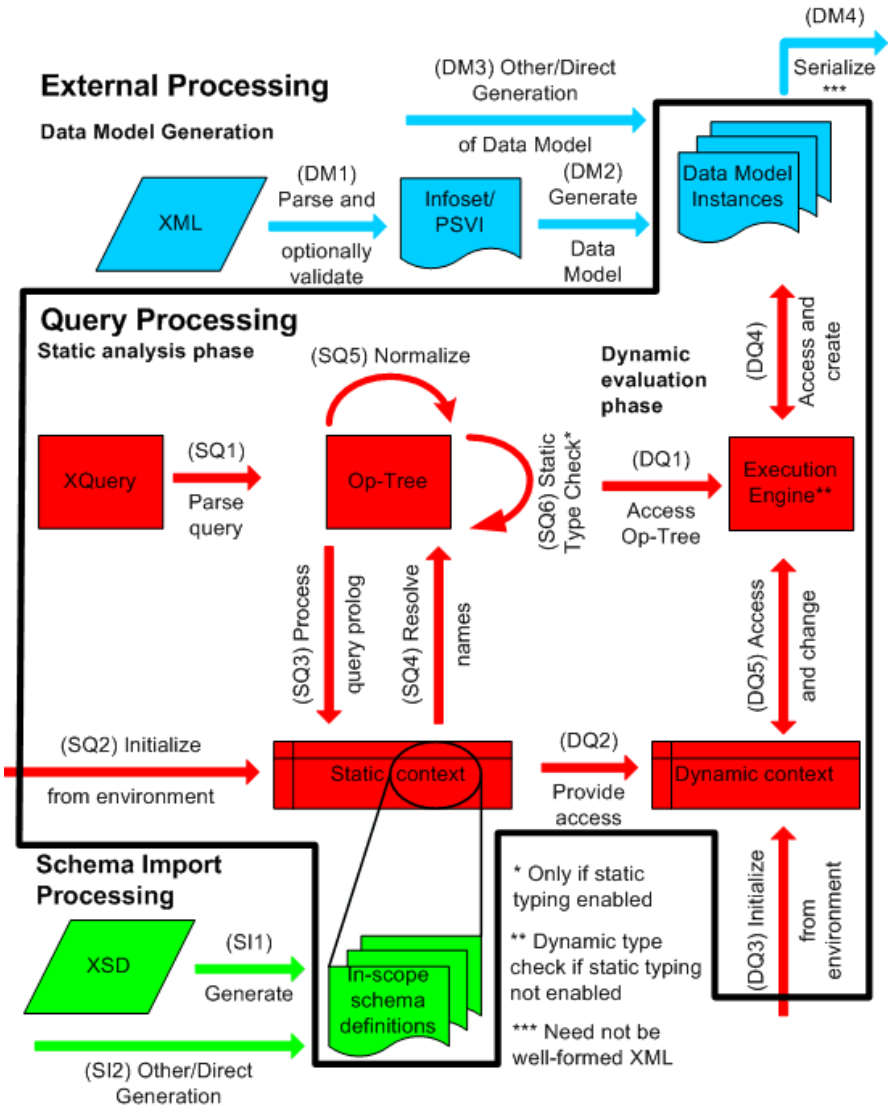

Figure 2.2: XQuery Processing Model Overview [Draper et al., 2007]

2.3.3 Semantics of XQuery

The semantics of XQuery, is defined in [Draper et al., 2007] in two steps: First the full XQuery syntax is reduced to the so called *XQuery Core* language. This step is called *normalisation*. XQuery Core is defined as abstract syntax whose intent is to provide a minimal set of expressions by replacing the richer expressions, written as syntactic sugar, with the simplest possible form of these expressions. XQuery Core is a minimal subset of XQuery losing no expressivity. The goals of this reduction are to minimise repetition in the

semantics definition and to make implementation and optimisation easier.

The next step in the semantics definition, *static type analysis*, can be used to infer types for expressions. At this stage no information about the input document(s) is used, only the query itself is analysed. Therefore not all type errors can be caught. In the last step the operational semantics of XQuery is defined using *dynamic evaluation rules*. They produce the XML result while ensuring type coherence.

The following semantics presentation is limited to FLWOR expressions, because this expression type is the only one needed to define the XSPARQL language.

2.3.3.1 Notations and Predefined Judgements

The basic building block of the XQuery semantics is the judgement. A judgement can either be true or false. The following judgement is true if the result of the evaluation of *Expr* is *Value*:

$$Expr \Rightarrow Value$$

The next judgement is true if the evaluation result of *Expr* is of type *Type*:

$$Expr : Type$$

The following judgement is true if $Type_1$ is a subtype of $Type_0$:

$$Type_1 <: Type_0$$

The notions of static context and dynamic context was already introduced in Section 2.3.2. In the semantics specification these contexts are implemented in two *environments*. An environment contains pairs of *symbols* (or keys) and *objects* grouped in *environment components*. To access the value, stored in the *varValue* component, of a variable *var*, of the environment *dynEnv* the following judgement would be used:

$$\text{dynEnv.varValue(var)}$$

Corresponding to the processing model a static environment *statEnv* and a dynamic environment *dynEnv* are used. In the semantics rules these environments are used to provide a context to store objects such as variable values, types or functions. To show which environment is used, the judgement is prefixed by the environment abbreviation followed by the $\vdash$ symbol. Therefore the following judgement would be read as: Given the dynamic environment *dynEnv* the expression $Expr$ is evaluated to the value $Value$:

$$\text{dynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{Value}$$

To map a symbol *symbol* to a new object *object* in the environment component *envComp* of the environment *env*, the following notation is used:

$$\text{env} + \text{envComp}(\text{symbol} \Rightarrow \text{object})$$

Inference rules are written as a list of judgements called *premises* followed by a *conclusion* judgement below:

$$\frac{\mathit{premise}_1 \quad \ldots \quad \mathit{premise}_n}{\mathit{conclusion}}$$

The conclusion of such an inference rule holds, if all the specified premises hold.

A special notation of inference rules used for normalisation, is the *mapping rule:*

$$\begin{array}{c} \text{object} \\ == \\ \text{mapped object} \end{array}$$

The following example mapping rule is used during normalisation of FLWOR expressions. It maps XQuery *ForClause*s to the simpler XQuery Core *ForClause*s:

$$\left[\!\!\left[\begin{array}{l} \texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ \mathit{Expr}_1, \\ \ldots, \\ \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ \mathit{Expr}_n \\ \mathit{FormalReturnClause} \end{array} \right]\!\!\right]_{\mathit{Expr}}$$

$$==$$

$$\begin{array}{l} \texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ [\![\mathit{Expr}_1]\!]_{\mathit{Expr}}\ \texttt{return} \\ \quad \ldots \\ \quad \texttt{for}\ \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ [\![\mathit{Expr}_n]\!]_{\mathit{Expr}}\ \texttt{return} \\ \quad\quad [\![\mathit{FormalReturnClause}]\!]_{\mathit{Expr}} \end{array}$$

2.3.3.2 FLWOR Expression

First XQuery Core only grammar productions, needed to simplify the semantics specification, are introduced (see Listing 2.9).

Additionally three new grammar productions used only in the semantics specification are introduced in Listing 2.10. These new rules are not part of XQuery Core but they allow to express the same semantics in less rules.

```
[24 (Core)] FLWORExpr ::= (ForClause | LetClause) "return" ExprSingle
[25 (Core)] ForClause ::= "for" "$" VarName TypeDeclaration?
    PositionalVar? "in" ExprSingle
[27 (Core)] LetClause ::= "let" "$" VarName TypeDeclaration? ":="
    ExprSingle
[26 (Core)] PositionalVar ::= "at" "$" VarName
[75 (Core)] TypeDeclaration ::= "as" SequenceType
[28 (Core)] OrderByClause ::= (("order" "by") | ("stable" "order" "by"))
     OrderSpecList
[29 (Core)] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[30 (Core)] OrderSpec ::= ExprSingle OrderModifier
[31(Core)] OrderModifier ::= ("ascending" | "descending")? ("empty" ("
    greatest" | "least"))? ("collation" URILiteral)?
```

Listing 2.9: XQuery Core grammar productions

```
[65 (Formal)] FormalFLWORClause ::= ForClause | LetClause | WhereClause
    | OrderByClause
[66 (Formal)] FormalReturnClause ::= FormalFLWORExpr | ("return" Expr)
[67 (Formal)] FormalFLWORExpr ::= FormalFLWORClause FormalReturnClause
```

Listing 2.10: XQuery semantics grammar productions

Normalisation

Rule 2.1. In this step FLWOR expressions are normalised to XQuery Core expressions. First the *ForClause* converted to simple nested *ForClause*s binding only a single variable:

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ \mathit{Expr}_1, \\ \cdots, \\ \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ \mathit{Expr}_n \\ \mathit{FormalReturnClause} \end{array} \right]\!\!\right]_{\mathit{Expr}} \\
\qquad == \\
\texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ [\![\mathit{Expr}_1]\!]_{\mathit{Expr}}\ \texttt{return} \\
\quad \cdots \\
\quad \texttt{for}\ \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ [\![\mathit{Expr}_n]\!]_{\mathit{Expr}}\ \texttt{return} \\
\quad\quad [\![\mathit{FormalReturnClause}]\!]_{\mathit{Expr}}
\end{array}
$$

Rule 2.2. Likewise compound *LetClause*s are normalised to simple *LetClause*s accordingly, binding one variable each:

$$
\begin{array}{c}
\left[\!\!\left[\begin{array}{l}
\texttt{let}\ \$VarName_1\ OptTypeDeclaration_1 := Expr_1, \\
\cdots, \\
\$VarName_n\ OptTypeDeclaration_n := Expr_n \\
FormalReturnClause
\end{array}\right]\!\!\right]_{Expr} \\
== \\
\texttt{let}\ \$VarName_1\ OptTypeDeclaration_1 := [\![Expr_1]\!]_{Expr}\ \texttt{return} \\
\quad \cdots \\
\quad\quad \texttt{let}\ \$VarName_n\ OptTypeDeclaration_n := [\![Expr_n]\!]_{Expr}\ \texttt{return} \\
\quad\quad\quad [\![FormalReturnClause]\!]_{Expr}
\end{array}
$$

Rule 2.3. XQuery *WhereClause*s are simplified to *IfExpr* essions:

$$
\begin{array}{c}
[\![\texttt{where}\ Expr_1\ FormalReturnClause]\!]_{Expr} \\
== \\
\texttt{if}([\![Expr_1]\!]_{Expr})\texttt{then}\ [\![FormalReturnClause]\!]_{Expr}\ \texttt{else}()
\end{array}
$$

Rule 2.4. The *OrderByClause* is normalised via an auxiliary mapping rule $[\![\cdot]\!]_{OrderSpecList}$:

$$
\begin{array}{c}
[\![\texttt{stable? order by}\ OrderSpecList\ FormalReturnClause]\!]_{Expr} \\
== \\
[\![OrderSpecList]\!]_{OrderSpecList}\ \texttt{return}\ [\![FormalReturnClause]\!]_{Expr}
\end{array}
$$

Rule 2.5. A *FormalReturnClause* is normalised by reduction to the contained *Expr*. This rule removes superfluous usage of the `return` keyword introduced by the *ForClause* and *LetClause* normalisation rules:

$$
\begin{array}{c}
[\![\texttt{return}\ Expr]\!]_{Expr} \\
== \\
[\![Expr]\!]_{Expr}
\end{array}
$$

On such normalised queries the semantics of the single *ForClause* and the single *LetClause* are defined.

For expression Next, we define the semantics for the normalised *ForClause* by means of static type analysis and dynamic evaluation semantics.

The first static type analysis rule handles the single *ForClause* without a position variable or typing information. Static type analysis

The auxiliary function *quantifier* determines the multiplicity of items of $Type_1$. Possible result values of this function are `one` (for a single value), ? (for none or one values), * (for zero or more values), or + (for one or more values).

Rule 2.6. By the first judgement the type, $Type_1$, of the expression $Expr_1$ is inferred. The second judgement expands the variable name to a variable in the static environment. Finally the type $Type_2$ of the result expression $Expr_2$ is inferred with the type of $Variable_1$ being the prime type of $Type_1$. The prime type is computed by building a union over all static item types in $Type_1$.

$$\frac{\begin{array}{c}\text{statEnv} \vdash Expr_1 : Type_1 \\ \text{statEnv} \vdash VarName_1 \textbf{ of var expands to } Variable_1 \\ \text{statEnv} + \text{varType}(Variable_1 \Rightarrow \text{prime}(Type_1)) \vdash Expr_2 : Type_2\end{array}}{\text{statEnv} \vdash \texttt{for } \$VarName_1 \texttt{ in } Expr_1 \texttt{ return } Expr_2 : Type_2 \cdot \text{quantifier}(Type_1)}$$

Rule 2.7. For `for` loops with a positional variable (`at`), we can set the type of that variable to `xs:integer` in the static environment.

$$\frac{\begin{array}{c}\text{statEnv} \vdash Expr_1 : Type_1 \\ \text{statEnv} \vdash VarName_1 \textbf{ of var expands to } Variable_1 \\ \text{statEnv} \vdash VarName_{pos} \textbf{ of var expands to } Variable_{pos} \\ \text{statEnv} + \text{varType}\left(\begin{array}{l} Variable_1 \Rightarrow prime(Type_1); \\ Variable_{pos} \Rightarrow \texttt{xs:integer} \end{array}\right) \vdash Expr_2 : Type_2\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$VarName_1 \texttt{ at } \$VarName_{pos} \texttt{ in } Expr_1 \\ \texttt{return } Expr_2 : Type_2 \cdot \text{quantifier}(Type_1)\end{array}}$$

Rule 2.8. In `for` loops with a type declaration (`as`), the type of the input expression $Expr_1$ is checked to be a subtype of the declared type.

$$\frac{\begin{array}{c}\text{statEnv} \vdash Expr_1 : Type_1 \\ Type_0 = [\![SequenceType]\!]_{sequencetype} \\ \text{statEnv} \vdash \text{prime}(Type_1) <: Type_0 \\ \text{statEnv} \vdash VarName_1 \textbf{ of var expands to } Variable_1 \\ \text{statEnv} + \text{varType}(Variable_1 \Rightarrow \text{prime}(Type_1)) \vdash Expr_2 : Type_2\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$VarName_1 \texttt{ as } SequenceType \texttt{ in } Expr_1 \\ \texttt{return } Expr_2 : Type_2 \cdot \text{quantifier}(Type_1)\end{array}}$$

Rule 2.9. Finally, a last rule contains the definition of a *ForClause* containing a position variable as well as a type declaration.

$$\frac{\begin{array}{c}\text{statEnv} \vdash Expr_1 : Type_1 \\ Type_0 = [\![SequenceType]\!]_{sequencetype} \\ \text{statEnv} \vdash \text{prime}(Type_1) <: Type_0 \\ \text{statEnv} \vdash VarName_1 \textbf{ of var expands to } Variable_1 \\ \text{statEnv} \vdash VarName_{pos} \textbf{ of var expands to } Variable_{pos} \\ \text{statEnv} + \text{varType}\left(\begin{array}{l} Variable_1 \Rightarrow \text{prime}(Type_1); \\ Variable_{pos} \Rightarrow \texttt{xs:integer} \end{array}\right) \vdash Expr_2 : Type_2\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$VarName_1 \texttt{ as } SequenceType \texttt{ at } \$VarName_{pos} \texttt{ in } Expr_1 \\ \texttt{return } Expr_2 : Type_2 \cdot \text{quantifier}(Type_1)\end{array}}$$

Dynamic evaluation

Rule 2.10. If the expression $Expr_1$ evaluates to the empty sequence, the whole `for` expression evaluates to the empty sequence.

$$\frac{\text{dynEnv} \vdash Expr_1 \Rightarrow ()}{\text{statEnv} \vdash \begin{array}{l}\texttt{for \$}VarName_1 \, OptTypeDeclaration \; OptPositionalVar \texttt{ in } Expr_1 \\ \texttt{return } Expr_2 \Rightarrow ()\end{array}}$$

Rule 2.11. Otherwise, if the expression $Expr_1$ evaluates to non-empty a sequence of items $Item_1, \ldots, Item_n$, the *ForClause*'s body $Expr_2$ is evaluated for each of these items. Eventually the evaluation of the *ForClause* results in the sequence of these evaluated items $Value_1, \ldots, Value_n$.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash Expr_1 \Rightarrow Item_1, \ldots, Item_n \\ \text{statEnv} \vdash VarName \textbf{ of var expands to } Variable \\ \text{dynEnv} + \text{varValue}(Variable \Rightarrow Item_1) \vdash Expr_2 \Rightarrow Value_1 \\ \ldots \\ \text{dynEnv} + \text{varValue}(Variable \Rightarrow Item_n) \vdash Expr_2 \Rightarrow Value_n\end{array}}{\text{dynEnv} \vdash \texttt{for \$}VarName \texttt{ in } Expr_1 \texttt{ return } Expr_2 \Rightarrow Value_1, \ldots, Value_n}$$

Rule 2.12. In `for` loops where a position variable is used, the number of the current iteration is assigned to it.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash Expr_1 \Rightarrow Item_1, \ldots, Item_n \\ \text{statEnv} \vdash VarName \textbf{ of var expands to } Variable \\ \text{statEnv} \vdash VarName_{pos} \textbf{ of var expands to } Variable_{pos} \\ \text{dynEnv} + \text{varValue}(Variable \Rightarrow Item_1; Variable_{pos} \Rightarrow 1) \vdash Expr_2 \Rightarrow Value_1 \\ \ldots \\ \text{dynEnv} + \text{varValue}(Variable \Rightarrow Item_n; Variable_{pos} \Rightarrow n) \vdash Expr_2 \Rightarrow Value_n\end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for \$}VarName \texttt{ at \$}VarName_{pos} \texttt{ in } Expr_1 \\ \texttt{return } Expr_2 \Rightarrow Value_1, \ldots, Value_n\end{array}}$$

Rule 2.13. In `for` loops with a type declaration, the type of each $Item_i$ is checked against the declared type.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash Expr_1 \Rightarrow Item_1, \ldots, Item_n \\ Type_0 = [\![SequenceType]\!]_{sequencetype} \\ \text{statEnv} \vdash Item_1 \textbf{ matches } Type_0 \\ \ldots \\ \text{statEnv} \vdash Item_n \textbf{ matches } Type_0 \\ \text{statEnv} \vdash VarName \textbf{ of var expands to } Variable \\ \text{dynEnv} + \text{varValue}(Variable \Rightarrow Item_1) \vdash Expr_2 \Rightarrow Value_1 \\ \ldots \\ \text{dynEnv} + \text{varValue}(Variable \Rightarrow Item_n) \vdash Expr_2 \Rightarrow Value_n\end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for \$}VarName \texttt{ as } SequenceType \texttt{ in } Expr_1 \\ \texttt{return } Expr_2 \Rightarrow Value_1, \ldots, Value_n\end{array}}$$

Rule 2.14. Analogously to the static type checking rules, another rule specifies the behaviour when both a position variable and a type declaration are present.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Item}_1, \ldots, \mathit{Item}_n \\ \mathit{Type}_0 = [\![\mathit{SequenceType}]\!]_{\mathit{sequencetype}} \\ \text{statEnv} \vdash \mathit{Item}_1 \textbf{ matches } \mathit{Type}_0 \\ \ldots \\ \text{statEnv} \vdash \mathit{Item}_n \textbf{ matches } \mathit{Type}_0 \\ \text{statEnv} \vdash \mathit{VarName} \textbf{ of var expands to } \mathit{Variable} \\ \text{statEnv} \vdash \mathit{VarName}_{pos} \textbf{ of var expands to } \mathit{Variable}_{pos} \\ \text{dynEnv} + \text{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_1; \mathit{Variable}_{pos} \Rightarrow 1) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_1 \\ \ldots \\ \text{dynEnv} + \text{varValue}(\mathit{Variable} \Rightarrow \mathit{Item}_n; \mathit{Variable}_{pos} \Rightarrow n) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_n\end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{VarName} \texttt{ as } \mathit{SequenceType} \texttt{ at } \$\mathit{VarName}_{pos} \texttt{ in } \mathit{Expr}_1 \\ \texttt{return } \mathit{Expr}_2 \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_n\end{array}}$$

Let Expression This section defines the semantics of the *LetClause* again first by static type analysis and then by dynamic evaluation semantics.

Static type analysis **Rule 2.15.** First the type of Expr_1, Type_1 is inferred. Next the statical environment is extended by type information for the expanded variable and the type of the resulting Expr_2, Type_2 is inferred.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1 \\ \text{statEnv} \vdash \mathit{VarName} \textbf{ of var expands to } \mathit{Variable} \\ \text{statEnv} + \text{varType}(\mathit{Variable} \Rightarrow \mathit{Type}_1) \vdash \mathit{Expr}_2 : \mathit{Type}_2\end{array}}{\text{statEnv} \vdash \texttt{let } \$\mathit{VarName} := \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2 : \mathit{Type}_2}$$

Rule 2.16. When an explicit type declaration occurs, the type of the first expression, Expr_1, has to be a subtype of the declared type.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1 \\ \mathit{Type}_0 = [\![\mathit{SequenceType}]\!]_{\mathit{sequencetype}} \\ \text{statEnv} \vdash \mathit{Type}_1 <: \mathit{Type}_0 \\ \text{statEnv} \vdash \mathit{VarName} \textbf{ of var expands to } \mathit{Variable} \\ \text{statEnv} + \text{varType}(\mathit{Variable} \Rightarrow \mathit{Type}_0) \vdash \mathit{Expr}_2 : \mathit{Type}_2\end{array}}{\text{statEnv} \vdash \texttt{let } \$\mathit{VarName} \texttt{ as } \mathit{SequenceType} := \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2 : \mathit{Type}_2}$$

Dynamic evaluation **Rule 2.17.** After assigning the result of Expr_1 to expanded variable, Expr_2 is evaluated in the extended static environment.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Value}_1 \\ \text{statEnv} \vdash \mathit{VarName}\ \textbf{of var expands to}\ \mathit{Variable} \\ \text{dynEnv} + \text{varValue}(\mathit{Variable} \Rightarrow \mathit{Value}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_2\end{array}}{\text{statEnv} \vdash \texttt{let}\ \$\mathit{VarName} := \mathit{Expr}_1\ \texttt{return}\ \mathit{Expr}_2 \Rightarrow \mathit{Value}_2}$$

Rule 2.18. With a type declaration the *LetClause* the evaluation result of $Expr_1$ is checked to be of the declared type.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Value}_1 \\ \mathit{Type}_0 = [\![\mathit{SequenceType}]\!]_{\mathit{sequencetype}} \\ \text{statEnv} \vdash \mathit{Value}_1\ \textbf{matches}\ \mathit{Type}_0 \\ \text{statEnv} \vdash \mathit{VarName}\ \textbf{of var expands to}\ \mathit{Variable} \\ \text{dynEnv} + \text{varValue}(\mathit{Variable} \Rightarrow \mathit{Value}_1) \vdash \mathit{Expr}_2 \Rightarrow \mathit{Value}_2\end{array}}{\text{statEnv} \vdash \texttt{let}\ \$\mathit{VarName}\ \texttt{as}\ \mathit{SequenceType} := \mathit{Expr}_1\ \texttt{return}\ \mathit{Expr}_2 \Rightarrow \mathit{Value}_2}$$

For the purposes of this thesis the overall semantics of `for` and `let` expressions are most important, we refer the interested reader to the full XQuery semantics [Draper et al., 2007] for further details.

2.4 Querying RDF Data with SPARQL

There are different query forms available explained below: `ask`, `construct`, `describe`, and `select`.

2.4.1 Syntax of SPARQL

A SPARQL query consists of the following main parts in exactly this order:

1. A *Prolog* for definition of base and default namespace as well as namespace abbreviations.
2. A clause determining the result format of the query. One of the following 4 *query forms* can be used:

   Construct returns an RDF graph specified by a *template* in a syntax similar to Turtle.

   Select projects the result to a list of variable bindings comparable to the result of an SQL query.

   Ask returns a boolean value which is *true* if the graph pattern can be matched at least once and *false* otherwise.

   Describe returns an implementation-dependent RDF graph describing the selected nodes.

```
[1] Query ::= Prologue ( SelectQuery | ConstructQuery | DescribeQuery |
    AskQuery )
[5] SelectQuery ::= 'SELECT' ( 'DISTINCT' | 'REDUCED' )? ( Var+ | '*' )
    DatasetClause* WhereClause SolutionModifier
[6] ConstructQuery ::= 'CONSTRUCT' ConstructTemplate DatasetClause*
    WhereClause SolutionModifier
[9] DatasetClause ::= 'FROM' ( DefaultGraphClause | NamedGraphClause)
[10] DefaultGraphClause ::= SourceSelector
[11] NamedGraphClause ::= 'NAMED' SourceSelector
[12] SourceSelector ::= IRIref | Var
[13] WhereClause ::= 'WHERE'? GroupGraphPattern
```

Listing 2.11: Main SPARQL syntax production rules

3. The *DatasetClause* determines a set of—possibly named—RDF source graphs. The dataset can be given explicitly, but an implicitly given dataset is also possible—for example when querying specific SPARQL services, also often called *SPARQL endpoints*.

4. The *WhereClause* provides the SPARQL graph pattern matching facility. This feature allows to match parts of RDF graphs by specifying so called *graph patterns*. Graph patterns can be triple patterns, which are triples allowing also variables. Furthermore a graph pattern can contain unions of other graph patterns or optional graph patterns. Graph patterns are evaluated on the graph representation of an RDF graph and are therefore independent of the concrete serialisation format.

5. The optional *solution modifier* part contains operators for ordering, duplicate elimination and slicing of the result set.

The main grammar productions are given in Listing 2.11.

Example 2.5. An example query is given in Listing 2.12: Return a list of the names of all persons occurring in the `relations.rdf` (see Listing 2.2) graph. In the prolog the `foaf` prefix is bound to the FOAF namespace `http://xmlns.com/foaf/0.1/`. The `SELECT` keyword implies that we are only interested in the `$name` variable. In the *WhereClause* `$person` is bound to a node of type `foaf:Person`. Then the `$name` variable of the *SelectClause* is bound to the name of that person. The *WhereClause* allows similar shortcuts as already specified for Turtle. The *SolutionModifier* declares to order the results by the `$name` variable. Table 2.1 on page 29 shows the result being a list of `$name` variable bindings.

When processing a SPARQL query first the prolog is processed by expanding all namespace prefixes. After that the prolog is not needed anymore. The

```
1 prefix foaf: <http://xmlns.com/foaf/0.1/>
2 select $name
3 from <relations.rdf>
4 where { $person a foaf:Person ; foaf:name $name . }
5 order by $name
```

Listing 2.12: Simple SPARQL query

Result #	$name
1	"Alice"
2	"Bob"
3	"Charles"

Table 2.1: Result of Listing 2.12

central part of a SPARQL query is the *WhereClause*. Its evaluation returns a list of variable binding sets. The result format as well as ordering and slicing can be seen as a query post-processing task.

2.4.1.1 Formal syntax of the WhereClause

Next we give an algebraic formalisation of the core fragment of SPARQL following [Pérez et al., 2006]. Additionally to the terms defined in Section 2.2 we assume the existence of a set of variables $\mathcal{V}_0$ being disjoint from the sets $\mathcal{U}$, $\mathcal{B}$, and $\mathcal{L}$. As an abbreviation we define $\mathcal{T} = \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$.

Definition 2.2 (Pérez et al., 2006)**.** A *triple pattern* is a tuple $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$.

If P, P_1, and P_2 are graph patterns and R is a SPARQL built-in condition, then the following are also *graph patterns*:

1. Triple pattern;
2. (P_1 AND P_2);
3. (P_1 UNION P_2);
4. (P_1 OPT P_2);
5. (P FILTER R).

Example 2.6. The following are valid triple patterns:

$$(\$X, \$Y, \$Z),\ (:p1, :\text{name}, \text{"Adam"}).$$

The following is a valid graph pattern:

$$((\$X, \$Y, \$Z) \text{ AND } (:p1, :\text{name}, \text{"Adam"})).$$

Definition 2.3 (Pérez et al., 2006)**.** A *SPARQL built-in condition* is an expression built of elements of the set $\mathcal{V} \cup \mathcal{U} \cup \mathcal{L}$, logical connectives ($\neg$, $\wedge$, $\vee$), equality and inequality symbols ($=$, $<$, $>$, $\leq$, $\geq$) and the unary predicates bound, isBlank, isIRI, isLiteral. We restrict our definition of FILTER expressions by ignoring the inequality symbols and all unary predicates but bound.

If $\$X, \$Y \in \mathcal{V}$ and $c \in \mathcal{U} \cup \mathcal{L}$ and R_1 and R_2 are built-in conditions then the following are built-in conditions:

1. bound($\$X$);
2. $\$X = c$ and $\$X = \$Y$, ;
3. $\neg R_1$, $(R_1 \vee R_2)$, and $(R_1 \wedge R_2)$.

The function var(P) of a graph pattern P returns the set of variables occurring in P. Consequently var(R) returns the set of variables occurring in the built-in condition R.

Furthermore we assume that all variables occurring in a built-in condition also occur in the corresponding graph pattern, i. e., for (P FILTER R) the condition var(R) $\subseteq$ var(P) holds.

Example 2.7. The following are valid built-in conditions: bound($\$X$), $\$X =$ "Adam", ((bound($\$X$)) $\wedge$ $\$X =$ "Adam") $\vee \neg$bound($\$Y$)).

2.4.2 Semantics of SPARQL graph patterns

We define the SPARQL semantics according to [Pérez et al., 2006].

Definition 2.4. Instead of the definition of a mapping as a partial function $\mu : \mathcal{V} \to \mathcal{T}$ in [Pérez et al., 2006], we define a *mapping* μ from $\mathcal{V} \cup \mathcal{B}$ to $\mathcal{T}$ as a partial function $\mu : (\mathcal{V} \cup \mathcal{B}) \to \mathcal{T}$. The domain of μ, dom(μ), is the subset of $\mathcal{V} \cup \mathcal{B}$ where μ is defined. By $\mu(t)$ we denote the triple obtained when replacing the variables and blank nodes in the triple pattern t according to μ. Two mappings μ_1 and μ_2 are *compatible* if for every $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, $\mu_1(v) = \mu_2(v)$.

Example 2.8. Consider the following three solution mappings:
$\mu_1 = \{(\$X, \text{"Adam"}), (\$Y, \text{"Betty"})\}$, $\mu_2 = \{(\$X, \text{"Adam"}), (\$Z, \text{"Carol"})\}$, and $\mu_3 = \{(\$Z, \text{"Derek"})\}$.

The following are results of compatibility between these three mappings:

1. Since $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \{\$X\}$ and $\mu_1(\$X) = \mu_2(\$X)$, the two mappings μ_1 and μ_2 are compatible;
2. Since $\text{dom}(\mu_1) \cap \text{dom}(\mu_3) = \emptyset$, the two mappings μ_1 and μ_3 are trivially compatible;

3. Since $\text{dom}(\mu_2) \cap \text{dom}(\mu_3) = \{\$Z\}$ and $\mu_1(\$Z) \neq \mu_2(\$Z)$, the two mappings μ_2 and μ_3 are not compatible.

Ω_1 and Ω_2 denote sets of solution mappings.

Definition 2.5 (Pérez et al., 2006)**.** We define the join, union, difference, and left outer join as:

1. $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$
2. $\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$
3. $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 | \text{ for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible mappings}\}$
4. $\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Example 2.9. Consider the following two solution sets:
$\Omega_1 = \{\{(\$X, \text{"Adam"}), (\$Y, \text{"Betty"})\}, \{(\$X, \text{"Adam"}), (\$Z, \text{"Carol"})\}\}$,
$\Omega_2 = \{\{(\$Z, \text{"Derek"})\}\}$,
then:

1. $\Omega_1 \bowtie \Omega_2 = \{\{(\$X, \text{"Adam"}), (\$Y, \text{"Betty"}), (\$Z, \text{"Derek"})\}\}$
2. $\Omega_1 \cup \Omega_2 = \left\{ \begin{array}{l} \{(\$X, \text{"Adam"}), (\$Y, \text{"Betty"})\}, \\ \{(\$X, \text{"Adam"}), (\$Z, \text{"Carol"})\}, \\ \{(\$Z, \text{"Derek"})\} \end{array} \right\}$
3. $\Omega_1 \setminus \Omega_2 = \{\{(\$X, \text{"Adam"}), (\$Z, \text{"Carol"})\}\}$
4. $\Omega_1 ⟕ \Omega_2 = \left\{ \begin{array}{l} \{(\$X, \text{"Adam"}), (\$Y, \text{"Betty"}), (\$Z, \text{"Derek"})\}, \\ \{(\$X, \text{"Adam"}), (\$Z, \text{"Carol"})\} \end{array} \right\}$

Based on the previous definitions we define the semantics of graph patterns as function $[\![\cdot]\!]_D$. This function takes a graph pattern expression as argument, while D denotes the RDF dataset, and returns a set of solution mappings.

Definition 2.6 (Pérez et al., 2006)**.** Let D be an RDF dataset over $\mathcal{T}$, t a triple pattern and P_1, P_2 graph patterns. Then the evaluation of a graph pattern over D, denoted by $[\![\cdot]\!]_D$, is defined recursively as follows:

1. $[\![t]\!]_D = \{\mu | dom(\mu) = var(t) \text{ and } \mu(t) \in D\}$ where $var(t)$ is the set of variables occurring in t;
2. $[\![P_1 \text{ AND } P_2]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$;
3. $[\![P_1 \text{ OPT } P_2]\!]_D = [\![P_1]\!]_D ⟕ [\![P_2]\!]_D$;

4. $[\![P_1 \text{ UNION } P_2]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$.

Definition 2.7 (Pérez et al., 2006)**.** The semantics for FILTER expressions is defined as follows: Given a mapping μ and a built-in condition R, we say that μ *satisfies* R, denoted by $\mu \models R$, if:

1. R is bound($\$X$) and $\$X \in \text{dom}(\mu)$;
2. R is $\$X = c$, $\$X \in \text{dom}(\mu)$ and $\mu(\$X) = c$;
3. R is $\$X = \$Y$, $\$X \in \text{dom}(\mu)$, $\$Y \in \text{dom}(\mu)$, and $\mu(\$X) = \mu(\$Y)$;
4. R is $(\neg R_1)$, R_1 is a built-in condition and it is not the case that $\mu \models R_1$;
5. R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions, and $\mu \vDash R_1$ or $\mu \models R_2$;
6. R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions, $\mu \vDash R_1$ and $\mu \models R_2$.

Definition 2.8 (Pérez et al., 2006)**.** Given an RDF dataset D and a FILTER expression $(P \text{ FILTER } R)$, then
$[\![(P \text{ FILTER } R]\!]_D = \{\mu \in [\![P]\!]_D \,|\, \mu \vDash R\}$.

Example 2.10. Consider the following RDF dataset D:

$$D = \left\{ \begin{array}{ll} (B_1, : \mathit{name}, \text{"Adam"}), & (B_1, : \mathit{email}, \text{me@adam.com}), \\ (B_2, : \mathit{name}, \text{"Betty"}), & (B_1, : \mathit{homepage}, \text{http://adam.com}), \\ (B_3, : \mathit{name}, \text{"Carol"}), & (B_2, : \mathit{email}, \text{betty@mail.com}), \\ (B_4, : \mathit{name}, \text{"Derek"}), & (B_3, : \mathit{phone}, \text{+43-1234-98765}) \end{array} \right\}$$

Next we give example graph patterns and their evaluation results. We chose to represent the set of solution mappings as a table instead of the familiar set notation.

- $P_1 = ((\$P, : \mathit{email}, \$E) \text{ OPT } (\$P, : \mathit{homepage}, \$H))$. Then
  $[\![P_1]\!]_D = [\![(\$P, : \mathit{email}, \$E)]\!]_D \bowtie [\![(\$P, : \mathit{homepage}, \$H)]\!]_D =$

$\$P$	$\$E$	$\$H$
B_1	me@adam.com	http://adam.com
B_2	betty@mail.com	

- Let
  $P_2 = ((\$P, : \mathit{name}, \$N) \text{ AND } ((\$P, : \mathit{email}, \$E)$
  $\text{UNION } (\$P, : \mathit{homepage}, \$E))$
  Then

$[\![P_2]\!]_D = [\![(\$P, : name, \$N)]\!]_D$
$\bowtie ([\![(\$P, : email, \$E)]\!]_D \cup [\![(\$P, : homepage, \$H)]\!]_D) =$

$\$P$	$\$N$	$\$E$	$\$H$
B_1	"Adam"	me@adam.com	
B_1	"Adam"		http://adam.com
B_2	"Betty"	betty@mail.com	

- $P_3 = (((\$P, : name, \$N)$ OPT $((\$P, : email, \$E))$ FILTER $\neg bound(?E))$.
Then
$[\![P_3]\!]_D = \{\mu \in ([\![(\$P, : name, \$N)]\!]_D \not\bowtie [\![(\$P, : email, \$E)]\!]_D)$
$| \ \mu \models \neg bound(\$E)\} =$

$\$P$	$\$N$	$\$E$
B_3	"Carol"	
B_4	"Derek"	

Definition 2.9 (Prud'hommeaux and Seaborne, 2008)**.** Before returning the result SPARQL replaces the *active graph* of a query (which can be understood as a representation of the query's dataset) is replaced by the so called *scoping graph*. The scoping graph is equivalent to the active graph, but it has no blank nodes in common with the active graph. The concept of the scoping graph becomes clearer when assuming that after evaluating the whole SPARQL graph pattern of a SPARQL *WhereClause*, all blank node labels are renamed. The new names of these blank nodes is completely arbitrary as long as one single blank node gets exactly one blank node label. This definition implies that blank node label es are only valid for one solution mappings set and it is therefore impossible to relate a blank node of a solution mappings set to a blank node of an RDF graph by itself.

Before any other following steps such as projection, ordering, and slicing, the domain of the solution mappings is reduced to $\mathcal{V}$, thus deleting all blank node mappings.

$$\text{result}(\Omega) = \{\mu' | \mu \in \Omega, \mu' = \text{reduce}(\mu), \mu' \neq \emptyset\}$$

$$\text{reduce}(\mu) = \{m | m \in \mu, \text{dom}(m) \in \mathcal{V}\}$$

The result(Ω) function removes every relation between $\mathcal{B}$ and $\mathcal{T}$ from the set of solution mappings Ω.

Example 2.11. Given the RDF dataset of Listing 2.13, we give the following example queries and their evaluation results. Labelled blank nodes are now given in the concrete SPARQL syntax.

```
1 _:b1 :name "Adam" .
2 _:b1 :email "me@adam.com" .
3 _:b1 :homepage "http://adam.com" .
4 _:b2 :name "Betty" .
5 _:b2 :email "betty@mail.com" .
6 _:b3 :name "Carol" .
7 _:b3 :phone "+43-1234-98765" .
8 _:b4 :name "Derek" .
```

Listing 2.13: Example RDF graph `semantics.rdf`

$P	$name
_:b1	"Adam"
_:b2	"Betty"
_:b3	"Carol"
_:b4	"Derek"

(a) Set of solution mappings

$P	$name
_:blank0	"Adam"
_:blank1	"Betty"
_:blank2	"Carol"
_:blank3	"Derek"

(b) Query result

Table 2.2: Results of first example

1. `SELECT * FROM <semantics.rdf> WHERE { $P :name $name . }`
   The graph pattern $(\$P, :\text{name}, \$name)$ evaluates to the set of solution mappings shown in Table 2.2a. Because of the blank node renaming the final query evaluation result differs in blank node labels (see Table 2.2b).

2. `SELECT * FROM <semantics.rdf> WHERE { _:bp1 :name _:bp2 . }`
   The graph pattern $(\_:\text{bp1}, :\text{name}, \_:\text{bp2})$ evaluates to the set of solution mappings shown in Table 2.3a. Since solution mappings containing only blank node mappings are deleted from the final result, the final result of this query is empty

3. `SELECT * FROM <semantics.rdf> WHERE { _:bp1 :name _:bp2 . OPTIONAL { _:bp1 :email $E . } }`
   The graph pattern $((\_:\text{bp1}, :\text{name}, \_:\text{bp2}) \text{ OPT } (\_:\text{bp1}, :\text{email}, \$E))$ evaluates to the set of solution mappings shown in Table 2.4a. Solution mappings containing only blank nodes are deleted completely from the final result (see Table 2.4b).

_:bp1	_:bp2
_:b1	"Adam"
_:b2	"Betty"
_:b3	"Carol"
_:b4	"Derek"

(a) Set of solution mappings

_:bp1	_:bp2

(b) Query result

Table 2.3: Results of second example

_:bp1	_:bp2	$E
_:b1	"Adam"	"me@adam.com"
_:b2	"Betty"	"betty@mail.com"
_:b3	"Carol"	
_:b4	"Derek"	

(a) Set of solution mappings

$E
"me@adam.com"
"betty@mail.com"

(b) Query result

Table 2.4: Results of third example

2.4.3 SPARQL Query Results XML Format

A format suited to exchange results of a SPARQL `select` or `ask` query is the SPARQL Query Results XML Format[Beckett and Broekstra, 2008]. Results of `construct` queries are RDF again.

The first child element of the root node with the name `head`, contains the list of variables of the SPARQL *SelectQuery*. The second child, `results`, contains a list of `result` elements, one for each solution mapping. Every `result` element contains one `binding` element for each variable binding.

Example 2.12. Listing 2.14 shows the result of the `select` query in Listing 2.12, i. e., the result as shown in Table 2.1 on page 29. The `head` element contains a list of the variable names used in the *WhereClause*. The `results` element contains one `result` element for each SPARQL solution set. Each result set contains a list of `binding` elements, one for each bound variable, each containing the bound value.

Note that we will use this format later in chapter 3 and 4 to process the results of a SPARQL engine.

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="name"/>
  </head>
  <results>
    <result>
      <binding name="name">
        <literal>Alice</literal>
      </binding>
    </result>
    <result>
      <binding name="name">
        <literal>Bob</literal>
      </binding>
    </result>
    <result>
     <binding name="name">
        <literal>Charles</literal>
      </binding>
   </result>
  </results>
</sparql>
```

Listing 2.14: SPARQL query results XML format

2.5 The XSPARQL Transformation Language

XSPARQL is a query language combining XQuery and SPARQL for transformations between RDF and XML. It was submitted as W3C member submission [Polleres et al., 2009, Krennwallner et al., 2009, Lopes et al., 2009, Passant et al., 2009].

The main usecases of XSPARQL are translating XML data to RDF and the other way round. Concrete example use cases are described in [Passant et al., 2009]. XSPARQL combines the advantages of XQuery and SPARQL making it possible to use a rich set of built-in functions (from XPath 2.0) and nested queries while being independent of the concrete RDF serialisation syntax.

Example 2.13. The query in Listing 2.15 converts the RDF data of Listing 2.2 to the example XML data in Listing 1.1. This transformation is an example for an RDF to XML conversion also called *lowering*.

2.5.1 Syntax of XSPARQL

XSPARQL is an XQuery extension incorporating constructs from SPARQL. There are two main extensions that lead XQuery to XSPARQL are the *SparqlFor-*

```
1  prefix foaf: <http://xmlns.com/foaf/0.1/>
2  <relations>
3  { for $Person $Name
4    from <relations.rdf>
5    where { $Person foaf:name $Name }
6    order by $Name
7    return
8      <person name="{$Name}">
9      { for $FName
10       from <relations.rdf>
11       where { $Person foaf:knows $Friend .
12               $Friend foaf:name $FName . }
13       return <knows>{$FName}</knows>
14     </person>
15   }
16 </relations>
```

Listing 2.15: Simple XSPARQL query

```
[33] FLWORExpr ::= (ForClause | LetClause | SparqlForClause )+
    WhereClause? OrderByClause? ReturnClause
[33a] ReturnClause ::= "return" ExprSingle | "construct"
    ConstructTemplate
[33b] SparqlForClause ::= "for" "distinct"? ("$" VarName ("$" Varname)*
    | "*") DatasetClause "where" GroupGraphPattern SolutionModifier
```

Listing 2.16: XSPARQL grammar productions Polleres et al. [2009]

Clause and the *ConstructTemplate*. The *SparqlForClause* allows syntax agnostic access to RDF data. The *ConstructTemplate* provides a concise way to specify RDF result graphs. [Polleres et al., 2009]

2.5.1.1 SparqlForClause

The *SparqlForClause* can be used whenever an XQuery `for` expression can be used. It can consist of up to five separate clauses (see also Listing 2.16):

1. The `for` clause itself is the same as a SPARQL `select` clause with the only difference that the starting keyword is `for` instead of `select`.

2. The *DatasetClause* consist of the static declaration of RDF data sources. Additionally XQuery variables containing dataset URLs are allowed here.

3. The mandatory `where` clause is based on the SPARQL `where` clause. It also allows using XQuery variables whenever SPARQL would allow

variables.

4. Order of results is controlled with `order by` as in SPARQL. The syntax for slicing of results, meaning limiting the number of returned results and the corresponding offset, both known from SQL, is the same as for SPARQL.

2.5.1.2 ConstructTemplate

The second extension object of SPARQL, the *ConstructTemplate*, can be used instead of an XQuery `return` expression. Additionally to the terms and variables already allowed in a SPARQL *ConstructTemplate*, RDF terms can be built dynamically.

In analogy to the *computed constructors* of XQuery (see [Boag et al., 2007, Section 3.7.3]), XSPARQL allows *constructed RDF terms*. As computed constructors are used to dynamically build nodes in a XML result tree, constructed RDF terms can be used to dynamically build RDF terms in an RDF graph. To minimise code obfuscation, a concise syntax is provided for such constructs. This is especially useful since all functions defined for XQuery/XSLT in [Malhotra et al., 2007] and even user defined functions are available in these templates. Since we are extending standard SPARQL new syntax is introduced for constructing the three different types of RDF terms: IRIs, blank nodes and literals.

Constructed IRIs Normally IRIs in SPARQL are enclosed in angle brackets. To create such an IRI in XSPARQL an expression needs to be embedded in angle brackets without any whitespace between the angle brackets and the curly braces: `<{`*Expr*`}>`. As a second format of IRIs a Qualified Name (QName), known from XML, can be used. It consists of a namespace prefix followed by a colon and the local part of the URI. To create such a URI in XSPARQL an *EmbeddedExpression* is allowed for both parts, namespace prefix and local part, e. g., `foaf:{`*Expr*`}`.

Constructed Blank Nodes To refer to blank nodes outside of the current context, labelled blank nodes can be constructed by giving the usual blank node label prefix `_:` immediately followed by an XQuery expression surrounded by curly braces.

Constructed Literals Different RDF literals can be created at runtime using different syntaxes. The most flexible one is the usage of a standard enclosed expression using curly braces. The optional datatype and language identifiers can also be created dynamically.

Example 2.14. The following incomplete example demonstrates these three variants:

```
declare variable $var := "1";
fn:concat("Alice",$var),
"Alice"^^{fn:concat(":",$var)},
"Alice"@{fn:concat("e","n")}
```

The examples would be evaluated to the following values:

```
("Alice1","Alice"^^:1,"Alice"@en)
```

2.5.1.3 SPARQL-style Queries

Additionally to extended XQuery queries, it is also possible to formulate SPARQL `construct` queries and use some features provided by XQuery. The extended *WhereClause*, which can be used in SPARQL-style queries as well as the extended `construct` clause were already described above.

2.5.1.4 Syntactic Restrictions

To avoid ambiguities and to make implementations simpler, the following syntactic restrictions are enforced in respect to XQuery:

- Variable names must not contain underscores "_", since they are used to make internal variables guaranteed unique.
- Other identifiers (namespace prefixes, function identifiers and blank node identifiers) must not start with an underscore, since these are also used internally.
- XQuery comments are not allowed, use XSPARQL comments instead, since the XQuery comment token "(:" could be confused with the start of some SPARQL filter expressions.
- Pragmas are not allowed, since the hash sign is used for XSPARQL comments.

Additionally some syntactic restrictions are applied to make the SPARQL parts compatible to XQuery:

- All keywords have to be in lowercase, since XQuery is case sensitive.
- Only `construct` queries allowed, not `select` `ask` or `describe`.
- Variables must start with '$' since '?' is not allowed in XQuery as it would be in SPARQL.

Although this section contains already a lot of details of the XSPARQL syntax, Appendix A shows the full XSPARQL grammar for reference.

2.5.2 Semantics of XSPARQL

The semantics of XSPARQL is defined on top of the XQuery semantics definition while using the same mechanism: First reducing the language to a core language by mapping rules and then defining the semantics on top of that. Since the XQuery semantics itself is already defined only new constructs extending XQuery have to be added. The SPARQL semantics is imported and glued in where needed. This semantics definition is a summary of the XSPARQL semantics in [Akhtar et al., 2007].

The new normalisation mapping rules $[\![\cdot]\!]_{Expr'}$ inherit all the standard XQuery $[\![\cdot]\!]_{Expr}$ mapping rules and extends them.

2.5.2.1 FLWOR' Expressions

Additionally to XQuery standard FLWOR expressions, a new construct, the *SparqlForClause* was introduced in the XSPARQL syntax.

Normalisation The XSPARQL semantics definition starts with normalisation rules for that new *SparqlForClause*.

Rule 2.19.

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \texttt{for } \$VarName_1 \ \cdots \ \$VarName_n \ DatasetClause \\ \texttt{where } GroupGraphPattern \ SolutionModifier \ ReturnClause \end{array} \right]\!\!\right]_{Expr'} \\
\qquad == \\
\left[\!\!\left[\begin{array}{l} \texttt{let } \$aux\_queryresult := \\ \quad \left[\!\!\left[\begin{array}{l} \$VarName_1 \ \cdots \ \$VarName_n \ DatasetClause \\ \texttt{where } GroupGraphPattern \ SolutionModifier \end{array} \right]\!\!\right]_{SparqlQuery} \\ \texttt{for } \$aux\_result \texttt{ in } \$aux\_queryresult\texttt{//}sparql\_result\texttt{:result} \\ \quad [\![VarName_1]\!]_{SparqlResult} \\ \quad \vdots \\ \quad [\![VarName_n]\!]_{SparqlResult} \\ ReturnClause \end{array} \right]\!\!\right]_{Expr}
\end{array}
$$

The $[\![\cdot]\!]_{SparqlQuery}$ rule is an auxiliary mapping rule taking care of SPARQL query parts.

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \$VarName_1 \ \cdots \ \$VarName_n \ DatasetClause \\ \texttt{where } GroupGraphPattern \ SolutionModifier \end{array} \right]\!\!\right]_{SparqlQuery} \\
\qquad == \\
fs{:}sparql\left(\left[\!\!\left[\begin{array}{l} fn{:}concat(\texttt{"SELECT } \$VarName_1 \ \cdots \ \$VarName_n \\ \quad DatasetClause \texttt{ where \{ "}, \\ \quad fn{:}concat(GroupGraphPattern), \texttt{ " \} } SolutionModifier\texttt{"}) \end{array} \right]\!\!\right]_{Expr'} \right)
\end{array}
$$

The *fs:sparql* function is an abstract function returning a SPARQL query result XML document [Beckett and Broekstra, 2008]. They only parameter is a SPARQL query given as a single string. Using the XML Schema definition of [Beckett and Broekstra, 2008] yields to the following definition:

```
1 fs:sparql($query as xs:string) as document-node(schema-element(
    sparql_result:sparql))
```

For each variable in each SPARQL result, four variables are created and initialised as defined by $[\![\cdot]\!]_{SparqlResult}$

Rule 2.20.

$$[\![\$VarName]\!]_{SparqlResult} == $$

```
let $VarName_Node  := $aux_result/sparql_result:binding[@name="VarName"]
let $VarName := data($VarName_Node/*)
let $VarName_NodeType := name($VarName_Node/*)
let $VarName_RDFTerm :=
  if($VarName_NodeType = "literal") then fn:concat("""",$VarName,"""")
  else if ($VarName_NodeType = "bnode") then fn:concat("_:",$VarName)
  else if ($VarName_NodeType = "literal") then fn:concat("<",$VarName,">")
  else ""
```

Static Typing All static typing rules are inherited from the XQuery semantics without change.

Dynamic Evaluation The function *fs:sparql* evaluates the first parameter as a SPARQL query according to the SPARQL semantics:

$$\frac{\text{The built-in function } \textit{fs:sparql} \text{ applied to } \textit{Value}_1 \text{ yields } \textit{Value}}{\text{dynEnv} \vdash \textbf{function } \textit{fs:sparql} \textbf{ with types } \texttt{xs:string} \textbf{ on values } (\textit{Value}_1) \textbf{ yields } \textit{Value}}$$

If the function parameter is not a valid SPARQL query or if another error occurs at runtime, the function itself yields an error:

$$\frac{\textit{Value}_1 \text{ cannot be evaluated according to SPARQL semantics}}{\text{dynEnv} \vdash \textbf{function } \textit{fs:sparql} \textbf{ with types } \texttt{xs:string} \textbf{ on values } (\textit{Value}_1) \textbf{ yields } \textit{fn:error()}}$$

One feature still not defined is the usage of variables in a SPARQL *GroupGraphPattern*. Variables are either bound by another *SparqlForClause* earlier, by another XQuery construct such as a FLWOR expression or they are left to by bound the *GroupGraphPattern*. If a variable is already bound, it is replaced by its value. Otherwise it is replaced with its own name, thus leaving evaluation to the SPARQL engine:

Rule 2.21. When a variable was bound by a *SparqlForClause* its RDF term representation is used.

$$\frac{\text{statEnv} \vdash \$\mathit{VarName\_RDFTerm}\ \textbf{bound}}{\text{statEnv} \vdash [\![\$\mathit{VarName}]\!]_{\mathit{VarSubst}} = [\![\$\mathit{VarName\_RDFTerm}]\!]_{\mathit{Expr}}}$$

Rule 2.22. When a variable was bound by an XQuery expression, it is replaced by its value.

$$\frac{\text{statEnv} \vdash \$\mathit{VarName}\ \textbf{bound} \qquad \text{statEnv} \vdash \textbf{not}(\$\mathit{Varname\_RDFTerm}\ \textbf{bound})}{\text{statEnv} \vdash [\![\$\mathit{VarName}]\!]_{\mathit{VarSubst}} = [\![\$\mathit{VarName}]\!]_{\mathit{Expr}}}$$

Rule 2.23. When a variable is currently not bound, it is replaced by its name as string.

$$\frac{\text{statEnv} \vdash \textbf{not}(\$\mathit{VarName}\ \textbf{bound}) \qquad \text{statEnv} \vdash \textbf{not}(\$\mathit{Varname\_RDFTerm}\ \textbf{bound})}{\text{statEnv} \vdash [\![\$\mathit{VarName}]\!]_{\mathit{VarSubst}} = [\![\texttt{"\$VarName"}]\!]_{\mathit{Expr}}}$$

FLWOR normalisation To be able to handle blank nodes in construct expressions, it is necessary to create a new position variable for `for` expressions if none is given.

Rule 2.24. First `for` expressions are normalised using the mapping rules of [Draper et al., 2007, Section 4.8]. After that a new (previously unbound) variable is introduced as position variable. To ensure that the new variable is indeed unbound before, it is prefixed with an underscore relying on the constraint underscores are not allowed as first character of variable names anymore (see XSPARQL syntax in Section 2.5.1).

$$\left[\!\!\left[\begin{array}{l} \texttt{for}\ \$\mathit{VarName}\ \mathit{OptTypeDecl}\ \texttt{in}\ \mathit{Expr} \\ \texttt{return}\ \mathit{FormalReturnClause} \end{array} \right]\!\!\right]_{\mathit{Expr'}}$$
$$==$$
$$\begin{array}{l} \texttt{for}\ \$\mathit{VarName}\ \mathit{OptTypeDecl}\ \texttt{at}\ \$\mathit{\_VarName\_Pos}\ \texttt{in}\ [\![\mathit{Expr}]\!]_{\mathit{Expr'}} \\ \texttt{return}\ [\![\mathit{FormalReturnClause}]\!]_{\mathit{Expr'}} \end{array}$$

Rule 2.25. Simple `let` expressions are normalised according to the XQuery semantics.

$$\left[\!\!\left[\begin{array}{l} \texttt{let}\ \$\mathit{VarName}\ \mathit{OptTypeDecl}\ \texttt{:=}\ \mathit{Expr} \\ \texttt{return}\ \mathit{FormalReturnClause} \end{array} \right]\!\!\right]_{\mathit{Expr'}}$$
$$==$$
$$\begin{array}{l} \texttt{let}\ \$\mathit{VarName}\ \mathit{OptTypeDecl}\ \texttt{:=}\ [\![\mathit{Expr}]\!]_{\mathit{Expr'}} \\ \texttt{return}\ [\![\mathit{FormalReturnClause}]\!]_{\mathit{Expr'}} \end{array}$$

2.5.2.2 CONSTRUCT Expressions

The second new grammar object introduced in the syntax section above, is the *ConstructTemplate*. It provides a Turtle-like syntax to specify RDF output.

Rule 2.26. A *ConstructTemplate* is mapped to a standard XQuery *ReturnClause*:

$$[\![\texttt{construct}\ \mathit{ConstructTemplate'}]\!]_{\mathit{Expr'}} \quad == \quad \Big[\!\Big[\ \texttt{return}\ \mathit{fn{:}concat}([\![\mathit{ConstructTemplate'}]\!]_{\mathit{SubjPredObjList}})\ \Big]\!\Big]_{\mathit{Expr}}$$

The new auxiliary mapping rules $[\![\cdot]\!]_{SubjPredObjList}$, $[\![\cdot]\!]_{Subject}$, $[\![\cdot]\!]_{PredObjList}$ and $[\![\cdot]\!]_{ObjList}$ are used to rewrite variables and blank nodes contained in the *ConstructTemplate*.

Definition 2.10 (New Judgements)**.** To ensure that the result is indeed a valid RDF graph built of valid RDF triples, the terms have to be tested for validity. Therefore we introduce three new judgements:

expr **is valid subject** is true if *expr* is bound and if it is either a blank node or an IRI, otherwise it is false.

expr **is valid predicte** is true if *expr* is bound and if it is an IRI, otherwise it is false.

expr **is valid object** is true if *expr* is bound and if it is either a blank node, an IRI or a literal, otherwise it is false.

Using these new judgements we can now define the rules needed to build RDF triples.

Rule 2.27.

$$\frac{\text{statEnv} \vdash \mathit{VarOrTerm'}\ \textbf{is valid subject}}{\text{statEnv} \vdash [\![\mathit{VarOrTerm'}\ \mathit{PropertyListNotEmpty'}]\!]_{\mathit{SubjPredObjList}} \quad == \quad \Big[\!\Big[\ \mathit{fn{:}concat}([\![\mathit{VarOrTerm'}]\!]_{\mathit{Subject}}, \ [\![\mathit{PropertyListNotEmpty'}]\!]_{\mathit{PredObjList}})\ \Big]\!\Big]_{\mathit{Expr}}}$$

Rule 2.28.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Verb}\ \textbf{is valid predicate} \\ \text{statEnv} \vdash \mathit{Object}_1\ \textbf{is valid object} \\ \vdots \\ \text{statEnv} \vdash \mathit{Object}_n\ \textbf{is valid object}\end{array}}{\begin{array}{l}\text{statEnv} \vdash [\![\mathit{Verb}\ \mathit{Object}_1, \ldots, \mathit{Object}_n]\!]_{\mathit{PredObjList}} \\ \qquad == \\ \Big[\!\Big[\ \mathit{fn{:}concat}([\![\mathit{Verb}]\!]_{\mathit{Expr'}}, \texttt{","}, [\![\mathit{Object}_1]\!]_{\mathit{Expr'}}, \texttt{","}, \texttt{","}, \ldots, \texttt{","}, \texttt{","}, [\![\mathit{Object}_n]\!]_{\mathit{Expr'}})\ \Big]\!\Big]_{\mathit{Expr}}\end{array}}$$

If any of those criteria fails, the triple containing it may be invalid, and would have to be removed from the result. The following rule is an example of such an invalidation rule:

Rule 2.29.

$$\frac{\text{statEnv} \vdash \textbf{not}(\textit{VarOrTerm}\ \textbf{is valid subject})}{\text{statEnv} \vdash [\![\textit{VarOrTerm}\ \textit{PropertyListNotEmpty}]\!]_{\textit{SubjPredObjList}} = [\![\text{" "}]\!]_{\textit{Expr}}}$$

One case to be careful about are labelled blank nodes in a *ConstructTemplate*. Labelled blank nodes cannot be output verbatim, but a unique blank node label has to be created for each evaluation of the *ConstructTemplate*. To truly create such unique labels, we decorated all the `for` expressions, and implicitly the *SparqlForClause* as well, with positional variables. We introduce a new static environment component called *statEnv*.posVars holding all in-context positional variables. The labels are then built by concatenating the positional variables, with the underscore as delimiter in between, to the static blank node label already given in the *ConstructTemplate*.

$$\frac{\text{statEnv} \vdash \textit{statEnv.posVars} = \textit{VarName}_1\textit{\_Pos}, \ldots, \textit{VarName}_n\textit{\_Pos}}{\text{statEnv} \vdash \begin{array}{c}[\![\text{"\_ : "} \textit{BnodeName}]\!]_{\textit{BNodeSubst}} = \\ [\![\textit{fn:concat}(\text{"\_ : "}, \textit{BNodeName}, \text{"\_"}, \textit{VarName}_1\textit{\_Pos}, \ldots, \textit{VarName}_n\textit{\_Pos})]\!]_{\textit{Expr}}\end{array}}$$

2.5.2.3 SPARQL Filter Operators

SPARQL defines its own functions allowed in a filter expression. In XSPARQL these functions are additionally implemented as standard XQuery functions:

```
1 BOUND($A as xs:string) as xs:boolean
2 isIRI($A as xs:string) as xs:boolean
3 isBLANK($A as xs:string) as xs:boolean
4 isLITERAL($A as xs:string) as xs:boolean
5 LANG($A as xs:string) as xs:string
6 DATATYPE($A as xs:string) as xs:anyURI
```

The semantics of these functions is defined as follows:

Rule 2.30. The *BOUND* function returns `fn:true` if its argument, a variable, is bound to a value, and `fn:false` otherwise:

$$\frac{\text{statEnv} \vdash \$\textit{VarName\_Node}\ \textbf{bound}}{\text{statEnv} \vdash [\![\textit{BOUND}(\$\textit{VarName})]\!]_{\textit{Expr}'} = \left[\!\!\left[\begin{array}{l}\texttt{if}(\textit{fn:empty}(\$\textit{VarName\_Node})) \\ \texttt{then}\ \textit{fn:false}()\ \texttt{else}\ \textit{fn:true}()\end{array} \right]\!\!\right]_{\textit{Expr}}}$$

Rule 2.31. The *isIRI* function returns `fn:true` if its argument, a variable, is of type IRI, and `fn:false` otherwise:

$$\frac{\text{statEnv} \vdash \$\mathit{VarName\_NodeType}\ \textbf{bound}}{\text{statEnv} \vdash \begin{array}{c}[\![\texttt{isIRI}(\$\mathit{VarName})]\!]_{\mathit{Expr}'} \\ = \\ \left[\!\!\left[\begin{array}{l}\texttt{if}(\mathit{fn{:}empty}(\$\mathit{VarName\_NodeType} = \texttt{"uri"})) \\ \texttt{then}\ \mathit{fn{:}false}()\ \texttt{else}\ \mathit{fn{:}true}()\end{array} \right]\!\!\right]_{\mathit{Expr}}\end{array}}$$

Rule 2.32. The *isBLANK* function returns `fn:true` if its argument, a variable, is a blank node, and `fn:false` otherwise:

$$\frac{\text{statEnv} \vdash \$\mathit{VarName\_NodeType}\ \textbf{bound}}{\text{statEnv} \vdash \begin{array}{c}[\![\mathit{isBLANK}(\$\mathit{VarName})]\!]_{\mathit{Expr}'} \\ = \\ \left[\!\!\left[\begin{array}{l}\texttt{if}(\mathit{fn{:}empty}(\$\mathit{VarName\_NodeType} = \texttt{"blank"})) \\ \texttt{then}\ \mathit{fn{:}false}()\ \texttt{else}\ \mathit{fn{:}true}()\end{array} \right]\!\!\right]_{\mathit{Expr}}\end{array}}$$

Rule 2.33. The *isLITERAL* function returns `fn:true` if its argument, a variable, is an RDF literal, and `fn:false` otherwise:

$$\frac{\text{statEnv} \vdash \$\mathit{VarName\_NodeType}\ \textbf{bound}}{\text{statEnv} \vdash \begin{array}{c}[\![\mathit{isLITERAL}(\$\mathit{VarName})]\!]_{\mathit{Expr}'} \\ = \\ \left[\!\!\left[\begin{array}{l}\texttt{if}(\mathit{fn{:}empty}(\$\mathit{VarName\_NodeType} = \texttt{"literal"})) \\ \texttt{then}\ \mathit{fn{:}false}()\ \texttt{else}\ \mathit{fn{:}true}()\end{array} \right]\!\!\right]_{\mathit{Expr}}\end{array}}$$

Rule 2.34. The *LANG* function returns the language tag of a literal annotated with a language tag:

$$\frac{\text{statEnv} \vdash \$\mathit{VarName\_Node}\ \textbf{bound}}{\text{statEnv} \vdash [\![\mathit{LANG}(\$\mathit{VarName})]\!]_{\mathit{Expr}'} = [\![\mathit{fn{:}string}(\$\mathit{VarName\_Node/@xml{:}lang})]\!]_{\mathit{Expr}}}$$

Rule 2.35. The *DATATYPE* function returns the datatype tag of a typed literal:

$$\frac{\text{statEnv} \vdash \$\mathit{VarName\_Node}\ \textbf{bound}}{\text{statEnv} \vdash [\![\mathit{DATATYPE}(\$\mathit{VarName})]\!]_{\mathit{Expr}'} = [\![\mathit{fn{:}string}(\$\mathit{VarName\_Node/@datatype})]\!]_{\mathit{Expr}}}$$

Chapter 3

Towards XSPARQL++

The last chapter introduced XSPARQL and the underlying query languages XQuery and SPARQL in detail. In this chapter we will describe XSPARQL extensions and refinements along with their semantics. The original XSPARQL language as introduced in the last chapter is referred to as *XSPARQL*. The extensions and refinements presented in this chapter lead to a new language we will call *XSPARQL++*.

First two new features, Constructed Dataset and Dataset Scoping, are explained with examples. *Constructed Dataset* provides a new way to formulate complex XSPARQL queries. Dataset Scoping gives a new interpretation of nested *SparqlForClause*s with an adapted semantics that returns more intuitive results for nested queries than the original XSPARQL.

In the second part of this chapter, a new XSPARQL semantics definition, based on the original XSPARQL semantics, is presented. Besides a formal definition of Constructed Dataset and Dataset Scoping it contains several detail improvements.

3.1 Constructed Dataset

This extension allows to manually optimise XSPARQL queries by creating an RDF graph containing only relevant triples and thus making query evaluation more efficient. For instance, this allows for a more concise way to formulate queries on XML data using a lot of internal references, by first converting it to a temporary RDF graph.

Constructed Datasets can also be seen as closely related to *database views*. Both allow to access dynamically built data sources (graphs for XSPARQL, tables for relational databases). They can simplify queries, provide aggregated data as source not only as result and thus make queries more maintainable.

Unfortunately SPARQL allows only IRIs in the *DatasetClause* and no variables or subqueries[1]. Any implementation allowing variables as datasets, while using a SPARQL engine to evaluate the query, must make the variable value retrievable at a specific URL.

Although XSPARQL syntactically allowed variables in a *DatasetClause* as well, the query would throw an error if the variable would not evaluate to an URL. In XSPARQL++ we allow also variables to be bound to an RDF graph, i. e. created by a *ConstructClause*. The following Listing shows a schematic XSPARQL++ query using the Constructed Dataset feature:

```
1 let $tempGraph := ...
2                   construct { ... }
3 for *
4 from $tempGraph
5 where { ... }
6 return ...
```

Section 3.6.4 will present the semantics specification of the Constructed Dataset feature.

Example 3.1. Martín and Gutierrez [2009] presented a problem being beyond the expressive power of SPARQL alone but not XSPARQL: create an RDF graph from DBLP [DBLP] data while returning the number of co-authored papers for each pair of co-authors. This problem, restricted to co-authors of a single person (in our example, Axel Polleres), can be formulated as an XSPARQL query using Constructed Datasets as shown in Listing 3.1.

The intermediary dataset `$ds`, created from line 4 to 12, contains the co-authors of each publication of Axel Polleres. The `$allauthors` variable is a sequence of all the persons Axel Polleres wrote a paper with, created from the previously Constructed Dataset `$ds`. By iterating over all pairs of authors, the number of all co-authored publications is counted (`$commonPubs`) and three triples documenting that are created for the end result of the query, under the condition that the number of `$commonPubs` is bigger than 0.

3.2 Dataset Scoping

When creating XML documents (e. g. from RDF sources) some kind of *grouping* is needed to generate the XML tree structure. Some examples of grouping in an XML document: a list of employees grouped by departments, a list of products grouped by buyers or a list of machine components grouped by manufacturer. Since grouping in this sense is not a feature of SPARQL so far, grouping has to be implemented externally. When working with RDF

[1] This might change with SPARQL 1.1[Harris and Seaborne, 2010]

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix dc:  <http://purl.org/dc/elements/1.1/>

let $ds := for *
  from <http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres>
  where { $pub dc:creator [] }
  construct { {
    for *
    from $pub
    where { $p dc:creator $o . }
    construct {$p dc:creator <{$o}>}
  } }

let $allauthors := for distinct $o
    from $ds
    where {$p dc:creator $o}
    order by $o
    return concat("<",$o,">")

for $auth at $auth_pos in $allauthors
  for $coauth in $allauthors[position() > $auth_pos]
    let $commonPubs := count( {
      for $pub
      from $ds
      where { $pub dc:creator $auth, $coauth }
      return $pub }
    )
    where ($commonPubs > 0)
    construct { [ :author1 $auth;
        :author2 $coauth;
        :commonPubs $commonPubs ] }
```

Listing 3.1: Number of co-authored publications for each pair of co-authors

data, *nested queries* provide an intuitive way to implement grouping. In this section we will present an intricate problem when dealing with such nested queries in XSPARQL and its solution by giving the query author control over the scope of datasets. We call this new feature *Dataset Scoping*.

3.2.1 Nested SPARQL FOR Clauses

Working with nested queries can lead to unexpected results. An example query (see Listing 3.2, originally presented by [Passant et al., 2009]) showing the effect (grouping the relations between persons by persons) is given now: For each person in a FOAF file, get a list of the associated friends in XML.

If we execute this query on the simple FOAF file shown in Listing 3.3,

```
1  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
2  <relations>
3  { for $Name from <http://xsparql.deri.org/data/relations.rdf>
4    where { $Person foaf:name $Name }
5    order by $Name
6    return <person name="{$Name}">
7           { for $FName from <http://xsparql.deri.org/data/relations.rdf>
8             where { $Person foaf:knows $Friend .
9                     $Person foaf:name $Name.
10                    $Friend foaf:name $FName. }
11            return <knows> { $FName }</knows>
12          }
13          </person>
14 }
15 </relations>
```

Listing 3.2: FOAF lowering

```
1  @prefix foaf: <http://xmlns.com/foaf/0.1/>.
2  [] foaf:name "Stefan";
3     foaf:homepage <http://www.stefandecker.org/> ;
4     foaf:knows [foaf:name "Bob"] .
5  [] foaf:name "Stefan";
6     foaf:homepage <http://stefanbischof.at/> ;
7     foaf:knows [foaf:name "Alice"] .
```

Listing 3.3: FOAF file about two different persons with the same name

containing two persons with the same name, the result is unexpected (see Listing 3.4). In contrast to the source data in Listing 3.3 where each person named "Stefan" knows exactly one other person, the result contains two persons with the same name, having exactly the same two friends. The problem is that the `foaf:name` object, used as a join variable here, is not necessarily unique for different persons. What we actually need, is to join over *distinct* persons regardless of their potentially coinciding names.

3.2.2 First Improvement

To improve the query we could try to use the `$Person` variable only for joining the persons with their friends. Listing 3.5 shows such a query created by modifying the query of Listing 3.2 by omitting the triple pattern containing `$Name` of the inner SPARQL query (line 10). See Listing 3.6 for the evaluation result.

This approach will not give the desired result either, because blank nodes labels are only valid within the solution set of a single SPARQL query, but,

```
<relations>
  <person name="Alice"/>
  <person name="Bob"/>
  <person name="Stefan">
    <knows>Bob</knows>
    <knows>Alice</knows>
  </person>
  <person name="Stefan">
    <knows>Bob</knows>
    <knows>Alice</knows>
  </person>
</relations>
```

Listing 3.4: Unexpected result of the query in Listing 3.2

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <http://stefanbischof.at/xsparql/relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
         { for $FName from <http://stefanbischof.at/xsparql/relations.
             rdf>
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>
         }
         </person>
}
</relations>
```

Listing 3.5: Improved FOAF Lowering

when used in a nested query, treated as variables, therein. Thus it is impossible to refer to a specific blank node from the outer query result in a nested SPARQL query, because blank nodes reused in a second query are handled like any blank node as a sort of existential quantifier: Such a blank node can be "bound" to any subject or object term. In our example this behaviour results in everybody knowing Bob and Alice because for every person SPARQL can find two triples of the form: Somebody (the blank node bound to `$Person`) knows somebody else (Alice and Bob).

Evaluation in Detail

In this section we show how the SPARQL parts of Listing 3.5 are evaluated. The outer *SparqlForClause*, lines 3–5, is rewritten to the SPARQL query in

```
<relations>
   <person name="Alice">
      <knows>Bob</knows>
      <knows>Alice</knows>
   </person>
   <person name="Bob">
      <knows>Bob</knows>
      <knows>Alice</knows>
   </person>
   <person name="Stefan">
      <knows>Bob</knows>
      <knows>Alice</knows>
  </person>
   <person name="Stefan">
      <knows>Bob</knows>
      <knows>Alice</knows>
   </person>
</relations>
```

Listing 3.6: Unexpected result of query improvement in Listing 3.5

```
select $Name
from <relations.rdf>
where { $Person foaf:name $Name }
order by $Name
```

Listing 3.7: SPARQL outer query of Listing 3.5

Listing 3.7. Since all variables are unbound the translation is straightforward. The query results in Table 3.1a on page 53 show four solution mappings. Although two solution mappings share the same binding for `$Name`, `$Person` is bound to four different blank nodes thus differentiating between all four persons.

Therefore the *ReturnClause* in lines 6–12 of Listing 3.5 is evaluated once for each of these solution mappings. In the first iteration the `$Person` variable of the inner *SparqlForClause* is replaced by its binding value, the first blank node. This rewriting leads to the rewritten SPARQL query shown in Listing 3.8. Evaluating this query leads to the results shown in Table 3.1b. Although Alice, identified by `_:b1`, has no outgoing `foaf:knows` relations, two solution mappings, are returned. During query evaluation all blank nodes of the *WhereClause* are mapped to RDF terms of the dataset. Since two blank nodes can be mapped to the blank node `_:b1` of the graph pattern, two solution mappings are returned.

The only difference of the second iteration to the first is the different

```
1 select $FName
2 from <relations.rdf>
3 where { _:b1 foaf:knows $Friend.
4         $Friend foaf:name $FName. }
```

Listing 3.8: First SPARQL inner query of Listing 3.5

$Person	$Name
_:b1	"Alice"
_:b2	"Bob"
_:b3	"Stefan"
_:b4	"Stefan"

(a) Result of Listing 3.7

$Friend	$FName
_:c1	"Alice"
_:c2	"Bob"

(b) Result of Listing 3.8

$Friend	$FName
_:d1	"Alice"
_:d2	"Bob"

(c) Result of Listing 3.9

Table 3.1: SPARQL results

blank node label. But since the blank node is mapped to an RDF term again the result shown in Table 3.1c is the same as for the first iteration.

The same applies for the last two iterations where the *SparqlForClause* is evaluated to the same solution set modulo blank node labels. This results lead to the strange result shown in Listing 3.6 where every person knows Alice and Bob.

In summary, if a variable v of a *SparqlForClause* S^O is bound to a blank node and v is reused in the *WhereClause* of the inner *SparqlForClause* S^I, then v in S^I is not referring to the same blank node as in S^O, but it is used as a sort of variable during evaluation of S^I. This behaviour may be unforeseeable for a query author, since it is not always known which variables will actually be bound to blank nodes.

3.2.3 Working Improvement – Dataset Scoping

To address this issue we introduce an extension called "Dataset Scoping" to XSPARQL, allowing to join over variables bound to blank nodes.

Firstly, by enlarging the active dataset (or dataset scope) from one *SparqlForClause* to several *SparqlForClause*s. This ensures that blank nodes returned by the outer query have the same identifiers in the active dataset of the inner query: Dataset Scoping retains the scoping graph after the evaluation of the outer query, and, instead of creating a new one as SPARQL semantics specifies, reuses this scoping graph for the following evaluation of the inner *SparqlForClause*.

```
select $FName
from <relations.rdf>
where { _:b2 foaf:knows $Friend.
        $Friend foaf:name $FName. }
```

Listing 3.9: Second SPARQL inner query of Listing 3.5

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <http://stefanbischof.at/xsparql/relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{$Name}">
         { for $FName
           where { $Person foaf:knows $Friend .
                   $Friend foaf:name $FName. }
           return <knows> { $FName }</knows>
         }
         </person>
}
</relations>
```

Listing 3.10: Improved Query using Dataset Scoping

Secondly, we treat blank nodes from the active dataset as constants, such that they will match like URIs or literals, instead of being treated like variables or blank nodes that are syntactically appearing in the query.

Dataset Scoping can be controlled syntactically: If the outer *SparqlForClause* specifies a *DatasetClause* but the inner *SparqlForClause* does not, Dataset Scoping is enabled, i. e., it is assumed that both nested queries are over the same active dataset. See Listing 3.10 for the example query using Dataset Scoping. Listing 3.11 shows the resulting XML document.

If the inner *SparqlForClause* contains a *DatasetClause*, traditional XSPARQL semantics without Dataset Scoping is applied, i. e., the evaluation result would be the XML document of Listing 3.6.

A formal description of the new semantics is given in Section 3.5. An implementation approach is described in Section 4.3.3.

3.3 XSPARQL++ Semantics Introduction

For the formal description of XSPARQL++ we use the notation of the XQuery formal semantics specification [Draper et al., 2007] (introduced earlier in Section 2.3.3). To define the XSPARQL++ semantics we extend the XQuery

```
<relations>
   <person name="Alice" />
   <person name="Bob" />
   <person name="Stefan">
      <knows>Bob</knows>
  </person>
   <person name="Stefan">
      <knows>Alice</knows>
   </person>
</relations>
```

Listing 3.11: Result of Listing 3.10

```
define type RDFTerm {
  uri | bnode | literal }

define type Binding {
  element variable of type xs:string,
  element term of type RDFTerm }

define type PatternSolution {
  element bindings of type Binding* }

define type RDFTriple {
  element subject of type RDFTerm,
  element predicate of type RDFTerm,
  element object of type RDFTerm }

define type RDFGraph {
  element RDFTriples of type RDFTriple* }

define type RDFDataset {
  element defaultGraph of type RDFGraph,
  element RDFNamedGraphs of type RDFNamedGraph* }

define type RDFNamedGraph {
  element name of type xs:string,
  element graph of type RDFGraph }
```

Listing 3.12: XSPARQL++ type definitions

semantics by two new objects, *SparqlForClause* and *ConstructClause*. Both were introduced in the syntax description in 2.5.1.

We extend the XQuery and XPath Data Model (XDM) [Fernández et al., 2007] by the following new types: New types

`RDFTerm` consisting of three types, namely `uri`, `bnode`, and `literal`, is

```
fs:bnode($arg as xs:string) as bnode
fs:literal($arg as xs:string) as literal
fs:literal($arg as xs:string, $datatype as xs:anyURI) as literal
fs:literal($arg as xs:string, $lang as xs:string) as literal
fs:uri($arg as xs:string) as uri
fs:uri($arg as xs:anyURI) as uri
```

Listing 3.13: Constructor functions for `RDFTerms`

```
fs:bnode($arg as xs:string) as bnode {
  validate { <bnode>$arg</bnode> }
}
```

Listing 3.14: Implementation of `bnode` constructor function

used for typing SPARQL variables. The three different RDF term types are introduced by the SPARQL Results XML Format [Beckett and Broekstra, 2008], while the type `RDFTerm` is equivalent to the `binding` of the SPARQL Results XML Format [Beckett and Broekstra, 2008].

`Binding` is the type of a pair, consisting of a variable name and its associated value as `RDFTerm`.

`PatternSolution` is the type of a sequence of `Bindings`. It represents a SPARQL query solution mapping, i. e., a SPARQL query result.

`RDFTriple` is defined as a sequence of three `RDFTerms`: `subject`, `predicate`, and `object`.

`RDFGraph` is the type of `construct` expressions and consists of a sequence of `RDFTriples`.

`RDFDataset` is the type of the *DatasetClause*s and defined as a default graph followed by a sequence of `RDFNamedGraphs`. An `RDFNamedGraph` consists of a `name` and an `RDFGraph`.

Listing 3.12 shows the formal definition of the new types, using XDM notation. To define the XSPARQL++ semantics we extend XQuery's $[\![\cdot]\!]_{Expr}$ normalisation rules.

Furthermore we introduce constructor functions for the three types `uri`, `bnode`, and `literal`. Listing 3.13 shows the static type signature of the new constructor functions, while Listing 3.14 shows the implementation of the constructor function for `bnode`. The other constructor functions are defined analogously.

After a short description of query prolog processing, the following sections we will give normalisation rules, the static, and the dynamic semantics of the two new objects *SparqlForClause* and *ConstructTemplate*.

3.4 Query Prolog

Namespace declarations in SPARQL syntax are converted to XQuery equivalents by normalisation rules. This rewriting allows the query author to use both XQuery and SPARQL namespace declarations for the query prolog.

$$[\![\texttt{prefix}\ \textit{NCName}\texttt{: <}\textit{URILiteral}\texttt{>}]\!]_{Expr} == [\![\texttt{declare namespace}\ \textit{NCNAME}\ \texttt{=}\ \textit{URILiteral}\ \texttt{;}]\!]_{Expr}$$

$$[\![\texttt{prefix :}\ \texttt{<}\textit{URILiteral}\texttt{>}]\!]_{Expr} == [\![\texttt{declare default element namespace}\ \textit{URILiteral}\ \texttt{;}]\!]_{Expr}$$

$$[\![\texttt{base}\ \texttt{<}\textit{URILiteral}\texttt{>}]\!]_{Expr} == [\![\texttt{declare base-uri}\ \textit{URILiteral}\ \texttt{;}]\!]_{Expr}$$

When the output format is RDF in Turtle syntax, the namespace declarations are prepended to the output. Although Turtle allows namespace declarations not only in the beginning of an RDF graph, the XSPARQL rewriting emits them before all the RDF triples.

3.5 FLWOR' Expression

The first part of a *FLWORExpr* is either a *SparqlForClause*, a standard XQuery *ForClause*, or a standard XQuery *LetClause*. If it is one of the standard XQuery clauses, it is normalised similarly as defined in Rule 2.24 and Rule 2.25. The exact new mapping rules are given below. The rest of this section describes the semantics of the *SparqlForClause* in detail. As for XQuery expressions, the semantics of the *SparqlForClause* is defined in three steps: normalisation, static typing, and dynamic evaluation.

3.5.1 Normalisation

In the first step FLWOR' expressions are normalised.

3.5.1.1 FOR *

The function vars(*GroupGraphPattern*) returns the list of previously unbound variables occurring in the *GroupGraphPattern*. If the *GroupGraphPattern* contains no previously unbound variables vars returns a random variable name. This measure just ensures that the rewritten query is still syntactically correct because the result of the whole *SparqlForClause* will be empty anyways. vars

Rule 3.1. A star instead of a list of variables in a *SparqlForClause* is handled like a star in a SPARQL *SelectClause* by returning all bound variables of the *WhereClause*.

$$
\begin{array}{c}
[\![\begin{array}{l} \texttt{for} * \\ \textit{DatasetClause} \\ \texttt{where}\ \textit{GroupGraphPattern} \\ \textit{SolutionModifier} \\ \textit{ReturnClause} \end{array}]\!]_{Expr} \\
== \\
[\![\begin{array}{l} \texttt{for}\ \text{vars}(\textit{GroupGraphPattern}) \\ \textit{DatasetClause} \\ \texttt{where}\ \textit{GroupGraphPattern} \\ \textit{SolutionModifier} \\ \textit{ReturnClause} \end{array}]\!]_{Expr}
\end{array}
$$

Rule 3.1 reduces the case where a star occurs in the `for` part of a *SparqlForClause* to the general case of a list of variables.

3.5.1.2 FLWOR Expressions

Plain XQuery FLWOR expressions are normalised in XSPARQL++ similarly as in XSPARQL (see Rules 2.24 and 2.25 in Section 2.5.2).

Rule 3.2. *ForClause*s are decorated with a positional variable after being normalised to XQuery Core.

$$
\begin{array}{c}
[\![\begin{array}{l} \texttt{for}\ \$\textit{VarName}_1\ \textit{OptTypeDeclaration}_1\ \textit{OptPositionalVar}_1\ \texttt{in}\ \textit{Expr}_1 \\ \quad \ldots, \\ \quad\ \$\textit{VarName}_n\ \textit{OptTypeDeclaration}_n\ \textit{OptPositionalVar}_n\ \texttt{in}\ \textit{Expr}_n \\ \quad\ \textit{FormalReturnClause} \end{array}]\!]_{Expr} \\
== \\
\begin{array}{l} \texttt{for}\ \$\textit{VarName}_1\ \textit{OptTypeDeclaration}_1 \\ \qquad\qquad [\![\textit{OptPositionalVar}_1]\!]_{PosVar}\ \texttt{in}\ [\![\textit{Expr}_1]\!]_{Expr}\ \texttt{return} \\ \quad \ldots \\ \qquad \texttt{for}\ \$\textit{VarName}_n\ \textit{OptTypeDeclaration}_n \\ \qquad\qquad [\![\textit{OptPositionalVar}_n]\!]_{PosVar}\ \texttt{in}\ [\![\textit{Expr}_n]\!]_{Expr}\ \texttt{return} \\ \qquad [\![\textit{FormalReturnClause}]\!]_{Expr} \end{array}
\end{array}
$$

```
for $x in ("red", "green", "blue
    "),
  $y at $position-of-y in ("
      light", "dark")
return fn:concat($y, " ", $x)
```

Listing 3.15: Normalisation example before rewriting

```
for $x at $_x_pos in ("red", "
    green", "blue"),
  $y at $position-of-y in ("
      light", "dark")
return fn:concat($y, " ", $x)
```

Listing 3.16: Normalisation of Listing 3.15

Rule 3.3. The new normalisation subrule $[\![\cdot]\!]_{PosVar}$ introduces a new positional variable if none is given. We assume a new position variable, distinct from any variable in scope, and represented by the formal semantics variable $*fs:new* (cf. [Draper et al., 2007, Section 4.12.6]).

$$[\![]\!]_{PosVar} == \texttt{at } \$fs{:}new$$

Rule 3.4. If a positional variable is present, then it is left unchanged.

$$[\![\texttt{at } \$PosVar]\!]_{PosVar} == \texttt{at } \$PosVar$$

The new rule $[\![\cdot]\!]_{PosVar}$ therefore ensures that every *ForClause* contains a positional variable. These positional variables are needed for the semantics definition of the *ConstructTemplate* below.

Furthermore we assume a new static environment component called *statEnv.posVars*, holding all positional variables of the current scope, i. e., the variables defined in an *OptPositionalVar* of every *ForClause*.

Example 3.2. The example query in 3.15 creates a list of colour names. The result of the query would be: `light red, dark red, light green, dark green, light blue, dark blue`. Listing 3.16 shows the normalisation of the XSPARQL query in 3.15 by adding position variables to the one `for` loop which is missing one.

*LetClause*s are normalised exactly as in XQuery.

The semantics of normalised *SparqlForClause*s will now be defined by means of static type analysis rules (Section 3.5.2) and dynamic evaluation rules (Section 3.5.3).

3.5.2 Static Type Analysis

Rule 3.5. The variables of the *SparqlForClause* are statically typed as `RDFTerm`.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{VarName}_1 \textbf{ of var expands to } \mathit{Variable}_1 \\ \vdots \\ \text{statEnv} \vdash \mathit{VarName}_n \textbf{ of var expands to } \mathit{Variable}_n \\ \text{statEnv} + \text{varType} \left(\begin{array}{c} \mathit{Variable}_1 \Rightarrow \texttt{RDFTerm}; \\ \cdots ; \\ \mathit{Variable}_n \Rightarrow \texttt{RDFTerm} \end{array} \right) \vdash \mathit{ReturnExpr} : \texttt{Type}_1 \end{array}}{\begin{array}{rl} & \texttt{for } \$\mathit{VarName}_1 \cdots \$\mathit{VarName}_n \ \mathit{DatasetClause} \\ \text{statEnv} \vdash & \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ & \texttt{return } \mathit{ReturnExpr} : \texttt{Type}_1 * \end{array}}$$

The static analysis rule of the *SparqlForClause* without an explicit dataset is defined analogously.

3.5.3 Dynamic Evaluation

For defining the dynamic evaluation semantics of the *SparqlForClause* we first introduce a new environment component and two new functions.

activeDataset

We introduce a dynamic environment component *dynEnv.activeDataset* which holds the dataset of a *SparqlForClause*. The *activeDataset* component is initially empty or set by the system environment. For *SparqlForClause*s containing a *DatasetClause*, the dataset is assigned to the *activeDataset* component. This measure allows nested *SparqlForClause*s to reuse a dataset of an enclosing *SparqlForClause* and is therefore part the Dataset Scoping feature implementation.

fs:sparql

As interface to SPARQL we introduce the new function *fs:sparql*. It evaluates a query, given as a set of parameters, according to the SPARQL semantics [Prud'hommeaux and Seaborne, 2008]. The first step before evaluating the query is to replace any bound variable of the *GroupGraphPattern* with its value in the current (dynamic) context. Unbound variables are (possibly) bound by the *SparqlForClause* later. By using the previously defined `PatternSolution` type, the function returns a *list of variable bindings* by matching a *GroupGraphPattern* `$groupGraphPattern` against the *activeDataset* (of the dynamic environment). The *SolutionModifier* is used to control ordering and slicing of the solution set. The function signature of *fs:sparql* is therefore defined as:

```
fs:sparql($groupGraphPattern as xs:string, $solutionModifiers as xs:
    string?) as PatternSolution*
```

fs:value

The auxiliary function *fs:value* returns the binding of a specified SPARQL variable *Variable* in the solution sequence *PS*. If the *Variable* is not bound in *PS*, *fs:value* returns the empty sequence $()$. We will use *fs:value* to access the results of the previously defined function *fs:sparql*. Its static type signature is defined as follows:

```
fs:value($ps as PatternSolution, $variable as xs:string) as RDFTerm
```

Using the two new functions *fs:sparql*, *fs:value*, and the new environment component *activeDataset* we define the dynamic evaluation semantics of the *SparqlForClause*.

Rule 3.6. We use the function *fs:sparql* to evaluate the *GroupGraphPattern*, including the *SolutionModifier*. For every pattern solution *PS* we create new variables containing the values of the result and finally evaluate *ReturnExpr* in this changed environment. This results in a sequence of values.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{VarName}_1 \textbf{ of var expands to } \mathit{Variable}_1 \\ \cdots \\ \text{statEnv} \vdash \mathit{VarName}_n \textbf{ of var expands to } \mathit{Variable}_n \\ \text{dynEnv} \vdash \mathit{fs:sparql}(\mathit{GroupGraphPattern}, \\ \qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv} + \text{varValue} \left(\begin{array}{c} \mathit{Variable}_1 \Rightarrow \mathit{fs:value}(\mathit{PS}_1, \mathit{Variable}_1); \\ \cdots ; \\ \mathit{Variable}_n \Rightarrow \mathit{fs:value}(\mathit{PS}_1, \mathit{Variable}_n) \end{array} \right) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \cdots \\ \text{dynEnv} + \text{varValue} \left(\begin{array}{c} \mathit{Variable}_1 \Rightarrow \mathit{fs:value}(\mathit{PS}_m, \mathit{Variable}_1); \\ \cdots ; \\ \mathit{Variable}_n \Rightarrow \mathit{fs:value}(\mathit{PS}_m, \mathit{Variable}_n) \end{array} \right) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\begin{array}{rl} & \texttt{for } \$\mathit{VarName}_1 \cdots \$\mathit{VarName}_n \\ \text{dynEnv} \vdash & \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ & \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}}$$

Rule 3.7. If the evaluation of the *fs:sparql* function does not yield any solution mapping, the evaluation result of the whole *SparqlForClause* is the empty sequence.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{VarName}_1 \textbf{ of var expands to } \mathit{Variable}_1 \\ \cdots \\ \text{statEnv} \vdash \mathit{VarName}_n \textbf{ of var expands to } \mathit{Variable}_n \\ \text{dynEnv} \vdash \mathit{fs:sparql}(\mathit{GroupGraphPattern}, \mathit{SolutionModifier}) \Rightarrow () \end{array}}{\begin{array}{rl} & \texttt{for } \$\mathit{VarName}_1 \cdots \$\mathit{VarName}_n \\ \text{dynEnv} \vdash & \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ & \texttt{return } \mathit{ReturnExpr} \Rightarrow () \end{array}}$$

Next we will define *SparqlForClause*s containing *DatasetClause*s. As the definition of the scoping graph (see Definition 2.9) implies, blank nodes are scoped to a set of pattern solutions of a single SPARQL query.

In XSPARQL++ instead, we redefine the scope of the scoping graph as the a whole XSPARQL query which may includes several *SparqlForClause*s. By this redefinition, and by treating blank nodes from the active dataset as constants

for the graph pattern matching, XSPARQL++ provides a measure to refer to blank nodes in a *SparqlForClause*, bound by an enclosing *SparqlForClause*.

We assume that the previously defined *fs:sparql* function adheres to this slightly modified version of graph pattern matching, compared to the original SPARQL semantics.

DatasetClauses

As an auxiliary grammar production rule used only for the semantics specification we introduce *DatasetClauses*:

```
DatasetClauses ::= DatasetClause*
```

To implement the Dataset Scoping feature, we define that every *SparqlForClause* containing an explicit *DatasetClause* creates a new scoping graph. If the *SparqlForClause* contains no *DatasetClause* the previous scoping graph is used.

fs:dataset

Therefore we introduce a new function *fs:dataset* which creates a new scoping graph `RDFDataset`. Any FLWOR' expression nested in a *DatasetClause* must evaluate to value of type `uri` or `RDFGraph`. If it is of type `uri` we assume that the corresponding `RDFGraph` can be retrieved over HTTP. The function is defined as follows:

```
fs:dataset($datasetClauses as xs:string) as RDFDataset
```

Rule 3.8. The *DatasetClause* is evaluated by the *fs:dataset* function. Before evaluating the *fs:sparql* function, the *activeDataset* component is set to the newly created dataset.

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{VarName}_1 \textbf{ of var expands to } \mathit{Variable}_1 \\ \vdots \\ \text{statEnv} \vdash \mathit{VarName}_n \textbf{ of var expands to } \mathit{Variable}_n \\ \text{dynEnv} \vdash \mathit{fs{:}dataset}(\mathit{DatasetClauses}) \Rightarrow \mathit{Dataset} \\ \text{dynEnv} + \text{activeDataset}(\mathit{Dataset}) \vdash \mathit{fs{:}sparql}(\mathit{GroupGraphPattern}, \\ \qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv} + \text{varValue}\begin{pmatrix} \mathit{Variable}_1 \Rightarrow \mathit{fs{:}value}(\mathit{PS}_1, \mathit{Variable}_1); \\ \cdots; \\ \mathit{Variable}_n \Rightarrow \mathit{fs{:}value}(\mathit{PS}_1, \mathit{Variable}_n) \end{pmatrix} \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \vdots \\ \text{dynEnv} + \text{varValue}\begin{pmatrix} \mathit{Variable}_1 \Rightarrow \mathit{fs{:}value}(\mathit{PS}_m, \mathit{Variable}_1); \\ \cdots; \\ \mathit{Variable}_n \Rightarrow \mathit{fs{:}value}(\mathit{PS}_m, \mathit{Variable}_n) \end{pmatrix} \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\begin{array}{l}\texttt{for } \$\mathit{VarName}_1 \cdots \$\mathit{VarName}_n \ \mathit{DatasetClauses} \\ \text{dynEnv} \vdash \ \texttt{where } \mathit{GroupGraphPattern}\ \mathit{SolutionModifier} \\ \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m\end{array}}$$

If the *fs:sparql* function returns no mappings, the result of the whole *SparqlForClause* will be the empty sequence, also when containing *DatasetClauses* (cf. Rule 3.7).

```
[60'] RDFLiteral' ::= String' ( LANGTAG | '@{' FLWORExpr '}' | ( '^^'
    IRIref') )?
[66'] String' ::= STRING_LITERAL1 | STRING_LITERAL2 |
    STRING_LITERAL_LONG1 | STRING_LITERAL_LONG2 | '{' FLWORExpr '}'
[67'] IRIref' ::= IRI_REF | '<{' FLWORExpr '}>' | PrefixedName'
[68'] PrefixedName' ::= PNAME_LN( (PN_PREFIX | '{' FLWORExpr '}')? ':' (
    PN_LOCAL | '{' FLWORExpr '}') ) | PNAME_NS
[69'] BlankNode' ::= BLANK_NODE_LABEL | ANON | '_:{' FLWORExpr '}'
```

Listing 3.17: Modified *ConstructTemplate* syntax

3.6 ConstructTemplate

The second syntactic object introduced by XSPARQL is the *ConstructTemplate*. It can be used instead of XQuery *ReturnExpression*s and allows to formulate RDF output in a concise manner using the Turtle like syntax of SPARQL *ConstructTemplate*s.

Listing 3.17 shows the main XSPARQL modifications of the SPARQL *ConstructTemplate* syntax, that is the introduction of constructed RDF terms. Appendix A.2.3 presents the SPARQL part of the XSPARQL grammar productions.

3.6.1 Normalisation of ConstructTemplates

Normalisation, as the first step in formalising the semantics of *ConstrucTemplate*s, introduces functions to validate the constructed RDF terms later.

Rule 3.9. First the special case of SPARQL style queries, a single *ConstructClause* preceding a single *WhereClause*, are rewritten to a more general query by prepending the query with `for *` and moving the *ConstructClause* to the end of the query.

$$
\left[\!\!\left[\begin{array}{l}\texttt{construct}\ \mathit{ConstructTemplate}'\\ \mathit{DatasetClause}\\ \mathit{WhereClause}\\ \mathit{SolutionModifier}\end{array}\right]\!\!\right]_{\mathit{Expr}}
$$

$$
==
$$

$$
\left[\!\!\left[\begin{array}{l}\texttt{for}\ *\ \mathit{DatasetClause}\\ \mathit{WhereClause}\\ \mathit{SolutionModifier}\\ \texttt{construct}\ \mathit{ConstructTemplate}'\end{array}\right]\!\!\right]_{\mathit{Expr}}
$$

This step reduces the query to the already known format of `for *` which was already handled in Section 3.5.1.1.

```
1 construct {
2    $person :name { fn:concat($firstname, " ", $lastname) } .
3 }
4 from <relations.rdf>
5 where {
6    $person :firstname $firstname .
7    $person :lastname  $lastname  .
8 }
```

Listing 3.18: XSPARQL `construct` query

```
1 for *
2 from <relations.rdf>
3 where {
4    $person :firstname $firstname .
5    $person :lastname  $lastname  .
6 }
7 construct {
8    $person :name { fn:concat($firstname, " ", $lastname) } .
9 }
```

Listing 3.19: Normalised XSPARQL `construct` query

Example 3.3. Listing 3.18 shows a simple SPARQL style query transforming a name given in two RDF triples to a concatenated form of the name in one RDF triple. It is normalised by the just introduced rule to the query in Listing 3.19.

In general we rewrite *ConstructClause*s to a standard `return` clauses. The result will be a string typed as `RDFGraph`.

This rule overwrites Rule 2.26.

Rule 3.10. Normalise `construct` clauses to XQuery `return` expressions.

$$[\![\texttt{construct}\ \mathit{ConstructTemplate}']\!]_{\mathit{Expr}} == \texttt{return}\ \mathit{fs{:}evalTemplate}([\![\mathit{ConstructTemplate}']\!]_{\mathit{normaliseTemplate}})$$

The normalisation rule $[\![\cdot]\!]_{\mathit{normaliseTemplate}}$ maps the various syntactic abbreviations to "simple triples", i. e., a sequence of *Subject*, *Predicate*, and *Object*.

We omit full details of this normalisation and give only an example of a normalisation rule dealing with Turtle ; shortcut notation to abbreviate predicate-object lists for a common subject.

Rule 3.11. Normalise one variant of the *Subject-PredicateObjectList* to simple triples.

$$\left[\!\!\left[\begin{array}{l} \mathit{Subject}\ \mathit{Predicate}_1\ \mathit{Object}_1; \\ \ldots; \\ \mathit{Predicate}_n\ \mathit{Object}_n \end{array} \right]\!\!\right]_{\mathit{normaliseTemplate}}$$
$$==$$
$$\begin{array}{c} \mathit{Subject}\ \mathit{Predicate}_1\ \mathit{Object}_1 \\ \ldots \\ \mathit{Subject}\ \mathit{Predicate}_n\ \mathit{Object}_n \end{array}$$

Rules for similar abbreviations allowed in SPARQL *ConstructTemplate*s are analogous and not shown here. Especially triple patterns using the square bracket syntax for blank nodes need additional normalisation rules introducing "fresh" labelled blank nodes.

3.6.2 Static type analysis of ConstructTemplates

The following rule avoids clashes of blank node labels, which could happen when creating a sequence consisting of more than one *ConstructTemplate*. Therefore we simply forbid *ConstructTemplate*s as part of sequences.

Rule 3.12. It is not allowed to construct an `RDFGraph` as part of a sequence. This applies not only to Constructed Datasets but to any occurrence of a *ConstructTemplate*.

$$\frac{\begin{array}{c} \mathit{Expr}_1, \mathit{Expr}_2 \\ \text{statEnv} \vdash \mathit{Expr}_1 : \texttt{RDFGraph}\ \textbf{or}\ \mathit{Expr}_2 : \texttt{RDFGraph} \end{array}}{\text{A static type error is raised for expression } \mathit{Expr}}$$

3.6.3 Dynamic evaluation of ConstructTemplates

During evaluation we have to ensure that the produced *Subject Predicate Object* triples are indeed valid RDF triples. Validating triples, and their RDF terms, is important, since on the subject position of an RDF triple only URIs and blank nodes are allowed. The predicate position allows only URIs and on the object position URIs, blank nodes, and RDF literals are allowed. Since a *ConstructTemplate* can create RDF terms dynamically, their validity can only be tested during dynamic evaluation.

We introduce the new function *fs:evalTemplate*; it evaluates its argument, a list of potential RDF triples given as a simple list of `RDFTerms`, where three consecutive `RDFTerms` form one triple, for validity and returns an element of type `RDFGraph`. The function checks for each given potential RDF triple if it builds a valid RDF triple. The function adds every valid triple to the final RDF graph and suppresses every invalid triple. The function's type signature is defined as follows:

fs:evalTemplate

```
fs:evalTemplate($template as RDFTerm*) as RDFGraph
```

Rule 3.13. The *fs:evalTemplate* checks its arguments for validity by using the function *fs:validTriple*. The function *fs:evalTemplate* builds a sequence by calling the function *fs:validTriple* repeatedly.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{fs:validTriple}(\mathit{Subject}_1, \mathit{Predicate}_1, \mathit{Object}_1) \Rightarrow \mathit{Triple}_1 \\ \ldots \\ \text{dynEnv} \vdash \mathit{fs:validTriple}(\mathit{Subject}_n, \mathit{Predicate}_n, \mathit{Object}_n) \Rightarrow \mathit{Triple}_n\end{array}}{\text{dynEnv} \vdash \mathit{fs:evalTemplate}\left(\begin{array}{c}(\mathit{Subject}_1, \mathit{Predicate}_1, \mathit{Object}_1 \\ ,\ldots, \\ \mathit{Subject}_n, \mathit{Predicate}_n, \mathit{Object}_n)\end{array}\right) \Rightarrow (\mathit{Triple}_1, \ldots, \mathit{Triple}_n)}$$

fs:validTriple The function *fs:validTriple* checks for every RDF term of a potential triple, if it is of an allowed type. As already outlined above: On subject position URIs and blank nodes are allowed, predicates allow URIs only, and on object position URIs, blank nodes an literals are allowed. The function is defined as follows:

```
fs:validTriple($subject as RDFTerm, $predicate as RDFTerm, object as
    RDFTerm) as RDFTriple
```

Rule 3.14. The *fs:validTriple* checks for three RDF terms if they have the correct type, depending on their position in the RDF triple.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{Subject}) \Rightarrow \mathit{ValueS} \\ \text{dynEnv} \vdash \mathit{ValueS} <: (\texttt{uri}|\texttt{bnode}) \\ \text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{Predicate}) \Rightarrow \mathit{ValueP} \\ \text{dynEnv} \vdash \mathit{ValueS} <: \texttt{uri} \\ \text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{Subject}) \Rightarrow \mathit{ValueO} \\ \text{dynEnv} \vdash \mathit{ValueS} <: (\texttt{uri}|\texttt{bnode}|\texttt{literal})\end{array}}{\text{dynEnv} \vdash \mathit{fs:validTriple}(\mathit{Subject}, \mathit{Predicate}, \mathit{Object}) \Rightarrow (\mathit{ValueS}, \mathit{ValueP}, \mathit{ValueO})}$$

Rule 3.15. If any of the three RDF terms is not of the correct type, the whole RDF triple is evaluated to the empty sequence.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{Subject}) \Rightarrow \mathit{ValueS} \\ \text{dynEnv} \vdash \mathit{Predicate} \Rightarrow \mathit{ValueP} \\ \text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{Object}) \Rightarrow \mathit{ValueO} \\ \text{dynEnv} \vdash \textbf{not}\left(\begin{array}{l}\mathit{ValueS} <: (\texttt{uri}|\texttt{bnode}) \\ \mathit{ValueP} <: \texttt{uri} \\ \mathit{ValueO} <: (\texttt{uri}|\texttt{bnode}|\texttt{literal})\end{array}\right)\end{array}}{\text{dynEnv} \vdash \mathit{fs:validTriple}(\mathit{Subject}, \mathit{Predicate}, \mathit{Object}) \Rightarrow ()}$$

```
[9] DatasetClause ::= "from" ( DefaultGraphClause | NamedGraphClause)
[10] DefaultGraphClause ::= SourceSelector
[11] NamedGraphClause ::= "named" (SourceSelector | IRIref' "(" "{"
    FLWORExpr "}" ")" )
[12] SourceSelector ::= IRIref' | "{" FLWORExpr "}" | Var
```

Listing 3.20: Syntax for Constructed Dataset

3.6.4 Constructed Dataset

The new Constructed Dataset feature allows the user to create and query intermediary RDF graphs. Constructed Dataset can be used in two different ways: A new RDF graph is created earlier and referenced as a variable, or it is created embedded in the *DatasetClause* as a nested *FLWORExpr*.

Listing 3.20 shows the syntax for the Constructed Dataset feature, modifying the original SPARQL grammar productions Section A.2.3. The production rules [9] and [10] are the original SPARQL grammar production rules. The *NamedGraphClause* (production [11]) allows an *IRIref'* followed by a *FLWORExpr* enclosed in pairs of parentheses and curly braces. The *SourceSelector* (production [12]), which is used by both the *DefaultGraphClause* and the *NamedGraphClause*, additionally allows an embedded *FLWORExpr* and a variable as well.

Rule 3.16. *SourceSelector* in Rule [10] it needs to be typed to `RDFGraph`.

$$\frac{\text{statEnv} \vdash \textit{SourceSelector} \Rightarrow \textit{Value} \quad \text{statEnv} \vdash \textit{Value} : \texttt{RDFGraph}}{\text{statEnv} \vdash \texttt{from}\ \textit{SourceSelector} : \texttt{RDFGraph}}$$

Rule 3.17. *SourceSelector* in Rule [11] needs to be typed to an `uri`.

$$\frac{\text{statEnv} \vdash \textit{SourceSelector} \Rightarrow \textit{Value} \quad \text{statEnv} \vdash \textit{Value} : \texttt{uri}}{\text{statEnv} \vdash \texttt{from named}\ \textit{SourceSelector} : \texttt{uri}}$$

3.6.5 Blank nodes

Anonymous blank nodes, i. e., blank nodes using the Turtle notation with square brackets, are handled by the $[\![\cdot]\!]_{\textit{normaliseTemplate}}$ rule in Section 3.6.1.

Special attention has to be paid to labelled blank nodes. The function *fs:bnode* converts the blank node of a graph pattern into a blank node of

a *ConstructTemplate*'s result. Since every `for` loop of XSPARQL++ contains a positional variable, we use these positional variables to "skolemise" the blank node labels, i. e., create a unique new blank node identifier for each `PatternSolution`. The function's type signature is defined as follows:

```
fs:bnode($term as RDFTerm) as RDFTerm
```

Rule 3.18. To generate a unique blank node label we skolemise the blank node with the help of the positional variables of all enclosing `for` clauses.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{RDFTerm} \Rightarrow \mathit{ValueR} \\ \text{dynEnv} \vdash \mathit{ValueR} <: \texttt{bnode} \\ \text{statEnv.posVars} = \mathit{PosVar}_1, \ldots, \mathit{PosVar}_n \\ \text{dynEnv} \vdash \mathit{fs:skolemConstant}\,(\mathit{ValueR}, \mathit{PosVar}_1, \ldots, \mathit{PosVar}_n) \Rightarrow \mathit{ValueRS}\end{array}}{\text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{RDFTerm}) \Rightarrow \mathit{ValueRS}}$$

Rule 3.19. If the argument of *fs:bnode* is an RDF term but not a blank node, it simply returns its argument.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{RDFTerm} \Rightarrow \mathit{ValueR} \\ \text{dynEnv} \vdash \textbf{not}(\mathit{ValueR} <: \texttt{bnode})\end{array}}{\text{dynEnv} \vdash \mathit{fs:bnode}(\mathit{RDFTerm}) \Rightarrow \mathit{ValueR}}$$

3.6.6 Constructed RDF Terms

Additionally to the terms and variables already allowed in a SPARQL *ConstructTemplate*, RDF terms can be built dynamically.

In analogy to the *computed constructors* of XQuery (see [Boag et al., 2007, Section 3.7.3]), XSPARQL allows *constructed RDF terms*. While the syntax for constructed RDF terms is the same as for XSPARQL, the semantics is adapted as follows.

3.6.6.1 Constructed IRIs

Normally IRIs in SPARQL are enclosed in angle brackets. To create such an IRI in XSPARQL an expression needs to be embedded in angle brackets without any white space between the angle brackets and the curly braces: `<{`*Expr*`}>` (see rule [67′] in Listing 3.17). As a second format of IRIs a Qualified Name (QName), known from XML, can be used. It consists of a namespace prefix followed by a colon and the local part of the URI. To create such a URI in XSPARQL an *EmbeddedExpression* is allowed for both parts, namespace prefix and local part, e. g., `foaf:{`*Expr*`}` (see rule [68′] in Listing 3.17).

Syntax of constructed IRIs A constructed IRI can either be created by embedding an XSPARQL expression in a pair of angle brackets and curly brackets (see Production [67'] in Listing 3.17), or by a QName while each the prefix and the local part can be dynamic with an XSPARQL expression enclosed in curly brackets (see Production [68'] in Listing 3.17).

Semantics of constructed IRIs To create correctly formatted and typed SPARQLURIs, we introduce two new constructor functions: The function *fs:iri* expects one argument which has to be a valid IRI typed as `xs:string` and returns a correctly formatted and typed SPARQLURI. A function with the same name but two arguments creates an SPARQLURI formatted as QName, while the first argument is interpreted as the namespace prefix, and the second parameter is interpreted as local name of the IRI. Both arguments have to be typed as `xs:string`. The static function signature is defined as follows: *fs:iri*

```
fs:iri($iri as xs:string) as uri
fs:iri($prefix as xs:string, $local as xs:string) as uri
```

Rule 3.20. IRIs using the angle bracket syntax.

$$[\![\texttt{<\{}FLWORExpr\texttt{\}>}]\!]_{Expr} == fn{:}iri([\![FLWORExpr]\!]_{Expr})$$

Rule 3.21. IRIs given as QName.

$$[\![\texttt{\{}FLWORExpr_1\texttt{\}:\{}FLWORExpr_2\texttt{\}}]\!]_{Expr} == fn{:}iri([\![FLWORExpr_1]\!]_{Expr}, [\![FLWORExpr_2]\!]_{Expr})$$

The rules for QNames with a static prefix and for a static local part are defined analogously (see also examples below).

Example 3.4. The following examples demonstrate simple constructed IRIs. The embedded expressions, variables or strings in the examples, could be any XSPARQL expression which eventually results in a valid IRI or QName:

```
1 let $x := "foaf"
2 construct {
3 [] a {$x}:Person ;
4    foaf:{"homepage"} <{ fn:concat("http", "://", "stefanbischof.at") }>
         ;
5    {$x}:{"name"} "Stefan Bischof" .
6 }
```

This example script results in the following RDF graph:

```
[] a foaf:Person ;
   foaf:homepage <http://stefanbischof.at> ;
   foaf:name "Stefan Bischof" .
```

3.6.6.2 Constructed Blank Nodes

To refer to blank nodes outside of the current context labelled blank nodes can be constructed by giving a prefix like `_:` immediately followed by an XQuery expression surrounded by curly braces (see rule [69'] in Listing 3.17).

Syntax of constructed blank nodes Constructed blank nodes are created by appending an XSPARQL expression enclosed in curly braces to the common blank node label prefix "_:".

Semantics of constructed blank nodes To create correctly formatted and typed RDF blank nodes, we introduce a new constructor function: The *fs:bnode* function *fs:bnode* expects one argument which will be used as label for the resulting labelled blank node and returns a correctly formatted and typed blank node. The static function signature is defined as follows:

```
fs:bnode($label as xs:string) as bnode
```

Rule 3.22. Concatenate the dynamically built blank node label to the standard blank node label prefix `_:`.

$$[\![\_:\{FLWORExpr\}]\!]_{Expr} == fn{:}bnode([\![FLWORExpr]\!]_{Expr})$$

Example 3.5. The following Listing shows an example of constructed blank nodes. The embedded *FLWORExpr'* consists in this case only of the variable `$id`.

```
let $id := "i13"
for $person in ("Alice","Bob")
construct {
_:{$id} a :Item ;
   :name "Mona Lisa" .
[] a :Person ;
   :name $person ;
   :bidsOn _:{$id} .
}
```

This XSPARQL query results in the following RDF graph:

```
_:i13 a :Item ;
   :name "Mona Lisa" .
[] a :Person ;
   :name "Alice" ;
   :bidsOn _:i13 .
_:i13 a :Item ;
   :name "Mona Lisa" .
[] a :Person ;
   :name "Bob" ;
   :bidsOn _:i13 .
```

Note that defining triples twice is no problem, the RDF graph will contain the relevant triples only once, as the RDF semantics eliminates duplicate triples[2].

3.6.6.3 Constructed Literals

Syntax of constructed literals Different RDF literals can be created at runtime using different syntaxes. The most flexible one is the usage of a standard enclosed expression using curly braces (see rule [66′] in Listing 3.17). The optional data type or language tags can also be created dynamically (see rule [60′] in Listing 3.17).

Semantics of constructed literals To create correctly formatted and typed RDF literals, we introduce three new constructor functions: The function *fs:literal* creates an RDF literal from a given string. The function *fs:literal-lang* creates an RDF literal with a language tag, and the function *fs:literal-dt* creates an RDF literal with a data type annotation. The static function signatures are defined as follows:

fs:literal *fs:literal-lang* *fs:literal-dt*

```
fs:literal($literal as xs:string) as literal
fs:literal-lang($literal as xs:string, $langtag as xs:string) as literal
fs:literal-dt($literal as xs:string, $dt as xs:string) as literal
```

Rule 3.23. Simple literals are enclosed by a pair of double quotes.

$$[\![\{FLWORExpr\}]\!]_{Expr} == fn{:}literal([\![FLWORExpr]\!]_{Expr})$$

Rule 3.24. A constructed literal with a language tag is built by concatenating both parts together with an '@' sign in between.

[2]Strictly speaking, our `RDFGraph` data type represents multisets of triples rather than actual RDF graphs, but we assume that RDF tools consuming the output of XSPARQL engines will take care of this duplicate elimination.

$$[\![\{FLWORExpr_1\}@\{FLWORExpr'_2\}]\!]_{Expr}$$
$$==$$
$$fn{:}literal\text{-}lang([\![FLWORExpr_1]\!]_{Expr}, [\![FLWORExpr_2]\!]_{Expr})$$

Rule 3.25. The same applies to a constructed literal with a data type annotation, while the delimiter is a double hat ('^^') symbol.

$$[\![\{FLWORExpr_1\}\hat{}\hat{}\{FLWORExpr'_2\}]\!]_{Expr}$$
$$==$$
$$fn{:}literal\text{-}dt([\![FLWORExpr_1]\!]_{Expr}, [\![FLWORExpr_2]\!]_{Expr})$$

The rules for a static literal part, static language tag, and static data type annotation are similar and thus not shown here.

Example 3.6. The example listing shows three different types of constructed literals: a plain literal (line 3), a literal with a language tag (line 4) and a literal typed as decimal (line 5):

```
1 let $x := "ice"
2 construct {
3 [] foaf:name {fn:concat("Al",$x)} ;
4    foaf:name "Alice"@{"en"} ;
5    foaf:age { 3 + 7 }^^xsd:decimal .
6 }
```

This XSPARQL query results in the following RDF graph:

```
1 [] foaf:name "Alice" ;
2    foaf:name "Alice"@en ;
3    foaf:age "10"^^xsd:decimal .
```

3.7 SPARQL Filter Operators

SPARQL includes several functions available in filter expressions of a *WhereClause* [Prud'hommeaux and Seaborne, 2008, Section 11]. In XSPARQL when used in a *SparqlForClause* they are evaluated by *fs:sparql*. In XSPARQL these functions are available as standard XQuery functions with the following type declarations:

```
bound($A as RDFTerm) as xs:boolean
isiri($A as RDFTerm) as xs:boolean
isblank($A as RDFTerm) as xs:boolean
isliteral($A as RDFTerm) as xs:boolean
lang($A as literal) as xs:string
datatype($A as literal) as xs:string
```

In contrast to the XSPARQL definition, the function names are given in lowercase only in XSPARQL++. In contrast to the SPARQL specification which defines that the functions *bound*, *isiri*, *isblank*, and *isliteral* return an error for an unbound argument, we decided to return `fn:false` in that case.

The semantics of these functions is defined by the following rules, replacing the Rules 2.30 to 2.35.

Rule 3.26. The function *bound* returns `fn:true` if its argument, a variable is bound to a value, and `fn:false` otherwise.

$$[\![bound(\$VarName)]\!]_{Expr} \\ == \\ [\![fn{:}empty(\$VarName)]\!]_{Expr}$$

Rule 3.27. The function *isiri* returns `fn:true` if its argument, a variable is of type `uri`, and `fn:false` otherwise.

$$[\![isiri(\$VarName)]\!]_{Expr} \\ == \\ [\![\$VarName \texttt{ instance of uri}]\!]_{Expr}$$

Rule 3.28. The function *isblank* returns `fn:true` if its argument, a variable is of type `blank`, and `fn:false` otherwise.

$$[\![isblank(\$VarName)]\!]_{Expr} \\ == \\ [\![\$VarName \texttt{ instance of bnode}]\!]_{Expr}$$

Rule 3.29. The function *isliteral* returns `fn:true` if its argument, a variable is of type `literal`, and `fn:false` otherwise.

$$[\![isliteral(\$VarName)]\!]_{Expr} \\ == \\ [\![\$VarName \texttt{ instance of literal}]\!]_{Expr}$$

Rule 3.30. The function *lang* returns returns the language tag of a literal containing a language annotation. If the literal contains no language tag, the empty string is returned.

$$[\![lang(\$VarName)]\!]_{Expr} \\ == \\ [\![fn{:}string(\$VarName/@xml{:}lang)]\!]_{Expr}$$

Rule 3.31. The function *datatype* returns the data type of a typed literal.

$$[\![datatype(\$VarName)]\!]_{Expr}$$
$$==$$
$$[\![fn{:}string(\$VarName/@datatype)]\!]_{Expr}$$

Appendix D.1 contains a formal description of semantic properties of XSPARQL++, i. e., the relation between XSPARQL++ and XQuery, as well as the relation between XSPARQL++ and SPARQL.

3.8 Query Evaluation Examples

We give two examples to show how the two main extensions of XQuery which lead to XSPARQL work. In the *Lifting* example we will present the evaluation of a *SparqlForClause* and in the *Lowering* example the evaluation of a *ConstructClause*.

3.8.1 XML to RDF

The lowering example query iterates over a sequence of names, and creates a person with a name for each of them.

```
1 for $name in ("Alice","Bob")
2 construct { _:b a foaf:Person ;
3                 foaf:name {$name} . }
```

Normalisation

For the normalisation first Rule 3.10 is applied, introducing a calls to the function *fs:evalTemplate*. The `$name` variable occurs as a constructed literal and is therefore rewritten by Rule 3.23 to an embedded call to *fn:literal* enclosing the `$name` variable in a pair of double quotes. The whole template is normalised by the $[\![\cdot]\!]_{normaliseTemplate}$ rule.Moreover the *ForClause* is decorated with a positional variable as specified by Rule 2.24.

This rewriting results in the following query (for easier readability we resign from using all the needed type conversions):

```
1 for $name at $_name_pos in ("Alice","Bob")
2 return fs:evalTemplate((
3  _:b, a, foaf:Person,
4  _:b, foaf:name, fn:literal($name) ))
```

Static typing

During XQuery static typing the individual `RDFTerms` will get their correct types and for every triple the *fs:validTriple* will be called.

Dynamic evaluation

The evaluation result is constructed by the Rule 2.12. The name strings of the sequence are assigned to the `$name` variable sequentially, while the `$_name_pos` variable is 1 for the first iteration, and incremented for each following iteration. The blank node label is created by calling the function *fs:skolemConstant*. We assume that this function works by appending an

underscore character and the value of the variable `$_name_pos` to the given blank node label.

This results in the following RDF graph in Turtle notation:

```
_:b_1 a foaf:Person .
_:b_1 foaf:name "Alice" .
_:b_2 a foaf:Person .
_:b_2 foaf:name "Bob" .
```

3.8.2 RDF to XML

The following example query gets the names of all persons in an RDF graph and returns them as an XDM sequence. The RDF graph used for this query is given in Listing 2.2 on page 14.

```
for * from <relations.rdf>
where { [] a foaf:Person ;
          foaf:name $name . }
return
  $name
```

First Rule 3.1 is applied, replacing the star in the *SparqlForClause* by the only variable occurring in the *WhereClause*: `$name`. Normalisation

```
for $name from <relations.rdf>
where { [] a foaf:Person ;
          foaf:name $name . }
return
  $name
```

In the static type analysis step the whole *SparqlForClause* is checked for correct typing. Using Rule 3.5 we can infer that the value of variable `$name` will be of type or subtype `RDFTerm`. Since `$name` is not bound earlier *fs:sparql* replaces the variable in the *WhereClause* with its name as string. Static type analysis

Thus the function *fs:sparql* will be called with the following parameters:

```
fs:sparql(($name), "[] a foaf:Person ; foaf:name $name . ", "")
```

The first step in Rule 3.6 is to evaluate the *fs:sparql* function to a sequence of SPARQL solution mappings. According to the definition of *fs:sparql* its parameters are evaluated equivalently to the following SPARQL query: Dynamic evaluation

```
select $name from <relations.rdf>
where { [] a foaf:Person ;
          foaf:name $name . }
```

The *fs:sparql* function returns the following three solution mappings (order is irrelevant):

#	$name
1	“Alice”
2	“Bob”
3	“Charles”

Next the *ReturnExpr* is evaluated once for every solution mapping while the value of the (XQuery) `$name` variable is set to the corresponding variable binding of the current mapping. The *ReturnExpr* consists of the variable only, thus the evaluation result is the value of the `$name` variable.

The result of the whole query is then given by the sequence of evaluation results of the *ReturnExpr*s. Therefore the following sequence is a possible result of the whole query[3]:

```
("Alice","Bob","Charles")
```

[3] other results may differ in the order of the sequence

Chapter 4

Implementation of XSPARQL

Based on the semantics presented in the last chapter, we have built a new implementation[1] capable of showing the advantages of the Dataset Scoping and Constructed Dataset features and the XSPARQL language in general. In this chapter we present our new implementation, especially how we solved some practical problems.

The terms *prior implementation* and *specification implementation* in this section refer to the implementation of [Lopes et al., 2009]. The terms *current implementation*, *presented implementation* or just *the implementation* mean the new one presented in this work.

We first describe the implementation requirements shortly and then give an overview of the implementation architecture. After that we explain the main part of the implementation, the query rewriter, in detail. Finally two examples show how the rewriter works in practice.

4.1 Requirements

The most important requirements are given in the following list ordered by decreasing priority:

Specification Conformance The main goal of the new implementation is to show the capabilities of XSPARQL in practice. Therefore the implementation should be compliant with the XSPARQL++ specification presented in the previous chapter.

Optimisation The implementation should create and use data structures appropriate for easy implementation of new optimisations.

[1] a demo is available at `http://xsparql.deri.org/demo`

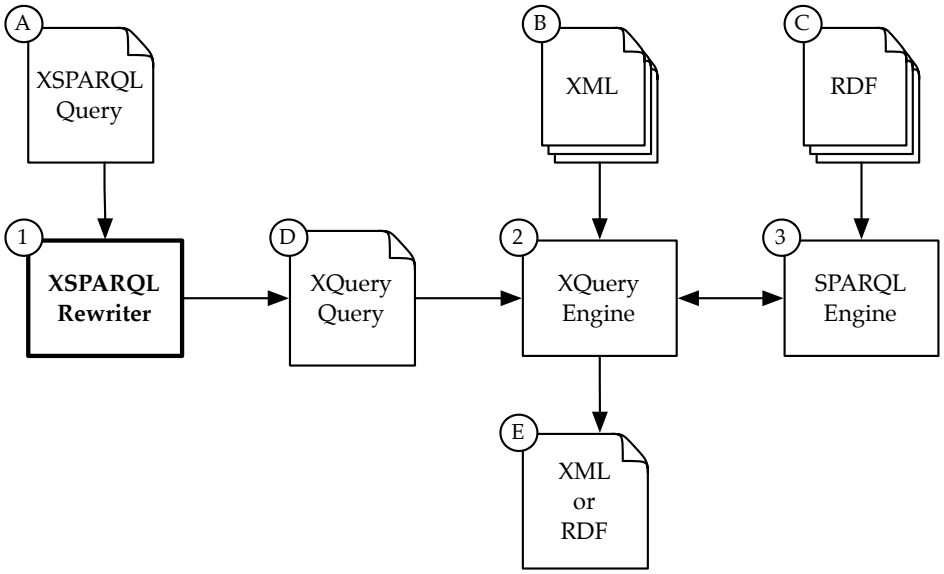

Figure 4.1: XSPARQL architecture

Maintainability The implementation should be better maintainable than the prior implementation thus enabling further development.

Platform Independence The implementation should run on different platforms while keeping implementation effort to a minimum.

API The implementation should provide a simple API to be not only usable as command line application but also in Web or GUI applications.

4.2 Overview

The presented implementation uses a similar architecture as the previous implementation as shown in Figure 4.1. The numbers and letters of the following list refer to the ones in Figure 4.1.

When evaluating an XSPARQL query, first the XSPARQL query Ⓐ is rewritten to an XQuery query Ⓓ by the XSPARQL *query rewriter* ①. This rewriting is executed completely statically thus no access to input data is needed and query evaluation can be delayed to any time in the future.

Next the query is evaluated by an XQuery engine ② such as Saxon [Kay]. XML data Ⓑ is processed directly by the XQuery engine. A standard SPARQL implementation ③ such as Joseki [Joseki] is used for evaluating the embedded SPARQL queries. The XQuery engine communicates with the SPARQL engine via HTTP by encoding the SPARQL query in a URL. The SPARQL engine

evaluates the query on RDF data © and returns the result in the SPARQL query results XML format [Beckett and Broekstra, 2008]. Since the query evaluation result is in an XML format, it is processed by XQuery straightforwardly afterwards. In the end, the XQuery engine emits the query result Ⓔ, be it XML or RDF Turtle Syntax or another natively supported XQuery data format.

Components

The XSPARQL query rewriter is written in Java thus enabling usage on several different platforms. Since the rewriter is a self-contained program, it is possible to store the resulting XQuery query and evaluate it independently at any other time, e. g., if input data and XSPARQL rewriter are not available in the same environment or to use the same query on several data sources. For instant query evaluation we integrated the Saxon library. By this measure queries can be evaluated straightforwardly. For the SPARQL HTTP endpoint, an HTTP server accepting a SPARQL query as parameter and returning the evaluation result to the HTTP client, we used Joseki.

4.2.1 XSPARQL Rewriter Overview

Internally the XSPARQL query rewriter implementation works by running sequentially through six phases:

1. The *Lexer* creates a token stream from the XSPARQL query string.
2. The *Parser* creates an XSPARQL Abstract Syntax Tree (AST) representing the original query. An AST is a condensed version of a full parse tree.
3. We perform pre-rewriting optimisation.
4. The *Rewriter* translates XSPARQL AST to XQuery AST.
5. We perform post-rewriting optimisation (pure XQuery optimisation).
6. The *Serializer* emits XQuery query as string for further processing.

4.2.1.1 Lexer & Parser (steps 1 and 2)

We are using a JFlex [JFlex] generated *Lexer*. The Lexer produces a stream of *tokens* out of the XSPARQL query string. It uses different internal states for the XQuery and the SPARQL query parts and thus returns different tokens depending on the context.

The ANTLR parser generator [Parr, 2007] creates a parser using an LL(*) parsing strategy from an XSPARQL grammar written in a similar syntax as the XSPARQL grammar in Appendix A. The parser produces an *Abstract Syntax Tree* (AST) representation of the query from the token stream of the lexer.

4.2.1.2 Optimisation (steps 3 and 5)

In theses steps the rewriter can optimise the parsed query by applying heuristic rules to the AST. Chapter 5 introduces one such optimisation in detail.

4.2.1.3 Rewriting (step 4)

The core rewriting step takes an XSPARQLAST and converts it to an XQueryAST. Section 4.3 describes this step in more detail, as it is the most complex step and the core of the XSPARQL rewriter.

4.2.1.4 Serialisation & Query Evaluation (step 6)

The parsing/rewriting process results in a pure XQueryAST. This AST is eventually converted to a string (serialisation). This string can then be stored in a file or directly evaluated.

4.3 Rewriting XSPARQL to XQuery

Unlike the prior implementation, the core of the new implementation is not based on a monolithic recursive rewrite function but on a set of rewriting rules dependent on the used constructs. The rewriter chooses the rules to apply not only by examining the tree but by using context information, such as variable scopes, too. ANTLR provides a concise syntax to specify such rewriting rules thus resulting in a maintainable solution.

As the semantics definition in Chapter 3, this section is also divided in two major parts: one handling the *SparqlForClause* used for lowering tasks and the other describing the *ConstructClause* used for lifting tasks.

4.3.1 Lowering – *SparqlForClause*

The *SparqlForClause*, including a SPARQL *WhereClause*, is rewritten to a standard SPARQL `select` query and evaluated via the following steps (this procedure follows the semantics definition in Section 3.5 closely):

1. Replace already bound variables occurring in the SPARQL *WhereClause* by their values. Convert values to `RDFTerms` if needed.

2. Create a SPARQL `select` query using the variables of the `for` clause for projection, the SPARQL *WhereClause*, and the optional solution modifiers.

3. Evaluate the built query by encoding it in a URL and sending it to a SPARQL endpoint.

4. Iterate over the returned bindings and assign the results to the corresponding XQuery variables.

After these four steps and evaluation of the resulting SELECT query at the SPARQL endpoint, the variables used in the SPARQL *WhereClause* are available as plain XQuery variables in the *ReturnClause*. The new variables are automatically typed as `RDFTerms`.

Here *fs:sparqlCall()* shall be viewed as an implementation of the abstract *fs:sparql* function used in the semantics definition in the previous chapter. It is slightly different (having different parameters), thus it has a different name.

The following overview rule shows the rewriting for a *SparqlForClause*:

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \texttt{for}\ \$VarName_1 \cdots \$VarName_n\ DatasetClause \\ \texttt{where}\ GroupGraphPattern\ SolutionModifier \\ ReturnClause \end{array} \right]\!\!\right]_{Expr} \\
\qquad == \\
\texttt{let}\ \$\_aux\_queryresult := fs{:}sparqlCall \left(\begin{array}{l} \$VarName_1 \cdots \$VarName_n, \\ DatasetClause, \\ GroupGraphPattern, \\ SolutionModifier \end{array} \right) \\
\texttt{for}\ \$\_aux\_result\ \texttt{at}\ \$\_aux\_result\_pos\ \texttt{in}\ \$\_aux\_queryresult \\
\quad [\![\$VarName_1 \cdots \$VarName_n]\!]_{SparqlResult(\$\_aux\_result)} \\
\quad [\![ReturnClause]\!]_{Expr}
\end{array}
$$

The auxiliary mapping rule $[\![\cdot]\!]_{SparqlResult(\$result)}$ binds the variables of the current solution mapping to XQuery variables for every iteration by "parsing" the SPARQL result format with suitable XPath expressions.

$$
\begin{array}{l}
[\![\$VarName_1 \cdots \$VarName_n]\!]_{SparqlResult(\$result)} \\
\qquad == \\
\texttt{let}\ \$VarName_1 := \$result/\_sparql\_res{:}binding[@name = \$VarName_1]/* \\
\qquad \cdots \\
\texttt{let}\ \$VarName_n := \$result/\_sparql\_res{:}binding[@name = \$VarName_n]/*
\end{array}
$$

Next we explain the rewriting for the various clauses, performed by the *fs:sparqlCall()* function.

To create a SPARQL `select` query, first all variables in the SPARQL *WhereClause* need to be examined. According to the semantics specification there are three different cases for every variable occurring at any position: Where Clause

1. The variable is previously unbound;

2. The variable is bound by a *SparqlForClause*;

3. The variable is bound by any standard XQuery expression.

To differentiate between these cases the implementation needs to know statically which variables were previously bound and if they were created by another *SparqlForClause* or by a plain XQuery *ForClause* or *LetClause*. We implemented this behaviour as a stack of variable binding scopes using ANTLR's *global dynamic attribute scopes* (see [Parr, 2007, Section 6.5]), with variables bound by a *SparqlForClause* marked accordingly.

Each variable occurring in the SPARQL *WhereClause* is processed depending on whether and how it was bound:

- If the variable was bound by a *SparqlForClause*, the value of the variable is transformed to the Turtle style string of the corresponding RDF term representation before replacing the variable.
- If this variable was bound by a standard *ForClause* or *LetClause*, the variable is replaced by its value.
- If the variable was not bound earlier, it is left untouched. In this case the SPARQL engine is supposed to bind it.

Dataset Clause

The *SourceSelector* of a *DatasetClause* can either be an IRI, as in standard SPARQL, or a variable. IRIs are translated to the SPARQLIRI representation by surrounding it with angle brackets.

If a variable is used as *SourceSelector* and it is not of type `RDFGraph`, it is assumed that the variable evaluates to an IRI (not only `xs:anyURI` but also `xs:string` could be used here), which additionally is accessible by the SPARQL engine and returns a valid RDF graph.

If the variable is of type `RDFGraph`, i. e., the variables value is a string representation of an RDF graph, Constructed Dataset, a feature introduced in Section 3.1, is used. It is not possible to pass an RDF graph as dataset directly (inline) in a SPARQL query. Therefore a temporary file containing the RDF graph is created and an IRI is assigned to it. This IRI is then passed to the SPARQL engine in the *DatasetClause*.

Since XQuery and its functions are free of side effects, there exists no standard function to create temporary files. Hence we provide a implementation dependent XQuery extension function *turtleGraphToURI* to create a temporary file containing its first parameter and return the associated IRI. For this approach to work two conditions must be fulfilled:

1. The Saxon XQuery engine is used because the extension mechanism currently used is implementation dependent.
2. To keep the overhead and minimise dependencies a local file URI is generated. This constraint demands that the XQuery processor is called

on the same machine as the SPARQL engine. Future implementations might loosen this restriction by providing the RDF graph file for download by using some network protocol or by inserting the graph into an RDF store using the SPARQL 1.1 update protocol [Schenk et al., 2010] which is currently being standardised..

Solution modifier

In a *SparqlForClause* only SPARQL solution modifiers and no XQuery solution modifiers are allowed. Since only previously unbound variables are allowed in a SPARQL solution modifier, the solution modifier part can be copied to the rewritten SPARQL query verbatim.

Evaluation

The so built query string is then passed to a SPARQLHTTP service which evaluates the query and returns the variable bindings using the SPARQL query results XML format [Beckett and Broekstra, 2008]. While iterating over the results new XQuery variables, reusing the variable names of the XSPARQL select clause and, are created for each result set. These variables contain the values returned by the SPARQL engine while retaining type information of the SPARQL query results XML format.

4.3.2 Lifting – SPARQL *ConstructClause*

As mentioned earlier, XSPARQL allows to emit RDF in Turtle notation using a *ConstructClause* instead of a standard XQuery *ReturnClause* (see Section 3.6). The query rewriter translates the *ConstructClause* to a standard *ReturnClause*. This newly created *ReturnClause* ensures at evaluation time, that only valid RDF triples are generated.

Since `RDFTerm` is no default data type of XDM, an XQuery has to use an already existing output format. We decided to create a sequence of strings from a *ConstructClause* which builds up to an RDF graph in the end.

To ensure that a valid Turtle RDF graph is produced, every term of each triple must be valid for its position in the triple, cf. Section 3.6.3 above.

4.3.2.1 Plain SPARQL Variables

In our implementation, the `_rdf_term` function creates an RDF string representation of a variable value, bound by a *SparqlForClause*, depending on the data type. The namespace prefix *sr* stands for the SPARQL Query Results XML Format namespace defined in [Beckett and Broekstra, 2008].

$$[\![\_rdf\_term(\$VarName)]\!]_{Expr}$$
$$==$$

```
typeswitch ($VarName)
  case $e as schema-element(literal)
    let $DT := data($e/@datatype)
    let $L := data($e/@xml:lang) return fn:concat("""",$e,
      if($L) then fn:concat("@",$L) else "",
      if($DT) then fn:concat("^^<",$DT,">") else "",
      """")
  case $e as schema-element(bnode) return fn:concat("_:",$e)
  case $e as schema-element(uri) return fn:concat("<",$e,">")
  default return ""
```

Plain variables in a *ConstructClause* are only allowed if bound by a *SparqlForClause*, i. e., variables of type `RDFTerm`, since RDF triples can only consist of RDF terms.

4.3.2.2 Constructed RDF Terms

A second set of objects besides variables which have to be validated are constructed RDF terms. A *literal construct* is an enclosed FLWOR expression which evaluates to an RDF literal. To ensure correct typing the constructor function *_xsparql:literal* transforms the input string to a value of type `literal` by escaping any double quotes (using the auxiliary function *_xsparql:escapequotes*) and then, enclosing the resulting string in a pair of double quotes.

_xsparql:literal

```
1 _xsparql:literal($argument as xs:string) as literal {
2   validate { <literal>{_xsparql:escape-quotes($argument)}</literal> }
3 }
```

The result of this enclosed expression is first assigned to a temporary variable. If this variable is a valid literal then the triple is returned (depending on the other RDF terms in the triple). XQuery tests the constructed RDF term to be a valid literal before building the triple by using the function *_xsparql:validLiteral*.

_xsparql:validLiteral

```
1 let $_temp := _xsparql:literal(Expr)
2 return if(_xsparql:validLiteral($_temp)) then
3   subject predicate $temp
4 else ""
```

The same construct is used for literals with a language tag or a data type annotation. For these cases new XQuery constructors *_xsparql:literallang* and *_xsparql:literaldt* are used. These two functions are defined accordingly and have the same static type signature as the corresponding constructor functions in the semantics definition of XSPARQL++.

A constructed IRI is created by enclosing expression in angle brackets. This task is performed by the *_xsparql:iri* constructor function. As the other constructor functions it is defined analogously to the corresponding constructor functions of the XSPARQL++ semantics in the last chapter.

Before building the triple XQuery validates the constructed RDF term to be a valid IRI by using the function *_xsparql:validIRI*.

```
1 let $_temp := _xsparql:iri(Expr)
2 return if(_xsparql:validIRI($_temp)) then
3   subject predicate $temp
4 else ""
```

Constructed blank nodes are created in a similar way by appending the enclosed expression to the static blank node prefix. It is defined analogously to the corresponding constructor functions of the XSPARQL++ semantics in the last chapter.

Blank nodes are tested for validity before building the triple. In this case the validate function is called *_xsparql:validBNode*

```
1 let $_temp := _xsparql:bnode(Expr)
2 return if(_xsparql:validBNode($_temp)) then
3   subject predicate $temp
4 else ""
```

4.3.2.3 Blank Nodes

As described in semantics specification in Section 3.6.5 the two main categories of blank nodes are *labelled blank nodes* and *anonymous blank nodes*. Both have their own issues which are handled in the implementation similar to the rewriting rules in the semantics specification.

Labelled Blank Nodes These are treated carefully by adding position variables from the surrounding `for` expressions. The different position variables are separated by the underscore character. Rule 3.18 on page 68 shows the rule used in the implementation.

Anonymous Blank Nodes In contrast to the semantics definition of the new language XSPARQL++ in the last chapter, the implementation handles anonymous blank nodes directly.When using anonymous blank nodes the issue of unique blank node labels vanishes. But instead the possibility of a *VerbObjectList* embedded in such an anonymous blank node leads to another issue discussed below.

The rewriting depends on the position of the blank node: subject or object.

Subject Position Anonymous blank nodes on subject position are reduced to the Turtle serialisation of an empty pair of square brackets.

```
1 let $_verbsobjects := generate a sequence of verb-objectlist pairs
2 return
3   if(fn:empty($_verbsobjects)) then
4     ""
5   else
6     "[]", $_verbsobjects, " . "
```

The `let` expression on line 1 generates a sequence of verb-objectlist pairs separated by the semicolon character. This sequence is assigned to a new temporary variable as string representation. See example in the next section for an example on how to create this sequence.

If this sequence is empty, i. e., no valid verb-objectlist could be generated, the whole triple is invalid and the empty string is returned. If this sequence is not empty, i. e., at least one valid verb-objectlist pair could be generated, it is returned as a string with a prefixed empty anonymous blank node.

This procedure ensures that no blank node stands on its own without a valid predicate or subject.

Combined Object/Subject Position A blank node on object position is *always* valid. For the depending predicate-object pairs we use the standard way of printing, while the only difference is the usage of a semicolon as separator instead of the dot.

```
subject predicate "["(predicate object ";")* "]"
```

4.3.3 Dataset Scoping

Dataset Scoping as described in Section 3.2 allows us to join variables over nested SPARQL queries that potentially bind to blank nodes, without leading to unintended behaviour.

One approach to implement this feature in a XSPARQL++ rewriter is to execute the different SPARQL queries in a single query using `union`. To make sure the results are the same and to distinguish the results of the different queries, it is necessary to *qualify* the variable names. One possibility is be to prefix every variable with an identifier unique to each query, e. g., `$query1_var1`, `$query2_var1` etc. In our implementation only variables shared by outer and inner query are qualified with an `_inner` postfix instead.

When using this approach the XQuery engine will get all the results for both, the inner and the outer SPARQL query in a single call to the SPARQL endpoint. Note that this change can actually be perceived as an optimisation similar to the one presented in Chapter 5. The join itself is then performed by XQuery alone in a XQuery *WhereClause*.

```
1  import module namespace _xsparql = "http://xsparql.deri.org/XSPARQLer/
       xsparql.xquery" at "http://xsparql.deri.org/XSPARQLer/xsparql-types.
       xquery";
2  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
3  <relations>
4  {
5    let $_aux_results0 := _xsparql:_sparql(("PREFIX foaf: <http://xmlns.
         com/foaf/0.1/>
6      SELECT $Person $Name $Person_inner $FName
7      FROM <http://stefanbischof.at/xsparql/relations.ttl>
8      WHERE {
9        { $Person foaf:name $Name .  }
10       union
11       { $Person_inner foaf:knows [ foaf:name $FName ] . }
12     }"))
13   for $_aux_result0 at $_aux_result0_pos in _xsparql:_sparqlResults(
         $_aux_results0)
14   let $Person := _xsparql:_resultNode( $_aux_result0, "Person" )
15   let $Name := _xsparql:_resultNode( $_aux_result0, "Name" )
16   return <person name="{$Name}">
17   {
18     for $_aux_result1 at $_aux_result1_pos in _xsparql:_sparqlResults(
           $_aux_results0)
19     let $Person_inner := _xsparql:_resultNode( $_aux_result1, "
           Person_inner" )
20     let $FName := _xsparql:_resultNode( $_aux_result1, "FName" )
21
22     (: join dependent variables :)
23     where $Person = $Person_inner
24     return <knows> {fn:data( $FName )}</knows>}
25   </person>
26 }
27 </relations>
```

Listing 4.1: Implementation of FOAF lowering using Dataset Scoping

Using this approach the example of Dataset Scoping in Listing 3.10 on page 54 is rewritten as shown in Listing 4.1. Instead of evaluating each rewritten SPARQL query on its own, the Dataset Scoping implementation merges the two SPARQL *WhereClause*s by a `union`.

The SPARQL query returns a single list of variable bindings, while the variables `$Person` and `$Name` belong to the former outer query, and the inner variables `$Person_inner` and `$FName` belong to the inner query.

To make the distinction between the two queries clearer: every solution set contains either variables of the outer query or variables of the inner query.

4.4 Rewriting Examples

In this section we explain the rewriting of two XSPARQL queries. One using the *SparqlForClause* and one using a *ConstructClause*.

4.4.1 RDF to XML–Lowering

In this section we show the rewriting of an XSPARQL query by using the query in Listing 2.15 on page 37 as an example. The resulting XQuery after rewriting is shown in Listing 4.2.

Line 1 of the prolog in Listing 4.2 imports a library of XQuery functions needed later. Line 2 declares the namespace needed when using the function *turtleGraphToURI* (not used in this example) and line 3 declares the FOAF namespace.

In the lines from 6 to 10 the outer SPARQL query is first constructed as a string and then passed to the *fs:sparqlCall()* function. This function encodes its argument to a URL and performs an HTTP GET request on a SPARQL endpoint. The evaluation result is then assigned to the `$_aux_results3` variable.

The `for` loop in line 11 iterates over the returned solution sets. In lines 13 and 15 the two variables of the *SparqlForClause* are introduced as XQuery variables, using the data of the current iteration, i. e., `$_aux_result3`.

After creating a person element containing the name, the inner SPARQL query is evaluated using the same mechanism as before. But this time not all the variables are inserted as string into the query, but the join variables, `$Person` and `$Name`, are replaced by their value piped through the *_rdf_term* function. This function ensures that the returned value is the correct format for SPARQL depending on the type.

After the SPARQL engine returned the results, a `for` loop iterates over them.

The remaining SPARQL variable `$FName` is bound to an XQuery variable and eventually the `knows` element containing the XML value of that variable is returned. The query is ended by XML end tags of person and relations.

4.4.2 XML to RDF–Lifting

For the lifting task we use an example query presented by [Polleres et al., 2009] shown in Listing 4.3. This query creates a FOAF RDF graph by processing an XML file. An RDF triple is created for every `knows` element of the source XML file. Although it is only a naive implementation of a lifting task, it demonstrates how *ConstructClause*s are rewritten by XSPARQL. Listing 4.4 shows the result when evaluating the query in Listing 4.3 on the XML document in Listing 1.1 on page 3. The rewritten query is shown in Listing 4.5.

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/
       xsparql.xquery" at "http://xsparql.deri.org/demo/xquery/xsparql-
       types.xquery";
2  declare namespace _javaSaxon = "java:org.deri.sparql.Sparql";
3  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
4  <relations>
5  {
6  let $_aux_results3 := _xsparql:_sparql( fn:concat( "PREFIX foaf: <http
       ://xmlns.com/foaf/0.1/>
7     ", " SELECT ", "$Person ", "$Name ", "
8     from ", "<http://xsparql.deri.org/data/relations.rdf>", "
9     where ", " { ", "$Person", " ", "foaf:name", " ", "$Name", " .
10    ", " } ", "order by ", "$Name " ) )
11 for $_aux_result3 at $_aux_result3_pos in _xsparql:_sparqlResults(
       $_aux_results3 )
12
13 let $Person := _xsparql:_resultNode( $_aux_result3, "Person" )
14
15 let $Name := _xsparql:_resultNode( $_aux_result3, "Name" )
16 return
17   <person name="{$Name}">
18   {
19   let $_aux_results5 := _xsparql:_sparql( fn:concat( "PREFIX foaf: <http
         ://xmlns.com/foaf/0.1/>
20     SELECT $FName
21     from <http://xsparql.deri.org/data/relations.rdf>
22     where  { ", _xsparql:_rdf_term( $Person ), " foaf:knows $Friend .
23     ", _xsparql:_rdf_term( $Person ), " foaf:name ", _xsparql:_rdf_term(
           $Name ), " .
24       $Friend foaf:name $FName .
25     } " ) )
26   for $_aux_result5 at $_aux_result5_pos in _xsparql:_sparqlResults(
         $_aux_results5 )
27   let $FName := _xsparql:_resultNode( $_aux_result5, "FName" )
28   return
29     <knows> {fn:data( $FName )}</knows>
30   }
31   </person>
32 }
33 </relations>
```

Listing 4.2: Rewriting of FOAF lowering

```
declare namespace foaf = "http://xmlns.com/foaf/0.1/";
for $person in doc("relations.xml")//person,
    $nameA in $person/@name,
    $nameB in $person/knows
construct
{
[ foaf:name {data($nameA)}; a foaf:Person ]
foaf:knows
[ foaf:name {data($nameB)}; a foaf:Person ].
}
```

Listing 4.3: Naive FOAF lifting

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
[ foaf:name "Alice" ;
  a foaf:Person ;
  foaf:knows [  foaf:name "Bob" ;
                a foaf:Person ] ] .
[ foaf:name "Alice" ;
  a foaf:Person ;
  foaf:knows [  foaf:name "Charles" ;
                a foaf:Person ]  ] .
[ foaf:name "Bob" ;
  a foaf:Person ;
  foaf:knows [  foaf:name "Charles" ;
                a foaf:Person ]  ] .
```

Listing 4.4: Result of the query in Listing 4.3

The first three of lines of Listing 4.5 the rewritten query consist of namespace declarations and the XSPARQL module import, as the lowering example before.

The first result string is an RDF namespace declaration in Turtle notation. This is needed for the resulting RDF graph.

The following three `for` loops iterate over all the `knows` children elements of all `person` elements, using a similar syntax as the original query.

Since the *ConstructClause* contains a anonymous blank node at subject position, the final result is retrieved in two steps: first a sequence of verb-objectlists is created, then, if not empty, it is returned with an empty anonymous blank node on subject position.

The first triple contains a constructed literal at object position. After evaluating it and assigning its value to the `$_rdf2` variable, it is tested for validity. If it is a valid object, a sequence of the verb, `foaf:name`, and the object, `$_rdf2`, separated by spaces and terminated by a semicolon is returned.

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/
       xsparql.xquery" at "http://xsparql.deri.org/demo/xquery/xsparql-
       types.xquery";
2  declare namespace _javaSaxon = "java:org.deri.sparql.Sparql";
3  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
4
5  "@prefix foaf: <http://xmlns.com/foaf/0.1/> . ",
6  for $person at $person_pos in doc( "http://xsparql.deri.org/data/
       relations.xml" )//person
7  for $nameA at $nameA_pos in $person/@name
8  for $nameB at $nameB_pos in $person/knows
9
10 return
11   _xsparql:_serialize( (
12   let $_rdf0 := (
13     let $_rdf2 := _xsparql:_binding_term( data( $nameA ) )
14     return
15       if (_xsparql:_validObject( $_rdf2 )) then
16         ("", " ", "foaf:name", " ", _xsparql:_rdf_term( $_rdf2 ), " ; ")
17       else "",
18       "", " ", "a", " ", "foaf:Person", " ; ", ("", " ", "foaf:knows", "
             [ ", (
19       let $_rdf8 := _xsparql:_binding_term( data( $nameB ) )
20       return
21         if (_xsparql:_validObject( $_rdf8 )) then
22           ("", " ", "foaf:name", " ", _xsparql:_rdf_term( $_rdf8 ), " ;
                 ")
23         else "",
24         "", " ", "a", " ", "foaf:Person", " ; ")[fn:position() lt fn:
               last()], " ] ", " ; "))
25   return
26     if (fn:not( fn:empty( $_rdf0 ) )) then
27         ("[", $_rdf0[fn:position(  ) lt fn:last(  )], " ] .&#xA;")
28     else ""
29   ) )
```

Listing 4.5: Rewriting of naive FOAF lifting

Otherwise the return value is the empty string.

Next the static verb-object pair declaring the subject node as being of type `foaf:Person` is returned. Since object term of the `foaf:knows` property is already valid, the opening square bracket can be printed safely.

The second constructed literal, creating the name of the second person, is evaluated as before.

For this second person, the static `rdf:type` relation (using the shorthand `a`), is returned too. The predicate `[fn:position() lt fn:last()]` removes the semicolon at the end of the last valid verb-objectlist pair, which is in this case `a foaf:Person ;`. Eventually the closing square bracket, for the anonymous blank node opened at line 16, is added.

Eventually the XQuery processor returns the final triple, but only if the sequence of verb-objectlists is not empty. The last semicolon is removed again.

Chapter 5

Query Optimisation

In the last chapter we presented our new implementation for the XSPARQL++ language. Our rewriting causes, in case of nested loops, potentially many interleaved calls to a SPARQL endpoint, which is probably not very efficient and particularly hard to deal with for built-in XQuery query optimisers.

In this chapter we introduce XDEP, an optimisation of XSPARQL++ queries containing nested *SparqlForClause*s. In the end we discuss some other possible optimisations.

5.1 Query Optimisation in General

Query languages such as SQL and SPARQL have a declarative syntax and semantics (based on relational algebra) and are thus independent of physical data representation. To answer a query although, the evaluation engine has to access the data. The common procedure for evaluating a query is to create a specific query evaluation plan per query. Normally a query can be rewritten to a large number of query evaluation plans, varying highly in evaluation time. A query optimiser tries to find the most efficient implementation in two phases: Static optimisation and dynamic optimisation [Kemper and Eickler, 2004, Garcia-Molina et al., 2008].

Static optimisation is performed by applying different equivalence rules on a logical representation of the query, for example swapping the order of arguments of a join operation or changing the order of other operators applied to the data.

The resulting logical query representation is then transformed to physical representation. In this *Dynamic Optimisation* phase the query optimiser tries to find an optimal query plan based on both data statistics and physical operator implementations.

Statistics about the data distribution can be used to aid the optimiser, mostly during dynamic optimisation.

5.2 XDEP Dependent Join Optimisation

Our implementation approach of rewriting an XSPARQL++ query to XQuery is naturally suited for performing logical query optimisation. The information we are using for optimisation is the query itself, knowledge about the communication with the SPARQL endpoint and the format of the returned data. Xdep is an approach for optimising queries with high execution times due to *SparqlForClause*s nested in a *ReturnClause* of a FLWOR' expression (e. g. Listing 3.5 on page 51).

To explain the details of the optimisation approach Xdep, some terms need to be defined first:

Definition 5.1 (XQuery Join, Join Variable)**.** In XQuery two nested `for` loops are performing a *join* if the inner loop reuses the variable of the outer loop to select corresponding data in a *WhereClause* or in the XPath expression. This variable is called *join variable*.

XQuery allows the straightforward implementation of *inner joins* by nesting *ForClause*s [Lehner and Schöning, 2004, Walmsley, 2007].

Definition 5.2 (Inner Join)**.** An *inner join* in XQuery is a join over two nested *ForClause*s.

Definition 5.3 (Dependent Join)**.** A *dependent join* in XSPARQL++ is an inner join over a *SparqlForClause*, referred to as *inner clause* or S^I, nested in either a *SparqlForClause* or a *ForClause*, referred to as *outer clause*, S^O, or F^O. *Dependent Variables* $Vars^{\text{dep}}$ are the variables the inner and the outer clause share: $Vars^{\text{dep}} = \text{vars}(S^O) \cap \text{vars}(S^I)$.

Xdep is a dependent join optimisation since it targets targets nested *SparqlForClause*s performing a dependent join. Xdep is a pre-rewriting optimisation (see phase 2 in Section 4.2.1).

We look at joins where the inner loop is a *SparqlForClause*. There are two relevant cases for the outer loop: it is either a plain XQuery *ForClause* or it is a *SparqlForClause*.

First in this section the standard rewriting of a nested *SparqlForClause* is shown using an example. After a short general description of Xdep its limitations of applicability are presented. Then the Xdep optimisation is explained in detail for a plain XQuery *ForClause* and then for a *SparqlForClause* as outer clause.

```
1  let $names := ("Alice", "Bob", "Stefan")
2  return
3  <persons>{
4  for $name in $names
5  return <friends of="{$name}"> {
6    for $fname from <relations.rdf>
7    where { [] foaf:name $name ;
8              foaf:knows [ foaf:name $fname ] }
9    return <person>{$fname}</person>}
10 </friends> }
11 </persons>
```

Listing 5.1: Nested *SparqlForClause*

5.2.1 XDEP Motivation

In order to describe XDEP we first demonstrate how nested *SparqlForClause*s are rewritten and evaluated by standard XSPARQL++ by examining an example query.

Example 5.1. The query example query in in Listing 5.1 returns an XML document containing lists of friends for three specific persons. The *ForClause* on line 4 iterates over a list of names `$names`. For each name in the list a query is sent to the SPARQL endpoint (lines 6–8), evaluated by the SPARQL engine and the result is returned. In the presented example this leads to three different SPARQL queries to be evaluated, once for each item in the `$names` sequence. For "Alice" the SPARQL query #1, shown in Listing 5.3, is sent to the SPARQL engine. SPARQL queries for the remaining two persons are handled in the same way (see the SPARQL queries #2 and #3 in Listings 5.4 and 5.5 respectively).

The different queries sent to the SPARQL endpoint are similar. The only difference between all these queries is the value which replaces the `$name` variable (see Listing 5.2 for the standard rewriting). Nevertheless the SPARQL engine has to parse, analyse, optimise and evaluate each query on its own. The results of the three SPARQL queries are given in Table 5.1.

After that the results have to be encoded in XML and then decoded by the XQuery engine again. Since the XQuery engine and the SPARQL endpoint communicate over HTTP, communication latency times have to be considered as well, especially if the two engines are not running on the same host.

The similarity of the queries of Listings 5.3, 5.4, and 5.5 indicate the optimisation potential we exploit with XDEP.

```
let $names := ("Alice", "Bob", "Stefan")
return
<persons> {
for $name at $_name_pos in $names
return
  <friends of="{$name}">{
    let $_aux_results3 := _xsparql:_sparql( fn:concat(
     "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
      SELECT $fname from <relations.rdf>
      where  { [] foaf:name ", _xsparql:_rdf_term( _xsparql:
          _binding_term( $name ) ), " ; foaf:knows [ foaf:name $fname ]
          .  } " ) )
    for $_aux_result3 at $_aux_result3_pos in _xsparql:_sparqlResults(
        $_aux_results3 )
    let $fname := _xsparql:_resultNode( $_aux_result3, "fname" )
    return
      <person>{fn:data( $fname )}</person>}
  </friends>
}
</persons>
```

Listing 5.2: Standard Rewriting of the Query in Listing 5.1

```
SELECT $fname FROM <relations.rdf>
WHERE { [] foaf:name "Alice" ;
           foaf:knows [ foaf:name $fname ] }
```

Listing 5.3: SPARQL query #1 of Listing 5.2

```
SELECT $fname FROM <relations.rdf>
WHERE { [] foaf:name "Bob" ;
           foaf:knows [ foaf:name $fname ] }
```

Listing 5.4: SPARQL query #2 of Listing 5.2

```
SELECT $fname FROM <relations.rdf>
WHERE { [] foaf:name "Stefan" ;
           foaf:knows [ foaf:name $fname ] }
```

Listing 5.5: SPARQL query #3 of Listing 5.2

$fname
"Bob"
"Charles"

(a) Results for "Alice" (Listing 5.3)

$fname
"Charles"

(b) Results for "Bob"(Listing 5.4)

$fname
No results at all

(c) Results for "Stefan"(Listing 5.5)

Table 5.1: Results of SPARQL queries of Listings 5.3, 5.4, and 5.5

5.2.2 General Optimisation Idea

Instead of calling SPARQL N times (while N is the number of elements/bindings of the outer loop), all the solution mappings which would be gathered by evaluating N SPARQL queries, will be gathered in a single SPARQL query at once. All the potential values of the join variable(s) are stored in a temporary variable. When iterating over the items of the outer loop, the current item is joined with the value of the temporary variable, thus selecting the corresponding data only. In summary there are two steps: (1) gather all the possibly relevant data before evaluating the outer query and (2) during evaluating the outer query select the corresponding data.

The query rewriter constructs a SPARQL query in an unconstrained form. *Unconstrained Form* in this context means, that the dependent variables are not replaced by a value, but the variable name itself remains in the SPARQL query 'as is'.

In every iteration of the outer loop the current item is joined with the data gathered in the first step. This join results in the same solution mappings as an unoptimised SPARQL query would.

The semantics of XDEP is defined by a new semantics rule $[\![\cdot]\!]_{ExprXDEP}$. This new rule is an additional normalisation rule and it is applied just before normalisation with $[\![\cdot]\!]_{Expr}$. In terms of the XQuery processing model (see Figure 2.2) $[\![\cdot]\!]_{ExprXDEP}$ could be applied after step SQ2.

Assumptions for a better runtime

For the optimisation to work efficiently we assume that the intermediary results fit into memory. If the results are too big to fit into the main memory, evaluation speed is probably reduced. In this case it is currently unknown how the resulting evaluation times would relate to the unoptimised query.

5.2.3 Conditions for Applicability

Definition 5.4 ([Angles and Gutierrez, 2008])**.** A graph pattern of the form P FILTER C is said to be *safe* if $\text{vars}(C) \subseteq \text{vars}(P)$.

Definition 5.5 (Parametrised Query)**.** A *parametrised query* is a nested *Sparql-ForClause P* FILTER *C* reusing a variable of some outer scope in the built-in constraint *C* only, and not in a *P*: $\text{vars}(C) \nsubseteq \text{vars}(P)$.

Since Xdep is not applicable in the general case, two conditions have to be satisfied:

Static Dataset The dataset clause of the inner query is static, meaning no variables from the outer for query are used for determining the dataset dynamically.
If the *DatasetClause* of the inner query contains a variable $X, then this variable must be bound outside of the outer query.

Dependent Join The inner query is not a parametrised query, i. e., the inner query must be a safe query. In other words, even if a dependent variable may be not bound by the outer query, it is not allowed for a dependent variable to occur in a *Filter* constraint only. Thus if $S^I = P$ FILTER *C*, then $\text{vars}(C) \subseteq \text{vars}(P)$.

Xdep is described below for two different cases: In the first case the outer loop is a plain XQuery *ForClause*, in the second case the outer loop is a *SparqlForClause*.

5.2.4 Outer XQuery `for` clause

In this case there is exactly one dependent variable, i. e., the variable of the outer *ForClause*. First two auxiliary functions are introduced:

fs:dep The function *fs:dep* (`variables`, *GroupGraphPattern*) returns the variables of the first argument `variables` occurring in the *GroupGraphPattern*. The *fs:nondep* function *fs:nondep* (`variables`, *GroupGraphPattern*) instead, returns all the variables which occur in the *GroupGraphPattern* but not in the `variables` sequence. For the Xdep mapping to XSPARQL++ we introduce the $[\![\cdot]\!]_{ExprXDEP}$ rule, as already outlined above.

The normalisation rule $[\![\cdot]\!]_{SparqlQuery(Dataset)}$ evaluates the SPARQL query, similar to the $[\![\cdot]\!]_{SparqlQuery}$ normalisation of the original XSPARQL semantics, but it implements the changed SPARQL graph pattern matching as introduced in 3 for XSPARQL++.

Rule 5.1. A dependent join with an outer XQuery `for` clause is rewritten as follows:

$$
\begin{array}{l}
[\![\ \texttt{for}\ \mathit{VarName}_f\ \mathit{OptTypeDeclaration}\ \mathit{OptPositionalVar}\ \texttt{in}\ \mathit{Expr}_f \\
\texttt{return} \\
\quad \texttt{for}\ \mathit{Vars}\ \texttt{from}\ \mathit{Dataset} \\
\quad \texttt{where}\ \mathit{GraphPattern}\ \mathit{SolutionModifier} \\
\quad \mathit{ReturnClause}
\ ]\!]_{\mathit{ExprXDEP}} \\
\qquad == \\
[\![\ \texttt{let}\ \$\_\mathit{results}\ := \ [\![\ \{\mathit{VarName}_f\} \cup \mathit{Vars} \\
\qquad\qquad \texttt{where}\ \mathit{GraphPattern} \\
\qquad\qquad \mathit{SolutionModifier}
\ ]\!]_{\mathit{SparqlQuery}(\mathit{Dataset})} \\
\texttt{for}\ \mathit{VarName}_f\ \mathit{OptTypeDeclaration}\ \mathit{OptPositionalVar}\ \texttt{in}\ \mathit{Expr}_f \\
\texttt{return} \\
\quad \texttt{for}\ \$\_\mathit{result}\ \texttt{at}\ \$\_\mathit{result\_Pos}\ \texttt{in}\ \$\_\mathit{results}//\_\mathit{sparql\_result}{:}\mathit{result} \\
\quad \texttt{where}\ [\![\ \mathit{fs{:}dep}((\mathit{VarName}_f), \mathit{GraphPattern})\ ]\!]_{\mathit{SparqlResDep}(\$\_\mathit{result})} \\
\quad \texttt{return} \\
\qquad [\![\ \mathit{fs{:}nondep}((\mathit{VarName}_f), \mathit{GraphPattern})\ ]\!]_{\mathit{SparqlResult}(\$\_\mathit{result})} \\
\qquad \mathit{ReturnClause}
\ ]\!]_{\mathit{Expr}}
\end{array}
$$

Instead of iterating over all the items in the sequence of the *ForClause*, first the *SparqlForClause* is evaluated (in our implementation by a separate SPARQL engine). Normally the dependent variable occurring in the *SparqlForClause* would have to be replaced by its value. XDEP instead evaluates the *SparqlForClause* unconstrained to perform the join later in XQuery alone. Thus the dependent variable has to be added to the variable list which is the first parameter of the $[\![\cdot]\!]_{\mathit{SparqlQuery}}$ rule. The result of the *SparqlForClause* is assigned to a temporary and new variable, i. e., `$_result` in the rewriting rule.

Next the *ForClause* iterates over the items of its sequence. In every iteration the current item is joined with the set of solution mappings assigned to `$_result` by using the auxiliary mapping rule *SparqlResDep*.

$$
\begin{array}{c}
[\![\$\mathit{VarName}_1 \cdots \$\mathit{VarName}_n]\!]_{\mathit{SparqlResDep}(\$\mathit{result})} \\
== \\
\mathit{fs{:}join}(\$\mathit{VarName}_1, \mathit{fn{:}data}(\$\mathit{result}/\mathit{sr{:}binding}[@\mathit{name} = "\mathit{VarName}_1"]/*)\ \texttt{and} \\
\cdots\ \texttt{and} \\
\mathit{fs{:}join}(\$\mathit{VarName}_n, \mathit{fn{:}data}(\$\mathit{result}/\mathit{sr{:}binding}[@\mathit{name} = "\mathit{VarName}_n"]/*)
\end{array}
$$

Next we define the function *fs:join* which joins the values of a dependent variable of the outer loop and the inner loop. Modelling the XSPARQL++ dependent join we understand "joins" here in the style of SPARQL, i.e. based on compatible mappings, rather than on strictly matching values. Two values $\mathit{Value}^{\text{outer}}$ and $\mathit{Value}^{\text{inner}}$ can be joined if they have the same value, if the value of the variable of the outer query is a blank node, or if the outer variable is unbound. This condition models the behaviour of SPARQL basic graph

fs:join

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/
       xsparql.xquery" at "http://xsparql.deri.org/demo/xquery/xsparql-
       types.xquery";
2  import schema namespace _sparql_res = "http://www.w3.org/2007/SPARQL/
       results#" at "http://www.w3.org/2007/SPARQL/result.xsd";
3  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
4  let $names := ("Alice", "Bob", "Stefan")
5  return
6  <persons> {
7  let $_aux_results3 := _xsparql:_sparql( fn:concat(
8       "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
9        SELECT $fname $name from <relations.rdf>
10       where  { [] foaf:name $name ; foaf:knows [ foaf:name $fname ] .  }
             " ) )
11 for $name at $_name_pos in $names
12 return
13   <friends of="{$name}">{
14     for $_aux_result3 at $_aux_result3_pos in _xsparql:_sparqlResults(
           $_aux_results3 )
15     where ($name = fn:data($_aux_result/_sparql_res{:}binding[@name = "
           name"]/*) or $name instance of schema-element(_sparql_res:bnode)
           )
16     return
17       let $fname := _xsparql:_resultNode( $_aux_result3, "fname" )
18       return
19         <person>{fn:data( $fname )}</person>}
20   </friends>
21 }
22 </persons>
```

Listing 5.6: Xdep optimised rewriting of Listing 5.1

pattern matching as if $Value^{\text{inner}}$ would have been replaced by the value of $Value^{\text{outer}}$:

$$\begin{gathered} \mathit{fs{:}join}(Value^{\text{outer}}, Value^{\text{inner}}) \\ == \\ (Value^{\text{outer}} = Value^{\text{inner}} \texttt{ or } Value^{\text{outer}} \texttt{ instance of schema-element}(\mathit{sr{:}bnode}) \\ \texttt{or } \mathit{unbound}(Value^{\text{outer}})) \end{gathered}$$

Example 5.2. In Listing 5.6 the query from Listing 5.1 is rewritten using Xdep. After the XQuery prolog, the inner SPARQL query is executed, but without replacing the dependent variable `$name` with a value. Instead, possible values of `$name` will be part of the solution mappings stored in the `$_aux_results3` variable.

$name	$fname
“Alice”	“Bob”
“Alice”	“Charles”
“Bob”	“Charles”

Table 5.2: Result of XDEP optimised SPARQL query of lines 7–10 of Listing 5.6

```
1  <?xml version="1.0"?>
2  <sparql xmlns="http://www.w3.org/2005/sparql-results#">
3    <head>
4      <variable name="name"/>
5      <variable name="fname"/>
6    </head>
7    <results>
8      <result>
9        <binding name="name">
10         <literal>Alice</literal>
11       </binding>
12       <binding name="fname">
13         <literal>Bob</literal>
14       </binding>
15     </result>
16     <result>
17       <binding name="name">
18         <literal>Alice</literal>
19       </binding>
20       <binding name="fname">
21         <literal>Charles</literal>
22       </binding>
23     </result>
24     <result>
25       <binding name="name">
26         <literal>Bob</literal>
27       </binding>
28       <binding name="fname">
29         <literal>Charles</literal>
30       </binding>
31     </result>
32   </results>
33 </sparql>
```

Listing 5.7: Result of XDEP optimised SPARQL query of lines 7–10 of Listing 5.6

Table 5.2 and Listing 5.7 show the result of the corresponding SPARQL query. Next XQuery iterates over the values of the outer loop, `$names` in this case.

For each `$name` exactly those solution mappings are selected for which the value of XQuery `$name` variable is the same as the value of the `$name` variable in the solution mapping.

Eventually the value of the non-dependent variables, in this case only `$fname`, are retrieved from the SPARQL results and the *ReturnClause* can be evaluated as before.

So instead of calling SPARQL once for each name, the SPARQL query is evaluated only once and the join is performed in the XQuery *WhereClause* in line 15 of Listing 5.6.

5.2.5 Outer *SparqlForClause*

Although the rule for optimising a *SparqlForClause* nested in another *SparqlForClause* below looks more complicated, the procedure is in fact the same: The inner SPARQL query, in an unconstrained form, is evaluated before the outer SPARQL query. But since a *SparqlForClause* can assign values to more than one variable, it is possible to join over multiple dependent variables instead of only one as before.

The following rule shows this by adding all the dependent variables to the variable list of the inner query. Again the dependent variables are not replaced by a value in the *GroupGraphPattern*, since they are needed in the SPARQL result, assigned to `$_results_inner`.

The outer *SparqlForClause* is rewritten as in standard XSPARQL.

Rule 5.2. For every solution mapping of the outer *SparqlForClause* a new *ForClause* iterates over the results of the inner query and joins the relevant parts in a *WhereClause* using the dependent variables. In the following *ReturnClause* all the non-dependent variables of the inner query are assigned to actual XQuery variables. Eventually the final *ReturnClause* is evaluated.

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{for } \mathit{Vars}^{\text{outer}} \texttt{ from } \mathit{Dataset}^{\text{outer}} \\
\texttt{where } \mathit{GraphPattern}^{\text{outer}}\ \mathit{SolutionModifier}^{\text{outer}} \\
\texttt{return} \\
\quad \texttt{for } \mathit{Vars}^{\text{inner}} \texttt{ from } \mathit{Dataset}^{\text{inner}} \\
\quad \texttt{where } \mathit{GraphPattern}^{\text{inner}}\ \mathit{SolutionModifier}^{\text{inner}} \\
\quad \mathit{ReturnClause}
\end{array}
\right]\!\!\right]_{\mathit{ExprXDEP}}
$$

$$==$$

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{let \$}\mathit{\_results\_inner} := \\
\left[\!\!\left[
\begin{array}{l}
\mathit{Vars}^{\text{inner}} \cup \mathit{fs{:}dep}(\mathit{Vars}^{\text{outer}}, \mathit{GraphPattern}^{\text{inner}}) \\
\mathit{GraphPattern}^{\text{inner}} \\
\mathit{SolutionModifier}^{\text{inner}}
\end{array}
\right]\!\!\right]_{\mathit{SparqlQuery}(\mathit{Dataset}^{\text{inner}})} \\
\texttt{let \$}\mathit{\_results\_outer} := \\
\left[\!\!\left[
\begin{array}{l}
\mathit{Vars}^{\text{outer}} \\
\mathit{GraphPattern}^{\text{outer}} \\
\mathit{SolutionModifier}^{\text{outer}}
\end{array}
\right]\!\!\right]_{\mathit{SparqlQuery}(\mathit{Dataset}^{\text{outer}})} \\
\texttt{for \$}\mathit{\_result\_outer} \texttt{ at \$}\mathit{\_result\_pos\_outer} \\
\quad \texttt{in \$}\mathit{\_results\_outer//\_sparql\_result{:}result} \\
\left[\!\!\left[\mathit{Vars}^{\text{outer}}\right]\!\!\right]_{\mathit{SparqlResult}(\texttt{\$}\mathit{\_result\_outer})} \\
\texttt{return} \\
\quad \texttt{for \$}\mathit{\_result\_inner} \texttt{ at \$}\mathit{\_result\_pos\_inner} \\
\quad\quad \texttt{in \$}\mathit{\_results\_inner//\_sparql\_result{:}result} \\
\quad \texttt{where } \left[\!\!\left[\mathit{fs{:}dep}(\mathit{Vars}^{\text{outer}}, \mathit{GraphPattern}^{\text{inner}}) \right]\!\!\right]_{\mathit{SparqlResDep}(\texttt{\$}\mathit{\_result\_inner})} \\
\quad \texttt{return} \\
\quad\quad \left[\!\!\left[\mathit{fs{:}nondep}(\mathit{Vars}^{\text{outer}}, \mathit{GraphPattern}^{\text{inner}}) \right]\!\!\right]_{\mathit{SparqlResult}(\texttt{\$}\mathit{\_result\_inner})} \\
\quad\quad \mathit{ReturnClause}
\end{array}
\right]\!\!\right]_{\mathit{Expr}}
$$

Example 5.3. In the example query in Listing 5.8 we create list of friends which two people have in common. For this example two FOAF graphs are used.

In the normal rewriting first a SPARQL call for getting all the friends of the first person is performed. Then for each of these friends a SPARQL call on the second dataset is made.

In the optimised version instead (see Listing 5.9), first the SPARQL call of the inner SPARQL query is executed (the second FOAF graph), getting all possibly needed data at once:

All the solution mappings which may be relevant for the inner loop later are stored in the `$_aux_results19` variable. After that, the outer *SparqlFor-Clause* is evaluated as before, by getting all the friends of the first dataset. Next XQuery iterates over the solution mappings of the outer query (standard XSPARQL++ rewriting).

Finally a for loop iterates over the data of the inner query, this time not performing a SPARQL query, but joining the already gathered data in an

```
1  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
2
3  <sharedfriends>
4  {
5    for $Name from <http://stefanbischof.at/foaf.rdf>
6    where { [] foaf:knows $Person .
7            $Person a foaf:Person ;
8                    foaf:name $Name . }
9    return
10       for $FriendName from <http://nunolopes.org/foaf.rdf>
11       where { [] foaf:knows $P .
12               $P a foaf:Person ;
13                  foaf:name $Name . }
14       return <sharedfriend name="{$Name}" />
15 }
16 </sharedfriends>
```

Listing 5.8: Shared Friends Query

XQuery *WhereClause*. Eventually the *ReturnClause* is evaluated, returning an XML element containing the shared friend's name in an attribute.

Xdep rewrites the query to need two SPARQL queries only, independent of the number of solution mappings, i. e., persons, of the outer loop.

Appendix D.2 presents a formal description and proof sketches stating that Xdep implements the XSPARQL++ semantics.

5.3 Practical Optimisations

Besides the dependent join optimisation Xdep we introduce three other simple optimisations by example only. Formal definition of these rules will be part of future work.

5.3.1 Optimise Projection for FOR * queries

When performing a SPARQL `for` $*$ query, the SPARQL implementation returns all bound variables. This can lead to transferring unused and therefore useless data. But this useless data has to be retrieved and is, in the worst case, only a waste of resources.

By automatically evaluating which variables are really used later in the *ReturnClause*, and deleting the unused variables from the *SparqlForClause*, an XSPARQL++ implementation could optimise data transfer.

```
1  import module namespace _xsparql = "http://xsparql.deri.org/XSPARQLer/
       xsparql.xquery" at "http://xsparql.deri.org/XSPARQLer/xsparql-types.
       xquery";
2  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
3  import schema namespace _sparql_res = "http://www.w3.org/2007/SPARQL/
       results#" at "http://www.w3.org/2007/SPARQL/result.xsd";
4  <sharedfriends> {
5    let $_aux_results19 := _xsparql:_sparql( (
6     "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
7      SELECT $P $Name from <http://nunolopes.org/foaf.rdf>
8      where { [] foaf:knows $P . $P a  foaf:Person ; foaf:name $Name .  }
           ") )
9    let $_aux_results15 := _xsparql:_sparql( (
10    "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
11     SELECT $Person $Name from<http://stefanbischof.at/foaf.rdf>
12     where { [] foaf:knows $Person . $Person  a  foaf:Person ; foaf:name
           $Name .  } ") )
13   for $_aux_result15 at $_aux_result15_pos in _xsparql:_sparqlResults(
         $_aux_results15 )
14     let $Name := _xsparql:_resultNode( $_aux_result15, "Name" )
15     return
16       for $_aux_result19 at $_aux_result19_pos in _xsparql:
             _sparqlResults( $_aux_results19 )
17       where ($Name = _xsparql:_resultNode( $_aux_result19, "Name" ) or
             $Name instance of schema-element(_sparql_res:bnode))
18       return
19         <sharedfriend name="{$Name}" />
20   }
21 </sharedfriends>
```

Listing 5.9: Optimised rewriting of query in Listing 5.8

In the following example (see Listing 5.10) the bindings of all three variables `$s`, `$p` and `$x` are passed to XQuery although only one, `$x`, is actually needed in the construct expression.

```
1 for *
2 from <http://xsparql.deri.org/data/alice.ttl>
3 where {$s $p $x}
4 construct {$x :p [$x _:a] }
```

Listing 5.10: FOR * Query Example

By deleting `$s` and `$p` from the variable list (see Listing 5.11), the transferred data is reduced to a minimum, instead of enumerating all variables occurring in the *WhereClause* as the XSPARQL++ and SPARQL specification

demands.

```
1 for $x
2 from <http://xsparql.deri.org/data/alice.ttl>
3 where { $s $p $x }
4 construct {$x :p [$x _:a] }
```

Listing 5.11: Optimised Projection

5.3.2 Simpler Serializer Function

The previous implementation of the *_xsparql:serialize* depended on the recursive usage of *fn:concat* to serialise its arguments according to their types. The original *fs:serialize* is as described in [Krennwallner et al., 2009, Section 2.1]. It turned out in our experiments that calling that function recursively once for each argument is slow for an increasing number of arguments. By using the *fn:string-join* function instead, cf. `http://xsparql.deri.org/demo/xquery/xsparql-types.xquery` for the definition, the number of function calls can be reduced, thus reducing evaluation time queries containing *ConstructClause*s.

The *_xsparql:serialize* function takes a sequence of items as argument and merges them by concatenation. If there are multiple static strings occurring one after another in this argument sequence, they can be already concatenated statically while rewriting the query. This can reduce evaluation time since the concatenation is already performed during query rewriting.

5.3.3 Static Expression Evaluation

Expressions that are independent of the dynamic environment, e. g., no usage of variables or embedded expressions, are called *static expressions*. Since no dynamic environment is need to evaluate these, they can be evaluated statically, i. e., before query evaluation. An XQuery engine can benefit by statically evaluating XQuery built-in functions. Since we have more information about the expressions introduced by XSPARQL++ than an XQuery engine during static optimisation, we can evaluate some of these expressions, or parts of them, statically. Since this measure moves some workload from the XQuery engine to the XSPARQL++ rewriter and because such expressions have to be evaluated at most once, the query evaluation time is reduced.

As XSPARQL++ defines, an expression used in a *ConstructClause* has to be checked for validity before the RDF triple can be built. Nevertheless there exist some cases where validity of an expression can be inferred statically.

```
select * from <graph.rdf>
where {:some $p :other }
construct {
  $p $p $p .
}
```

Listing 5.12: Example scenario query

```
return _xsparql:serialize((
  if(validSubject($p)) then
    if(validPredicate($p)) then
      if(validObject($p)) then
        $p, " ", $p, " ", $p, " . "
      else
        ""
    else
      ""
  else
    ""
))
```

Listing 5.13: Standard rewriting of example scenario

For demonstration with give the explanation of one scenario if the expression is only a variable `$p` bound at the predicate position of a *WhereClause* in a *SparqlForClause* in Listing 5.12.

The rewritten query (see Listing 5.13) checks the validity of the `$p` variable for every position.

Since `$p` has to be an IRI (at predicate position only IRI s are allowed) and IRIs are allowed on any position, none of the `else` branches will ever be reached. Since the triple will always be valid, all the validity checking function calls can safely be removed (see Listing 5.14).

It is also a fact, that any variable bound by a *SparqlForClause* would be a valid RDF object. Therefore validity checks for variables bound by *SparqlForClause*s at object position in a *ConstructClause* are superfluous and can safely be removed too.

Remark 5.1. In SPARQL alone the same would apply to RDF subjects, but literals are still not allowed on subject position in RDF.

Table 5.3 shows the cases for which validity checking is actually needed, given that the expression used in the constructed RDF term is a variable only. The columns show the position at which the variable was bound in the *SparqlForClause*. The rows show the position the variable is used in the *ConstructClause*. Therefore validity checking for variable only expressions

```
return _xsparql:serialize((
        $p, " ", $p, " ", $p, " . "
))
```

Listing 5.14: Optimised rewriting of example scenario

	binding position		
construct position	subject	predicate	object
subject	no	no	no
predicate	yes	no	yes
object	no	no	no

Table 5.3: Need of Validity Checking

in constructed RDF terms, is only needed for variables used in predicate position when bound at either subject or object position.

This optimisation approach could be extended to other expressions than variables only, when (partly) static evaluation is possible.

Chapter 6

Empirical Evaluation & Discussion

In order to validate our expectations of more efficient query evaluation of the optimisation of the last chapter (to reduce query evaluation times applying dependent join optimisation) we provide a practical evaluation in this chapter.

The presented methodology as well as the results and discussion shall show the effects of the devised optimisation approach XDEP in practise.

6.1 Measuring XSPARQL Performance

To measure performance of computer hardware and programs, computer scientists use benchmarks in various forms. The more complex hardware or software gets, the harder it is to provide a valid test procedure, which truly measures the performance of a system.

Since query language implementations provide a lot of features there generally exist several possibilities to write the same query, it is hard to make reliable statement of implementation performance. A benchmark suite solves this problem by providing an example scenario containing concrete datasets and queries to compare different query engines.

We chose the XMark XQuery benchmark suite [Schmidt et al., 2002] because it is one of the most widely used XQuery benchmarks for comparing XQuery engines [Afanasiev and Marx, 2008]. It uses synthetic data of an auction site, with users selling items and bidding on items. One advantage is the data generator, which can create coherent test data of arbitrary size. This allows to compare query evaluation times for different source data sizes.

We conducted the following three experiments:

1. Run the XQuery benchmark queries on datasets of different size;

2. For the second experiment we translated the benchmark queries and datasets manually to XSPARQL and evaluated them on datasets of different size. The datasets were translated to RDF before;

3. We applied the dependent join optimisation described in Chapter 5 to the relevant queries of experiment #2 and executed them again on the different datasets.

By comparing the results of experiment 1 and 2, we relate plain XQuery with XSPARQL query evaluation. Since XSPARQL uses two execution engines, we expect slower evaluation times for XSPARQL for most queries. Nevertheless evaluation times are influenced by the SPARQL engine implementation. Since there exists no algorithm to translate data and queries from XQuery/XML to XSPARQL/RDF, the evaluation times are also influenced by details of the manual query translation and the specific format of the source data translation.

We expect faster evaluation times for experiment 3 than the baseline times of experiment 2. Besides the evaluation time of the inner SPARQL query the ratio of evaluation times (baseline vs. optimised) should increase on the number of iterations of the inner query. For the generated datasets the number of iterations depends directly on the size of the datasets. Therefore we expect the evaluation times of experiment 3 growing slower than the evaluation times of experiment 2 with increasing dataset size.

Benchmark queries

Since we are performing three experiments, three query sets are needed. The general starting point are the XMark queries.

For the first experiment we use the XMark XQuery queries. They are available on the XMark homepage[1] and in Appendix B. We replaced the the function call *fn:doc*, used for referring to the XML source document by XMark, with an external variable containing the path to the XML source document. This measure provides a consistent way of passing the data source path to both XQuery and XSPARQL queries.

The queries for the second experiment were created by translating the XMark XQuery queries to XSPARQL. Although the translation was done manually, we tried to minimise a possible performance impact. Appendix B lists the translated queries.

To show the effects of our optimisation approach in Chapter 5 a third set of queries was created: After translating the XSPARQL queries of experiment #2 to XQuery we applied the XDEP optimisation on them.

The only queries qualifying for XDEP are the queries 8, 9 and 10. They contain nested *SparqlForClause*s and fulfil the constraints specified in Sec-

[1] `http://www.xml-benchmark.org/`

dataset #	xmlgen	XML (bytes)	RDF (bytes)
1	0.05	5 750 712	5 106 023
2	0.10	11 669 705	10 337 185
3	0.20	23 514 450	20 687 081
4	0.50	58 005 732	51 525 338
5	1.00	116 517 075	104 453 072

Table 6.1: Benchmark data source sizes

dataset #	XML (elements)	RDF (triples)
1	76 963	69 952
2	154 015	139 891
3	307 851	277 403
4	763 909	690 482
5	1 527 585	1 381 668

Table 6.2: Benchmark data source sizes

tion 5.2.3. Queries 11 and 12 contain nested *SparqlForClause*s as well but they are parametrised queries and XDEP is therefore not applicable.

Benchmark data

To show the performance of the query evaluation, data sources in ascending sizes were created by using the XMark data generator *xmlgen*[2]. Data generation with xmlgen depends on a single parameter determining the size of the result file. Table 6.1 shows how this single factor relates to the final size of the generated XML source document. Table 6.2 shows how the different datasets are encoded in XML and RDF.

The resulting XML file contains data of an artificial auction site. People can open auctions on items in different categories on different continents; they can bid on these items and finally sell the items. Table 6.3 shows several characteristics of the different datasets.

For the experiments #2 and #3, data has to be in RDF format. To make the two scenarios, XML data and RDF data comparable, the XML documents generated by xmlgen, are translated to RDF. Since this translation is a usecase of XSPARQL, it was used for this task too. Table 6.1 shows the file size of the translated RDF file in Turtle notation compared to the original XML file. It also lists the resulting number of triples for each dataset; a number which is more relevant in the RDF world than the pure file size since it shows data complexity more accurately.

Table 6.3 shows the several characteristics of the benchmark data for

[2] `http://www.xml-benchmark.org/generator.html`

dataset #	persons	item cat.	open auc.	closed auc.	avg. open bids
1	1 275	50	600	487	5.1
2	2 550	100	1 200	975	5.2
3	5 100	200	2 400	1 950	5.0
4	12 750	500	6 000	4 875	4.9
5	25 500	1 000	12 000	9 750	5.0

Table 6.3: Benchmark datasets characteristics

the different datasets. The most important for our later analysis will be the number of persons and item categories. These characteristics apply not only to the RDF data, but also to the original XML document. The table shows a direct correlation between the xmlgen factor, the file sizes and specific data characteristics namely the number of persons, the number of item categories, the number of open auctions, the number of closed auctions, and the average number of open bids on an open auction.

In summary we perform an evaluation in three experiments #1, #2, #3, using all or some of the 20 XMark benchmark queries (and their XSPARQL counterparts) on five different datasets, #1–#5 (in XML and RDF format).

6.2 Experimental Results

The test system was an AMD Opteron 250 dual core system with 4 GB main memory running Ubuntu 8.10. For evaluating the XQuery queries we used Saxon, a widely known XQuery and XSLT evaluation engine, in the version 9.2 enterprise edition (EE). We needed to use the EE edition because our new implementation relies heavily on XQuery data types and especially on the types defined in the schema for the SPARQL query results XML format [Beckett and Broekstra, 2008], and none of the other editions provides schema-aware evaluation of XQuery queries. We need a schema-aware XQuery engine, since that feature allows us to validate XML documents and use the XML types defined in the schema of the SPARQL query results XML format [Beckett and Broekstra, 2008]. Furthermore, according to [Afanasiev and Marx, 2008] Saxon is one of the best performing XQuery engines.

For the SPARQL query evaluation we used Joseki 3.4.0 which provides a SPARQL endpoint for the ARQ SPARQL library. To run both engines we use Sun Java 1.6.0 in a 64bit installation. Shell scripts automated the benchmarking process, GNU Octave and Gnuplot are the tools used for data processing and visualisation.

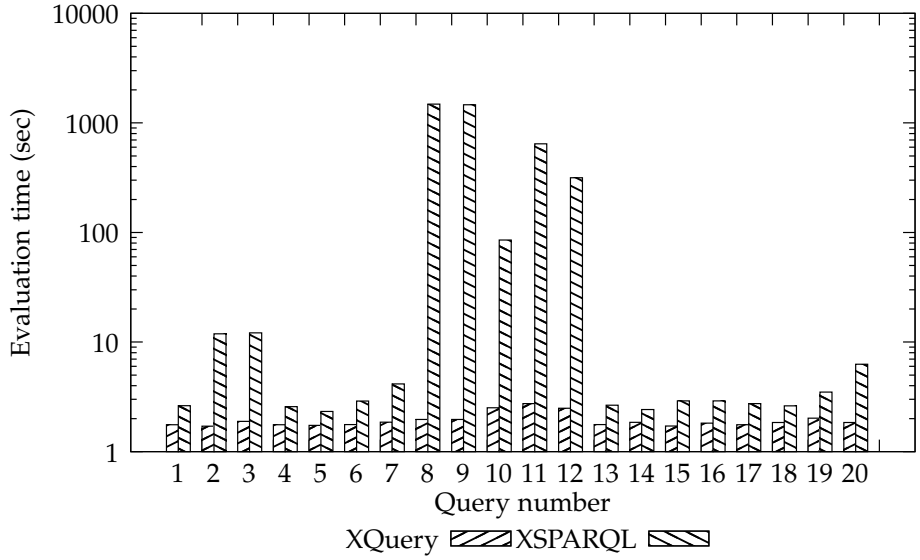

Figure 6.1: Mean evaluation times of ex. #1 and #2 on dataset #1 in sec

We evaluated every query 20 times and computed an arithmetic mean value as final result. If the evaluation of a query took over 10 hours, our test framework aborted the evaluation process.

This section presents the evaluation results and the comparison of these. Appendix C contains all query evaluation times for reference.

6.2.1 Experiment #1 vs. #2

In Experiment #1 we measured the evaluation times of all 20 queries of the XMark XQuery benchmark suite on XML documents of different sizes as specified above. Figure 6.1 and Table 6.4 show the query evaluation times of experiment #1 as well as those of experiment #2 using dataset #1. Appendix lists the query evaluation times for all dataset sizes.

As expected all the XSPARQL queries where slower than the original XQuery queries. For the following queries XSPARQL was much slower than XQuery:

Query 2 and 3 We assume that these queries are taking so much longer to evaluate than most of the other queries because of the big overhead that results of the big number of results the queries produce.

Query 8–12 By examining these queries we find all of them containing nested *SparqlForClause*s. Therefore these were also the candidates for XDEP optimisation evaluated in experiment #3. These results acknowledge the

Query #	XQuery	XSPARQL	Query #	XQuery	XSPARQL
1	1.8	2.6	11	2.7	641.5
2	1.7	11.9	12	2.5	314.5
3	1.9	12.0	13	1.8	2.7
4	1.8	2.6	14	1.8	2.4
5	1.7	2.3	15	1.7	2.9
6	1.8	2.9	16	1.8	2.7
7	1.8	4.1	17	1.8	2.9
8	2.0	1 478.7	18	1.8	2.6
9	2.0	1 459.3	19	2.0	3.5
10	2.5	85.0	20	1.8	6.3

Table 6.4: Mean evaluation times of ex. #1 and #2 on dataset #1 in sec

dataset #	Query 8	Query 9	Query 10
1	20.2	12.3	36.2
2	55.5	26.7	107.7
3	189.1	74.2	400.4
4	1 114.4	383.3	2 330.9
5	4 501.8	1 433.8	9 414.9

Table 6.5: Optimised query evaluation times in seconds

motivation of queries with nested *SparqlForClause*s probably having the most promising optimisation potential.

Query 20 This query takes longer to evaluate because it contains not one but three different *SparqlForClause*s.

In conclusion it is clear, that XSPARQL queries with more than one *SparqlForClause* show the biggest optimisation potential.

6.2.2 Experiment #2 vs. #3

This section compares the results of experiment #2, standard XSPARQL, with the corresponding results of experiment #3, Xdep optimised XSPARQL. Since only the queries 8, 9, and 10 are relevant for Xdep optimisation, the results of these queries only are discussed below.

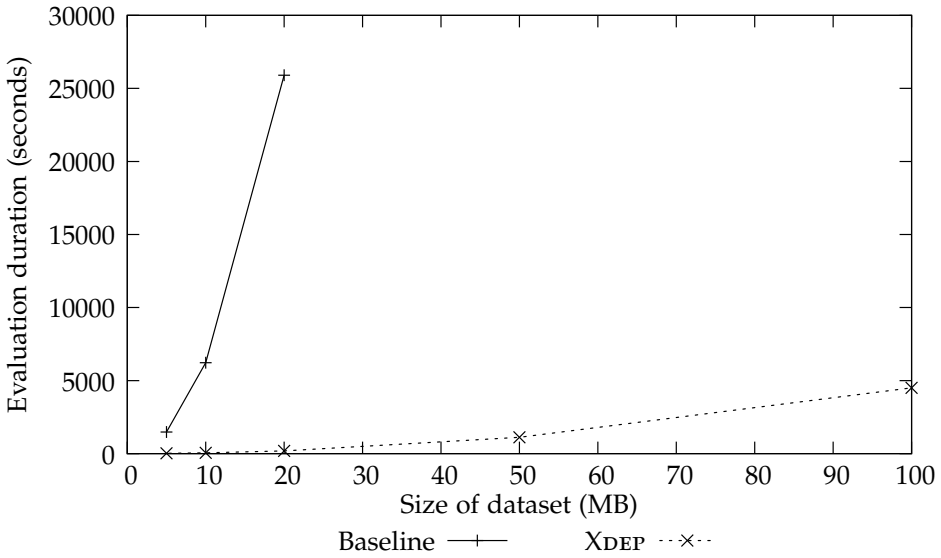

Figure 6.2: Evaluation times for query 8 in seconds

6.2.2.1 Query 8

Query 8 is the first query relevant to XDEP optimisation. It contains two nested *SparqlForClause*s performing a dependent join (see Appendix B for all unoptimised and optimised queries).

Figure 6.2 shows the mean query evaluation times for query 8 using standard XSPARQL rewriting compared with XDEP rewriting with increasing dataset size. Since query evaluation times of the standard XSPARQL rewriting for datasets bigger than 20 MB exceeded the timeout, no results are available for those datasets. The times for XDEP are also listed in Table 6.5.

The XDEP optimised evaluation times are lower than the XSPARQL baseline times. The graph also shows that the XDEP rewritten query can process 10 times as much data as the unoptimised XSPARQL query (XDEP optimised query needs less time to evaluate the 100 MB dataset then the unoptimised query needs for the 10 MB dataset). A performance increase factor is given later in this Chapter.

Listing 6.1 shows query 8 without the preamble. The outer *SparqlForClause* iterates over all persons in the dataset, the inner *SparqlForClause* first retrieves all items one person bought, and counts them afterwards. Therefore the inner *SparqlForClause* is evaluated once for every person in the dataset. Since the number of persons in a dataset correlates directly with the size of the dataset (see Table 6.3), the number of outer iterations depends directly on the size of the dataset as well. This applies not only to query 8 but to every XMark query containing nested *SparqlForClause*s.

```
for $id $name
from $graph
where {
$person a foaf:Person ;
        :id $id ;
        foaf:name $name .
}
return <item person="{$name}">
  {
    let $x := for * from $graph
      where {
        $ca a :ClosedAuction ;
            :buyer [ :id $id ] .
      }
      return $ca
    return count($x)
  }
</item>
```

Listing 6.1: Query 8

Without optimisation one SPARQL call is done for each person separately. The time needed for one of these SPARQL calls depends on the used engine, but it also on the size of the dataset: The bigger the dataset the more persons are contained (see Table 6.3 on page 112) and the more SPARQL calls have to be performed. Secondly, the bigger the dataset the longer each SPARQL call needs. Therefore we get the discovered increase, assumed quadratic, in evaluation time for increasing dataset size.

In the optimised version, only two SPARQL calls need to be performed, independent of the number of persons contained in the dataset. One single SPARQL call would select all the needed information of the bought items, and another SPARQL call would select all the persons information.

Although the increase of query evaluation time with increasing dataset size retained quadratic, the practical evaluation showed, that the gain of the optimisation is bigger the more SPARQL calls can be saved, which confirms our assumption that the number of interleaved SPARQL calls is indeed a major bottleneck.

6.2.2.2 Query 9

Query 9 is syntactically similar to query 8. Table 6.4 shows that the query evaluation times for both queries using the standard XSPARQL rewriter are similar to. The only difference between these two queries is the inner *SparqlForClause*, especially the *WhereClause* (see Appendix Section B.2). Figure 6.3

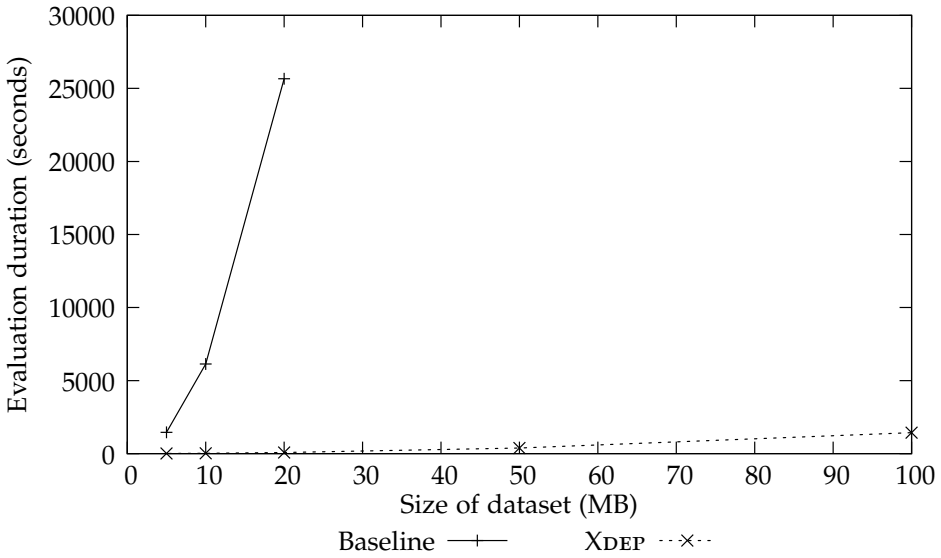

Figure 6.3: Evaluation times for query 9 in seconds

shows the results of the XDEP optimised version of query 9 in relation to the standard XSPARQL evaluation times.

The savings for query 9 are even bigger than for query 8.

6.2.2.3 Query 10

A similar behaviour can be observed for query 10. But for this query the difference between the baseline and the optimised version is smaller. Queries 8 and 9 iterate over all the persons, while query 10 iterates over all the item categories. As shown in Table 6.3, the number of categories, and therefore the number of iterations of the outer loop, is by a factor 25 smaller than for the queries 8 and 9. Additionally the SPARQL *WhereClause* has a different structure thus also influencing the evaluation times.

6.2.2.4 Comparison

Figures 6.5 and 6.6 show the results of experiments #2 and #3 in comparison. The XQuery queries 8 and 9 have nearly the same evaluation times and are therefore nearly indistinguishable.

This section compares the resulting evaluation times of experiment #2 and #3 first by dataset size and then by the number of iterations of the inner loop

Figure 6.5 shows the relations between the standard XSPARQL rewriting and XDEP using linear axes based on the dataset size again. The figure

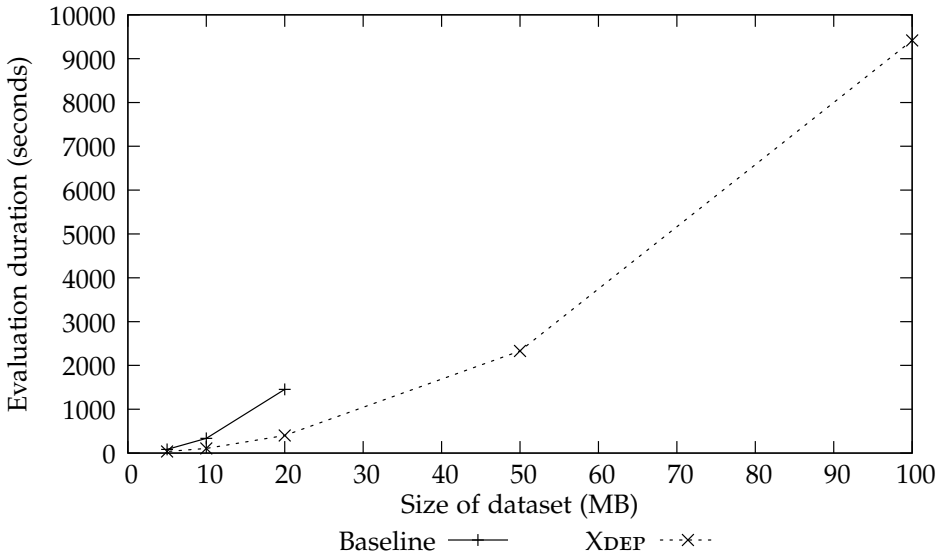

Figure 6.4: Evaluation times for query 10 in seconds

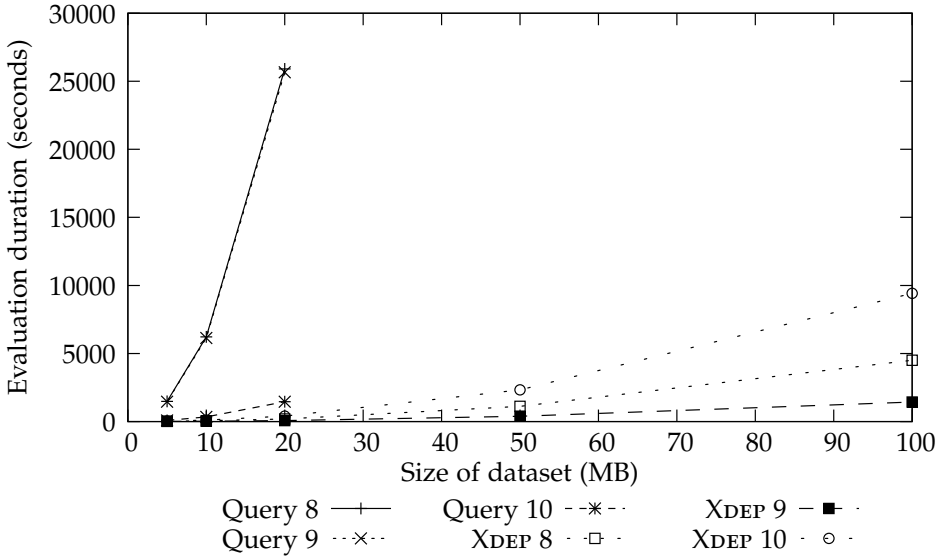

Figure 6.5: Comparison of query evaluation times by dataset size

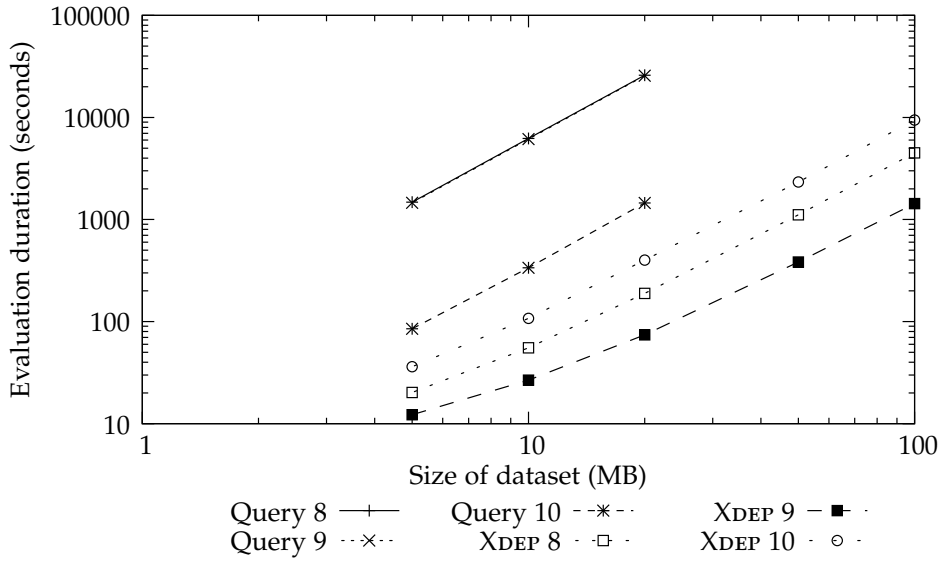

Figure 6.6: Same as Figure 6.5 but using a log-log scale

contains the same data as Figures 6.2, 6.3, and 6.4 for comparison. It is already clear that the improvement of XDEP is bigger for query 8 and 9 than it is for query 10.

Figure 6.6 uses a log-log scale instead. It shows not only a constant improvement of the XDEP optimised queries over plain XSPARQL, but the curve of the optimised queries become steeper with increasing dataset size and therefore become faster with increasing dataset size.

Another influence, as already outlined above, is the number of iterations of the inner *SparqlForClause*. Figure 6.7 depicts this relation. The distribution of the data makes a comparison difficult, but linear axes provide a direct way to see relations between the different data points.

Figure 6.8 uses a log-log scale and gives a better overview of the results. In this figure it becomes evident fewer inner iterations, as in query 10, lead to a minor improvement. While more inner iterations, as in query 8 and 9, lead to bigger improvements.

The performance gain of XDEP in relation to standard XSPARQL can also be shown as *performance gain factor*. Figure 6.9 and Table 6.6 show this factor, computed by dividing the evaluation runtime for standard XSPARQL evaluation by the evaluation runtime of the XDEP optimised queries. Since query evaluation of XSPARQL queries for datasets bigger than 20 MB took too long, the factor is only computable for datasets up to that size.

Figure 6.10 shows the performance gain factor dependent on the number of inner iterations. This visualisation shows again that the query with fewer

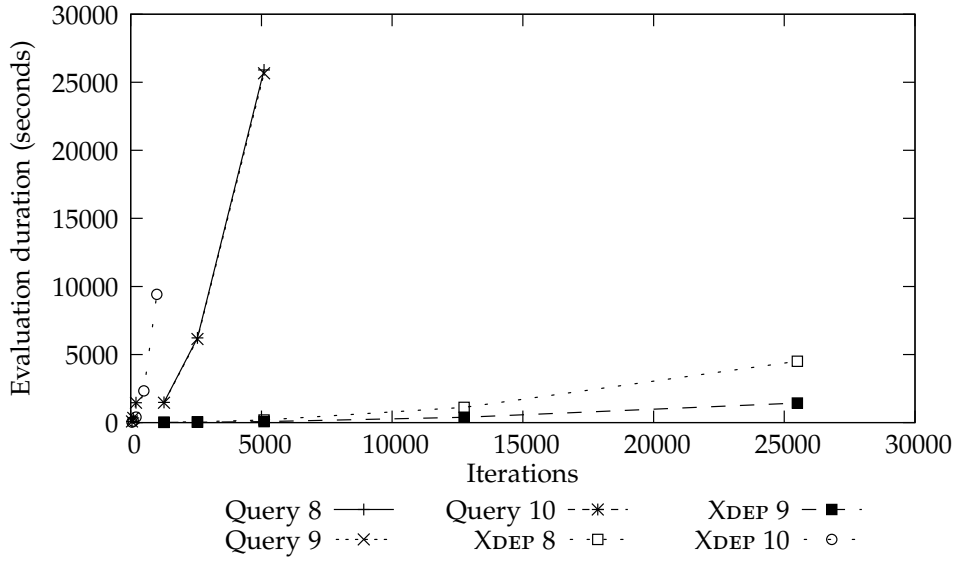

Figure 6.7: Comparison of query evaluation times by inner iterations

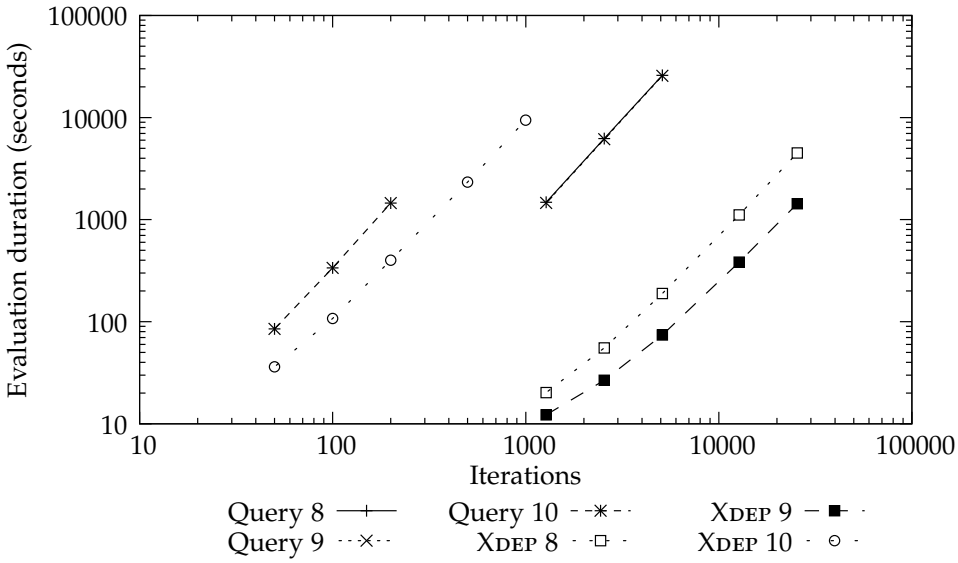

Figure 6.8: Same as Figure 6.7 but using a log-log scale

dataset #	Query 8	Query 9	Query 10
1	73.1	119.1	2.4
2	112.3	229.8	3.1
3	137.0	345.6	3.6

Table 6.6: XDEP performance gain factor

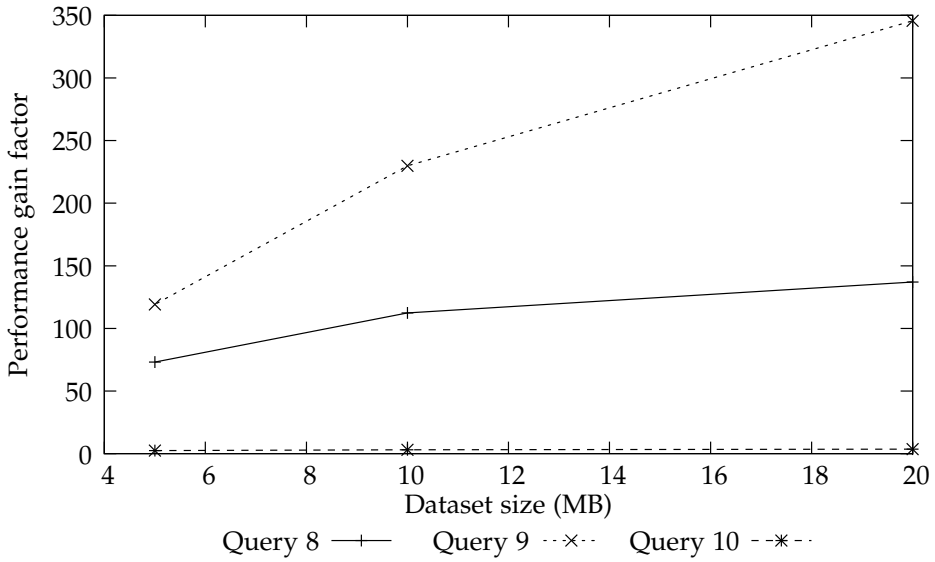

Figure 6.9: XDEP performance gain factor dependent on dataset size

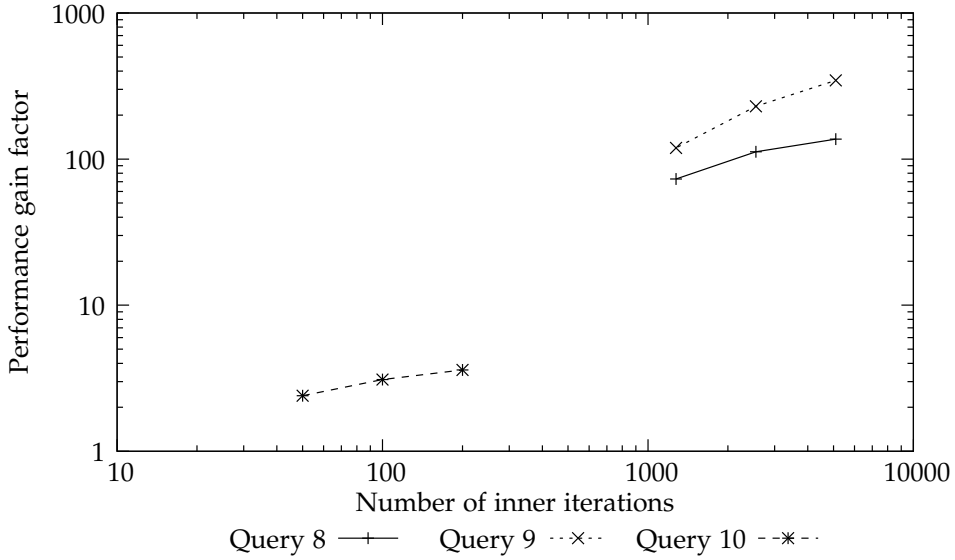

Figure 6.10: XDEP performance gain factor dependent on inner iterations

inner iterations (*SparqlForClauses*) improves less with XDEP than the queries with more inner iterations. But it seems that the number of inner iterations is not explaining all of the difference in the performance gain factor. Other factors include the concrete SPARQL implementation, especially its optimisation algorithms. The runtimes of SPARQL queries may depend heavily on the concrete serialisation of the *WhereClause*.

Figure 6.9 and Table 6.6 show Xdep as being an improvement over standard XSPARQL rewritings. Although the performance gain factor varies highly over the different queries, it is clear that the factor improves with increasing dataset size.

In summary three factors influencing the improvement achieved by Xdep could be identified:

1. We determine the *number of iterations of the inner SparqlForClause* influencing the improvement rate, or performance gain.
2. Since the evaluation time of a SPARQL query is also determined by the *size of the dataset*, the evaluation time of an XSPARQL query depends on that size too.
3. But there is another component, the structure of the SPARQL *WhereClause* of the inner *SparqlForClause*, which has an impact on final query evaluation result times too. The exact influence of this factor is harder to determine, since it can not be varied in the same way as the other two.

Chapter 7

Summary & Conclusions

In this thesis we presented the ongoing work of improving XSPARQL on multiple levels. We highlighted the problem of XML-RDF data integration, a possible solution by using XSPARQL and several issues when using XSPARQL in practise. We introduced new features thus adding use case scenarios. The first, Constructed Dataset, especially useful for complex RDF to RDF translations, introduces temporary RDF graphs to XSPARQL. Dependent on the data source, XSPARQL queries containing nested *SparqlForClause*s may show unintended behaviour, since variables can behave as being unbound. The second feature, Dataset Scoping, fixes this unintended behaviour by changing the native pattern matching of SPARQL in such a way that blank nodes matched from the data, are matched as constants in a nested loop "joins" between different SPARQL queries over the same dataset.

providing an intuitive way to join not only by literals and IRIs but also by blank nodes of different SPARQL queries. Furthermore we gave an elaborate formal semantics description of XSPARQL++, relying on the XQuery type system, and fixing minor issues of original XSPARQL.

Then we presented a new implementation based on standard compiler construction techniques. While still using the architecture of the former implementation, we instead use a rule-based rewriting generator, allowing straightforward implementation of rewriting rules. The new implementation is a modular and more maintainable, platform independent solution.

We presented Xdep, an approach for optimising XSPARQL queries containing nested *SparqlForClause*s, defined the constraints under which it is applicable, gave formal rules, as well as example rewritings. Furthermore we documented more practical and simpler optimisations.

To quantify the performance impact of the Xdep optimisation we performed a practical evaluation using the new implementation and the Xdep optimisation. We showed that unoptimised XSPARQL queries containing

nested *SparqlForClause*s have potential for optimisation. Thus we identified the communication between the XQuery engine and the SPARQL engine as the main bottleneck of the implementation architecture. We showed that we could use this optimisation potential with the presented Xdep dependent join optimisation, by showing a better performance for all tested queries on all tested dataset sizes.

7.1 Future Work

One way of extending XSPARQL in the future is to explore possible syntax simplifications to make query authoring easier.

Since the results of our evaluation are promising, new ways of optimising nested XSPARQL queries will be examined. A more detailed analysis of Xdep could reveal the factors influencing performance exactly, and therefore improve performance predictability. This analysis could be achieved by using other, possibly self made, micro benchmarks.

In order to increase the validity of our propositions about query evaluation times and measure performance in other use case scenarios we plan to perform SPARQL/RDF benchmarks, including support for reasoning on RDF data, such as the Berlin SPARQL benchmark [Bizer and Schultz, 2009] or the RDF repository benchmark in [Thakker et al., 2010].

Currently Xdep does not implement the Dataset Scoping semantics when the outer loop is a *SparqlForClause*. In the future we will extend the definition of Xdep to include the Dataset Scoping feature and make Xdep truly transparent to the query author.

One way of improving XSPARQL evaluation performance would be to redesign the interface between the XQuery and SPARQL engines. Communication over HTTP allows distant evaluation of the SPARQL parts of the query, but it also adds several different delays to query evaluation such as network latency or query/result encoding. By integrating SPARQL tighter to the XQuery engine, we could improve query evaluation time. We could use an extension mechanism of XQuery engines, be it an XQuery pragma or an implementation dependent API, to enable direct communication of the XQuery and SPARQL engines.

Appendix A

Grammars

After a short introduction in grammar notation we present the XSPARQL grammar in Section A.2, consisting of an XQuery derived part and a SPARQL derived part. Section A.3 presents the grammar for XML while Section A.4 shows the grammar for XML Namespaces. In Section A.5 we present the grammar of the RDF serialisation syntax Turtle.

A.1 Notation

All the following grammars use the notation defined by the XML specification [Bray et al., 2008]: The grammar production are of the following form:

```
[1] Symbol ::= Definition Expression
```

The rule number in square brackets in the beginning help to find a specific rule. Next is the symbol to be defined. There exists exactly one such production rule for each symbol. The main part of the rule consists of the definition of the symbol.

A.2 XSPARQL Grammar

This section provides a complete XSPARQL grammar not available in the XSPARQL specification. The grammar is an adaption of the grammars of XQuery [Boag et al., 2007] and SPARQL [Prud'hommeaux and Seaborne, 2008]. New production rules are marked with a prime symbol. Adapted rules are currently not marked at all. There is also a web version available[1] including internal links and markup for changed rules.

[1] see `http://stefanbischof.at/xsparql/grammar`

A.2.1 Changes

A few parts of the grammar rules of Akhtar et al. [2008] needed specific changes to produce a valid grammar

- Prefix Declaration was in the wrong place, added in rule XQuery[6];
- Base Declaration was missing, added in rule XQuery[6];
- Filter operators were mixed case, see rule SPARQL[57];
- We chose a different way to distinguish between the *TriplesSameSubject* for *SparqlForClause*s and the *TriplesSameSubject'* for *ConstructClause*s (rules SPARQL [32']–[43']);
- More general definition of constructed IRI (rules SPARQL [67'] and [68']), constructed literal (rules SPARQL [60'] and [66'], and constructed blank node (rule SPARQL [69']).

A.2.2 XQuery Grammar

The base for the XSPARQL grammar is the XQuery grammar [Boag et al., 2007]. Here the XQuery part of the XSPARQL grammar, containing the new clauses [33a] and [33b] and the changed rules [6] and [33].

Non Terminals

```
[1] Module ::= VersionDecl? (LibraryModule | MainModule)
[2] VersionDecl ::= "xquery" "version" StringLiteral ("encoding"
    StringLiteral)? Separator
[3] MainModule ::= Prolog QueryBody
[4] LibraryModule ::= ModuleDecl Prolog
[5] ModuleDecl ::= "module" "namespace" NCName "=" URILiteral Separator
[6] Prolog ::= BaseDecl?  (((DefaultNamespaceDecl | Setter |
    NamespaceDecl | ImportX) Separator)  | PrefixDecl  )* ((VarDecl |
    FunctionDecl | OptionDecl) Separator)*
[7] Setter ::= BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl |
    ConstructionDecl |
 OrderingModeDecl | EmptyOrderDecl | CopyNamespacesDecl
[8] ImportX ::= SchemaImport | ModuleImport
[9] Separator ::= ";"
[10] NamespaceDecl ::= "declare" "namespace" NCName "=" URILiteral
[11] BoundarySpaceDecl ::= "declare" "boundary-space" ("preserve" | "
    strip")
[12] DefaultNamespaceDecl ::= "declare" "default" ("element" | "function
    ") "namespace" URILiteral
[13] OptionDecl ::= "declare" "option" QName StringLiteral
```

```
[14] OrderingModeDecl ::= "declare" "ordering" ("ordered" | "unordered")
[15] EmptyOrderDecl ::= "declare" "default" "order" "empty" ("greatest"
    | "least")
[16] CopyNamespacesDecl ::= "declare" "copy-namespaces" PreserveMode ","
     InheritMode
[17] PreserveMode ::= "preserve" | "no-preserve"
[18] InheritMode ::= "inherit" | "no-inherit"
[19] DefaultCollationDecl ::= "declare" "default" "collation" URILiteral
[20] BaseURIDecl ::= "declare" "base-uri" URILiteral
[21] SchemaImport ::= "import" "schema" SchemaPrefix? URILiteral ("at"
    URILiteral ("," URILiteral)*)?
[22] SchemaPrefix ::= ("namespace" NCName "=") | ("default" "element" "
    namespace")
[23] ModuleImport ::= "import" "module" ("namespace" NCName "=")?
    URILiteral ("at" URILiteral ("," URILiteral)*)?
[24] VarDecl ::= "declare" "variable" "$" QName TypeDeclaration? ((":="
    ExprSingle) | "external")
[25] ConstructionDecl ::= "declare" "construction" ("strip" | "preserve
    ")
[26] FunctionDecl ::= "declare" "function" QName "(" ParamList? ")" ("as
    " SequenceType)? (EnclosedExpr | "external")
[27] ParamList ::= Param ("," Param)*
[28] Param ::= "$" QName TypeDeclaration?
[29] EnclosedExpr ::= "{" Expr "}"
[30] QueryBody ::= Expr
[31] Expr ::= ExprSingle ("," ExprSingle)*
[32] ExprSingle ::= FLWORExpr
 | QuantifiedExpr
 | TypeswitchExpr
 | IfExpr
 | OrExpr
[33] FLWORExpr ::= (ForClause | LetClause | SparqlForClause )+
    WhereClause? OrderByClause?
 "return" ExprSingle   ReturnClause
[33a] ReturnClause ::=
     "return" ExprSingle | "construct" ConstructTemplate
[33b] SparqlForClause ::=
     "for" "distinct"? ("$" VarName ("$" Varname)* | "*") DatasetClause "where" GroupGraphPattern So
[34] ForClause ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in
    " ExprSingle ("," "$" VarName
 TypeDeclaration? PositionalVar? "in" ExprSingle)*
[35] PositionalVar ::= "at" "$" VarName
[36] LetClause ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle
    ("," "$" VarName TypeDeclaration?
 ":=" ExprSingle)*
[37] WhereClause ::= "where" ExprSingle
```

```
[38] OrderByClause ::= (("order" "by") | ("stable" "order" "by"))
    OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest
    " | "least"))? ("collation"
 URILiteral)?
[42] QuantifiedExpr ::= ("some" | "every") "$" VarName TypeDeclaration?
    "in" ExprSingle ("," "$" VarName
 TypeDeclaration? "in" ExprSingle)* "satisfies" ExprSingle
[43] TypeswitchExpr ::= "typeswitch" "(" Expr ")" CaseClause+ "default"
    ("$" VarName)? "return"
 ExprSingle
[44] CaseClause ::= "case" ("$" VarName "as")? SequenceType "return"
    ExprSingle
[45] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[46] OrExpr ::= AndExpr ( "or" AndExpr )*
[47] AndExpr ::= ComparisonExpr ( "and" ComparisonExpr )*
[48] ComparisonExpr ::= RangeExpr ( (ValueComp
 | GeneralComp
 | NodeComp) RangeExpr )?
[49] RangeExpr ::= AdditiveExpr ( "to" AdditiveExpr )?
[50] AdditiveExpr ::= MultiplicativeExpr ( ("+" | "-")
    MultiplicativeExpr )*
[51] MultiplicativeExpr ::= UnionExpr ( ("*" | "div" | "idiv" | "mod")
    UnionExpr )*
[52] UnionExpr ::= IntersectExceptExpr ( ("union" | "|")
    IntersectExceptExpr )*
[53] IntersectExceptExpr ::= InstanceofExpr ( ("intersect" | "except")
    InstanceofExpr )*
[54] InstanceofExpr ::= TreatExpr ( "instance" "of" SequenceType )?
[55] TreatExpr ::= CastableExpr ( "treat" "as" SequenceType )?
[56] CastableExpr ::= CastExpr ( "castable" "as" SingleType )?
[57] CastExpr ::= UnaryExpr ( "cast" "as" SingleType )?
[58] UnaryExpr ::= ("-" | "+")* ValueExpr
[59] ValueExpr ::= ValidateExpr | PathExpr ~~| ExtensionExpr~~
[60] GeneralComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[61] ValueComp ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[62] NodeComp ::= "is" | "<<" | ">>"
[63] ValidateExpr ::= "validate" ValidationMode? "{" Expr "}"
[64] ValidationMode ::= "lax" | "strict"
[65] ExtensionExpr ::= Pragma+ "{" Expr? "}"
[66] Pragma ::= "(#" S? QName (S PragmaContents)? "#)" /* ws: explicit
    */
[67] PragmaContents ::= (Char* - (Char* '#)' Char*))
[68] PathExpr ::= ("/" RelativePathExpr?)
 | ("//" RelativePathExpr)
```

```
 | RelativePathExpr /* xgs: leading-lone-slash */
[69] RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*
[70] StepExpr ::= FilterExpr | AxisStep
[71] AxisStep ::= (ReverseStep | ForwardStep) PredicateList
[72] ForwardStep ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[73] ForwardAxis ::= ("child" "::")
 | ("descendant" "::")
 | ("attribute" "::")
 | ("self" "::")
 | ("descendant-or-self" "::")
 | ("following-sibling" "::")
 | ("following" "::")
[74] AbbrevForwardStep ::= "@"? NodeTest
[75] ReverseStep ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[76] ReverseAxis ::= ("parent" "::")
 | ("ancestor" "::")
 | ("preceding-sibling" "::")
 | ("preceding" "::")
 | ("ancestor-or-self" "::")
[77] AbbrevReverseStep ::= ".."
[78] NodeTest ::= KindTest | NameTest
[79] NameTest ::= QName | Wildcard
[80] Wildcard ::= "*"
 | (NCName ":" "*")
 | ("*" ":" NCName) /* ws: explicit */
[81] FilterExpr ::= PrimaryExpr PredicateList
[82] PredicateList ::= Predicate*
[83] Predicate ::= "[" Expr "]"
[84] PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr |
    ContextItemExpr | FunctionCall | OrderedExpr
 | UnorderedExpr | Constructor
[85] Literal ::= NumericLiteral | StringLiteral
[86] NumericLiteral ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
[87] VarRef ::= "$" VarName
[88] VarName ::= QName
[89] ParenthesizedExpr ::= "(" Expr? ")"
[90] ContextItemExpr ::= "."
[91] OrderedExpr ::= "ordered" "{" Expr "}"
[92] UnorderedExpr ::= "unordered" "{" Expr "}"
[93] FunctionCall ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")" /*
    xgs: reserved-function-names */
 /* gn: parens */
[94] Constructor ::= DirectConstructor
 | ComputedConstructor
[95] DirectConstructor ::= DirElemConstructor
 | DirCommentConstructor
 | DirPIConstructor
```

```
[96] DirElemConstructor ::= "<" QName DirAttributeList ("/>" | (">"
    DirElemContent* "</" QName S? ">"))
 /* ws: explicit */
[97] DirAttributeList ::= (S (QName S? "=" S? DirAttributeValue)?)* /*
    ws: explicit */
[98] DirAttributeValue ::= ('"' (EscapeQuot | QuotAttrValueContent)*
    '"')
 | ("'" (EscapeApos | AposAttrValueContent)* "'") /* ws: explicit */
[99] QuotAttrValueContent ::= QuotAttrContentChar
 | CommonContent
[100] AposAttrValueContent ::= AposAttrContentChar
 | CommonContent
[101] DirElemContent ::= DirectConstructor
 | CDataSection
 | CommonContent
 | ElementContentChar
[102] CommonContent ::= PredefinedEntityRef | CharRef | "{{" | "}}" |
    EnclosedExpr
[103] DirCommentConstructor ::= "<!--" DirCommentContents "-->" /* ws:
    explicit */
[104] DirCommentContents ::= ((Char - '-') | ('-' (Char - '-')))* /* ws:
     explicit */
[105] DirPIConstructor ::= "<?" PITarget (S DirPIContents)? "?>" /* ws:
    explicit */
[106] DirPIContents ::= (Char* - (Char* '?>' Char*)) /* ws: explicit */
[107] CDataSection ::= "<![CDATA[" CDataSectionContents "]]>" /* ws:
    explicit */
[108] CDataSectionContents ::= (Char* - (Char* ']]>' Char*)) /* ws:
    explicit */
[109] ComputedConstructor ::= CompDocConstructor
 | CompElemConstructor
 | CompAttrConstructor
 | CompTextConstructor
 | CompCommentConstructor
 | CompPIConstructor
[110] CompDocConstructor ::= "document" "{" Expr "}"
[111] CompElemConstructor ::= "element" (QName | ("{" Expr "}")) "{"
    ContentExpr? "}"
[112] ContentExpr ::= Expr
[113] CompAttrConstructor ::= "attribute" (QName | ("{" Expr "}")) "{"
    Expr? "}"
[114] CompTextConstructor ::= "text" "{" Expr "}"
[115] CompCommentConstructor ::= "comment" "{" Expr "}"
[116] CompPIConstructor ::= "processing-instruction" (NCName | ("{" Expr
     "}")) "{" Expr? "}"
[117] SingleType ::= AtomicType "?"?
[118] TypeDeclaration ::= "as" SequenceType
```

```
[119] SequenceType ::= ("empty-sequence" "(" ")")
 | (ItemType OccurrenceIndicator?)
[120] OccurrenceIndicator ::= "?" | "*" | "+" /* xgs: occurrence-
    indicators */
[121] ItemType ::= KindTest | ("item" "(" ")") | AtomicType
[122] AtomicType ::= QName
[123] KindTest ::= DocumentTest
 | ElementTest
 | AttributeTest
 | SchemaElementTest
 | SchemaAttributeTest
 | PITest
 | CommentTest
 | TextTest
 | AnyKindTest
[124] AnyKindTest ::= "node" "(" ")"
[125] DocumentTest ::= "document-node" "(" (ElementTest |
    SchemaElementTest)? ")"
[126] TextTest ::= "text" "(" ")"
[127] CommentTest ::= "comment" "(" ")"
[128] PITest ::= "processing-instruction" "(" (NCName | StringLiteral)?
    ")"
[129] AttributeTest ::= "attribute" "(" (AttribNameOrWildcard ("," 
    TypeName)?)? ")"
[130] AttribNameOrWildcard ::= AttributeName | "*"
[131] SchemaAttributeTest ::= "schema-attribute" "("
    AttributeDeclaration ")"
[132] AttributeDeclaration ::= AttributeName
[133] ElementTest ::= "element" "(" (ElementNameOrWildcard ("," TypeName
     "?"?)?)? ")"
[134] ElementNameOrWildcard ::= ElementName | "*"
[135] SchemaElementTest ::= "schema-element" "(" ElementDeclaration ")"
[136] ElementDeclaration ::= ElementName
[137] AttributeName ::= QName
[138] ElementName ::= QName
[139] TypeName ::= QName
[140] URILiteral ::= StringLiteral
```

Terminals

```
[141] IntegerLiteral ::= Digits
[142] DecimalLiteral ::= ("." Digits) | (Digits "." [0-9]*)
[143] DoubleLiteral ::= (("." Digits) | (Digits ("." [0-9]*)?)) [eE]
    [+-]? Digits
[144] StringLiteral ::= ('"' (PredefinedEntityRef | CharRef | EscapeQuot
     | [^"&])* '"') | ("'" (PredefinedEntityRef | CharRef | EscapeApos |
     [^'&])* "'")
```

```
[145] PredefinedEntityRef ::= "&" ("lt" | "gt" | "amp" | "quot" | "apos
    ") ";"
[146] EscapeQuot ::= '""'
[147] EscapeApos ::= "''"
[148] ElementContentChar ::= Char - [{}<&]
[149] QuotAttrContentChar ::= Char - ["{}<&]
[150] AposAttrContentChar ::= Char - ['{}<&]
[151] Comment ::= ~~"(:" (CommentContents | Comment)* ":)"~~
[152] PITarget ::= see PITarget in XML grammar
[153] CharRef ::= see CharRef in XML grammar
[154] QName ::= see QName in XML Namespace grammar
[155] NCName ::= see NCName in XML Namespace grammar
[156] S ::= see NT-S in XML grammar
[157] Char ::= see Char in XML grammar

   The following symbols are used only in the definition of terminal
       symbols; they are not terminal symbols in the above grammar.

[158] Digits ::= [0-9]+
[159] CommentContents ::= (Char+ - (Char* ('(:' | ':)') Char*))
```

A.2.3 SPARQL Grammar

The second building block of the XSPARQL grammar is the SPARQL grammar[Prud'hommeaux and Seaborne, 2008]. In the following grammar part the rules [12] and [57] were changed. We added the rules [32']–[43'], [45'], [60'], and [66']–[69'].

Non Terminals

```
[1] Query ::= Prologue
 ( SelectQuery | ConstructQuery | DescribeQuery | AskQuery )
[2] Prologue ::= BaseDecl? PrefixDecl*
[3] BaseDecl ::= 'BASE' IRI_REF
[4] PrefixDecl ::= 'PREFIX' PNAME_NS IRI_REF
[5] SelectQuery ::= 'SELECT' ( 'DISTINCT' | 'REDUCED' )? ( Var+ | '*' )
    DatasetClause* WhereClause
 SolutionModifier
[6] ConstructQuery ::= 'CONSTRUCT' ConstructTemplate DatasetClause*
    WhereClause SolutionModifier
[7] DescribeQuery ::= 'DESCRIBE' ( VarOrIRIref+ | '*' ) DatasetClause*
    WhereClause? SolutionModifier
[8] AskQuery ::= 'ASK' DatasetClause* WhereClause
[9] DatasetClause ::= 'FROM' ( DefaultGraphClause | NamedGraphClause)
[10] DefaultGraphClause ::= SourceSelector
[11] NamedGraphClause ::= 'NAMED' SourceSelector
[12] SourceSelector ::= IRIref | Var
```

```
[13] WhereClause ::= 'WHERE'? GroupGraphPattern
[14] SolutionModifier ::= OrderClause? LimitOffsetClauses?
[15] LimitOffsetClauses ::= ( LimitClause OffsetClause? | OffsetClause
    LimitClause? )
[16] OrderClause ::= 'ORDER' 'BY' OrderCondition+
[17] OrderCondition ::= ( ( 'ASC' | 'DESC' ) BracketedExpression )
 | ( Constraint | Var )
[18] LimitClause ::= 'LIMIT' INTEGER
[19] OffsetClause ::= 'OFFSET' INTEGER
[20] GroupGraphPattern ::= '{' TriplesBlock? ( ( GraphPatternNotTriples
    | Filter ) '.'? TriplesBlock? )*
 '}'
[21] TriplesBlock ::= TriplesSameSubject ( '.' TriplesBlock? )?
[22] GraphPatternNotTriples ::= OptionalGraphPattern |
    GroupOrUnionGraphPattern | GraphGraphPattern
[23] OptionalGraphPattern ::= 'OPTIONAL' GroupGraphPattern
[24] GraphGraphPattern ::= 'GRAPH' VarOrIRIref GroupGraphPattern
[25] GroupOrUnionGraphPattern ::= GroupGraphPattern ( 'UNION'
    GroupGraphPattern )*
[26] Filter ::= 'FILTER' Constraint
[27] Constraint ::= BracketedExpression | BuiltInCall | FunctionCall
[28] FunctionCall ::= IRIref ArgList
[29] ArgList ::= ( NIL | '(' Expression ( ',' Expression )* ')' )
[30] ConstructTemplate ::= '{' ConstructTriples? '}'
[31] ConstructTriples ::= TriplesSameSubject' ( '.' ConstructTriples? )?
[32] TriplesSameSubject ::= VarOrTerm PropertyListNotEmpty | TriplesNode
     PropertyList
[32'] TriplesSameSubject' ::=VarOrTerm' PropertyListNotEmpty' |
 TriplesNode' PropertyList'
[33] PropertyListNotEmpty ::= Verb ObjectList ( ';' ( Verb ObjectList )?
     )*
[33'] PropertyListNotEmpty' ::= Verb' ObjectList' ( ';' ( Verb'
    ObjectList' )? )*
[34] PropertyList ::= PropertyListNotEmpty?
[34'] PropertyList' ::= PropertyListNotEmpty'?
[35] ObjectList ::= Object ( ',' Object )*
[35'] ObjectList' ::= Object' ( ',' Object' )*
[36] Object ::= GraphNode
[36'] Object' ::= GraphNode'
[37] Verb ::= VarOrIRIref | 'a'
[37'] Verb' ::= VarOrIRIref' | 'a'
[38] TriplesNode ::= Collection | BlankNodePropertyList
[38'] TriplesNode' ::= Collection' | BlankNodePropertyList'
[39] BlankNodePropertyList ::= '[' PropertyListNotEmpty ']'
[39'] BlankNodePropertyList' ::= '[' PropertyListNotEmpty' ']'
[40] Collection ::= '(' GraphNode+ ')'
[40'] Collection' ::= '(' GraphNode'+ ')'
```

```
[41] GraphNode ::= VarOrTerm | TriplesNode
[41'] GraphNode' ::= VarOrTerm' | TriplesNode'
[42] VarOrTerm ::= Var | GraphTerm
[42'] VarOrTerm' ::= Var | GraphTerm'
[43] VarOrIRIref ::= Var | IRIref
[43'] VarOrIRIref' ::= Var | IRIref'
[44] Var ::= VAR1 | VAR2
[45] GraphTerm ::= IRIref | RDFLiteral | NumericLiteral | BooleanLiteral
     | BlankNode | NIL
[45'] GraphTerm' ::= IRIref' | RDFLiteral' | NumericLiteral |
    BooleanLiteral | BlankNode' |
 NIL
[46] Expression ::= ConditionalOrExpression
[47] ConditionalOrExpression ::= ConditionalAndExpression ( '||'
    ConditionalAndExpression )*
[48] ConditionalAndExpression ::= ValueLogical ( '&&' ValueLogical )*
[49] ValueLogical ::= RelationalExpression
[50] RelationalExpression ::= NumericExpression ( '=' NumericExpression
    | '!=' NumericExpression | '<'
 NumericExpression | '>' NumericExpression | '<=' NumericExpression |
     '>=' NumericExpression )?
[51] NumericExpression ::= AdditiveExpression
[52] AdditiveExpression ::= MultiplicativeExpression ( '+'
    MultiplicativeExpression | '-'
 MultiplicativeExpression | NumericLiteralPositive |
     NumericLiteralNegative )*
[53] MultiplicativeExpression ::= UnaryExpression ( '*' UnaryExpression
    | '/' UnaryExpression )*
[54] UnaryExpression ::=  '!' PrimaryExpression
 | '+' PrimaryExpression
 | '-' PrimaryExpression
 | PrimaryExpression
[55] PrimaryExpression ::= BrackettedExpression | BuiltInCall |
    IRIrefOrFunction | RDFLiteral |
 NumericLiteral | BooleanLiteral | Var
[56] BrackettedExpression ::= '(' Expression ')'
[57] BuiltInCall ::=  'str' '(' Expression ')'
 | 'lang' '(' Expression ')'
 | 'langmatches' '(' Expression ',' Expression ')'
 | 'datatype' '(' Expression ')'
 | 'bound' '(' Var ')'
 | 'sameterm' '(' Expression ',' Expression ')'
 | 'isiri' '(' Expression ')'
 | 'isuri' '(' Expression ')'
 | 'isblank' '(' Expression ')'
 | 'isliteral' '(' Expression ')'
 | RegexExpression
```

```
[58] RegexExpression ::= 'REGEX' '(' Expression ',' Expression ( ','
    Expression )? ')'
[59] IRIrefOrFunction ::= IRIref ArgList?
[60] RDFLiteral ::= String ( LANGTAG | ( '^^' IRIref ) )?
[60']RDFLiteral' ::= String' ( LANGTAG
    | '@{' FLWORExpr '}' | ( '^^' IRIref') )?
[61] NumericLiteral ::= NumericLiteralUnsigned | NumericLiteralPositive
    | NumericLiteralNegative
[62] NumericLiteralUnsigned ::= INTEGER | DECIMAL | DOUBLE
[63] NumericLiteralPositive ::= INTEGER_POSITIVE | DECIMAL_POSITIVE |
    DOUBLE_POSITIVE
[64] NumericLiteralNegative ::= INTEGER_NEGATIVE | DECIMAL_NEGATIVE |
    DOUBLE_NEGATIVE
[65] BooleanLiteral ::= 'true' | 'false'
[66] String ::= STRING_LITERAL1 | STRING_LITERAL2 | STRING_LITERAL_LONG1
     | STRING_LITERAL_LONG2
[66']String' ::= STRING_LITERAL1 | STRING_LITERAL2 |
    STRING_LITERAL_LONG1 |
 STRING_LITERAL_LONG2 | '{' FLWORExpr '}'
[67] IRIref ::= IRI_REF | PrefixedName
[67']IRIref' ::= IRI_REF | '<{' FLWORExpr '}>' | PrefixedName'
[68] PrefixedName ::= PNAME_LN | PNAME_NS
[68']PrefixedName' ::=PNAME_LN ( (PN_PREFIX | '{' FLWORExpr '}')? ':'
 (PN_LOCAL | '{' FLWORExpr '}') ) | PNAME_NS
[69] BlankNode ::= BLANK_NODE_LABEL | ANON
[69']BlankNode' ::= BLANK_NODE_LABEL | ANON | '_:{' FLWORExpr '}'
```

Terminals

```
[70] IRI_REF ::= '<' ([^<>"{}|^'\]-[#x00-#x20])* '>'
[71] PNAME_NS ::= PN_PREFIX? ':'
[72] PNAME_LN ::= PNAME_NS PN_LOCAL
[73] BLANK_NODE_LABEL ::= '_:' PN_LOCAL
[74] VAR1 ::= '?' VARNAME
[75] VAR2 ::= '$' VARNAME
[76] LANGTAG ::= '@' [a-zA-Z]+ ('-' [a-zA-Z0-9]+)*
[77] INTEGER ::=[0-9]+
[78] DECIMAL ::=[0-9]+ '.' [0-9]* | '.' [0-9]+
[79] DOUBLE ::=[0-9]+ '.' [0-9]* EXPONENT | '.' ([0-9])+ EXPONENT |
    ([0-9])+ EXPONENT
[80] INTEGER_POSITIVE ::= '+' INTEGER
[81] DECIMAL_POSITIVE ::= '+' DECIMAL
[82] DOUBLE_POSITIVE ::= '+' DOUBLE
[83] INTEGER_NEGATIVE ::= '-' INTEGER
[84] DECIMAL_NEGATIVE ::= '-' DECIMAL
[85] DOUBLE_NEGATIVE ::= '-' DOUBLE
[86] EXPONENT ::=[eE] [+-]? [0-9]+
[87] STRING_LITERAL1 ::= "'" ( ([^#x27#x5C#xA#xD]) | ECHAR )* "'"
```

```
[88] STRING_LITERAL2 ::= '"' ( ([^#x22#x5C#xA#xD]) | ECHAR )* '"'
[89] STRING_LITERAL_LONG1 ::= "'''" ( ( "'" | "''" )? ( [^'\] | ECHAR )
    )* "'''"
[90] STRING_LITERAL_LONG2 ::= '"""' ( ( '"' | '""' )? ( [^"\] | ECHAR )
    )* '"""'
[91] ECHAR ::= '\' [tbnrf\"']
[92] NIL ::= '(' WS* ')'
[93] WS ::= #x20 | #x9 | #xD | #xA
[94] ANON ::= '[' WS* ']'
[95] PN_CHARS_BASE ::=[A-Z] | [a-z] | [#x00C0-#x00D6] | [#x00D8-#x00F6]
    | [#x00F8-#x02FF] | [#x0370-#x037D] | [#x037F-#x1FFF] | [#x200C-#
    x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] | [#x3001-#xD7FF] | [#
    xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]
[96] PN_CHARS_U ::= PN_CHARS_BASE | '_'
[97] VARNAME ::= ( PN_CHARS_U - '_' | [0-9] ) ( PN_CHARS_U | [0-9] | #
    x00B7 | [#x0300-#x036F] | [#x203F-#x2040] )*
[98] PN_CHARS ::= PN_CHARS_U | '-' | [0-9] | #x00B7 | [#x0300-#x036F] |
    [#x203F-#x2040]
[99] PN_PREFIX ::= PN_CHARS_BASE ((PN_CHARS|'.')* PN_CHARS)?
[100] PN_LOCAL ::= ( PN_CHARS_U | [0-9] ) ((PN_CHARS|'.')* PN_CHARS)?
 Note that SPARQL local names allow leading digits while XML local names
      do not.
```

A.3 XML Grammar

This section gives the grammar production rules for XML [Bray et al., 2008].

```
[1] document ::=  prolog element Misc*
[2] Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#
    x10000-#x10FFFF] /* any Unicode character, excluding the surrogate
    blocks, FFFE, and FFFF. */
[3] S ::= (#x20 | #x9 | #xD | #xA)+
[4] NameStartChar ::= ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#
    xF6] | [#xF8-#x2FF] | [#x370-#x37D] | [#x37F-#x1FFF] | [#x200C-#
    x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] | [#x3001-#xD7FF] | [#
    xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]
[4a] NameChar ::= NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-#
    x036F] | [#x203F-#x2040]
[5] Name ::= NameStartChar (NameChar)*
[6] Names ::= Name (#x20 Name)*
[7] Nmtoken ::= (NameChar)+
[8] Nmtokens ::= Nmtoken (#x20 Nmtoken)*
[9] EntityValue ::= '"' ([^%&"] | PEReference | Reference)* '"'
|  "'" ([^%&'] | PEReference | Reference)* "'"
[10] AttValue ::= '"' ([^<&"] | Reference)* '"'
|  "'" ([^<&'] | Reference)* "'"
[11] SystemLiteral ::= ('"' [^"]* '"') | ("'" [^']* "'")
[12] PubidLiteral ::= '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
```

```
[13] PubidChar ::= #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_
    %]
[14] CharData ::= [^<&]* - ([^<&]* ']]>' [^<&]*)
[15] Comment ::= '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
[16] PI ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17] PITarget ::=  Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
[18] CDSect ::=  CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20] CData ::= (Char* - (Char* ']]>' Char*))
[21] CDEnd ::= ']]>'
[22] prolog ::=  XMLDecl? Misc* (doctypedecl Misc*)?
[23] XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24] VersionInfo ::=  S 'version' Eq ("'" VersionNum "'" | '"'
    VersionNum '"')
[25] Eq ::=  S? '=' S?
[26] VersionNum ::= '1.' [0-9]+
[27] Misc ::=  Comment | PI | S
[28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('['
    intSubset ']' S?)? '>'  [VC: Root Element Type]
[WFC: External Subset]
[28a] DeclSep ::=  PEReference | S  [WFC: PE Between Declarations]
[28b] intSubset ::= (markupdecl | DeclSep)*
[29] markupdecl ::=  elementdecl | AttlistDecl | EntityDecl |
    NotationDecl | PI | Comment [VC: Proper Declaration/PE Nesting]
[WFC: PEs in Internal Subset]
[30] extSubset ::=  TextDecl? extSubsetDecl
[31] extSubsetDecl ::= ( markupdecl | conditionalSect | DeclSep)*
[32] SDDecl ::=  S 'standalone' Eq (("'" ('yes' | 'no') "'") | ('"' ('
    yes' | 'no') '"'))
(Productions 33 through 38 have been removed.)
[39] element ::=  EmptyElemTag
| STag content ETag [WFC: Element Type Match]
[VC: Element Valid]
[40] STag ::= '<' Name (S Attribute)* S? '>'  [WFC: Unique Att Spec]
[41] Attribute ::=  Name Eq AttValue  [VC: Attribute Value Type]
[WFC: No External Entity References]
[WFC: No < in Attribute Values]
[42] ETag ::= '</' Name S? '>'
[43] content ::=  CharData? ((element | Reference | CDSect | PI |
    Comment) CharData?)*
[44] EmptyElemTag ::= '<' Name (S Attribute)* S? '/>' [WFC: Unique Att
    Spec]
[45] elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'  [VC:
    Unique Element Type Declaration]
[46] contentspec ::= 'EMPTY' | 'ANY' | Mixed | children
[47] children ::= (choice | seq) ('?' | '*' | '+')?
[48] cp ::= (Name | choice | seq) ('?' | '*' | '+')?
```

```
[49] choice ::= '(' S? cp ( S? '|' S? cp )+ S? ')'  [VC: Proper Group/PE
     Nesting]
[50] seq ::= '(' S? cp ( S? ',' S? cp )* S? ')' [VC: Proper Group/PE
    Nesting]
[51] Mixed ::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
| '(' S? '#PCDATA' S? ')' [VC: Proper Group/PE Nesting]
[VC: No Duplicate Types]
[52] AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
[53] AttDef ::=  S Name S AttType S DefaultDecl
[54] AttType ::=  StringType | TokenizedType | EnumeratedType
[55] StringType ::= 'CDATA'
[56] TokenizedType ::= 'ID' [VC: ID]
[VC: One ID per Element Type]
[VC: ID Attribute Default]
| 'IDREF' [VC: IDREF]
| 'IDREFS'  [VC: IDREF]
| 'ENTITY'  [VC: Entity Name]
| 'ENTITIES'  [VC: Entity Name]
| 'NMTOKEN' [VC: Name Token]
| 'NMTOKENS'  [VC: Name Token]
[57] EnumeratedType ::=  NotationType | Enumeration
[58] NotationType ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
    [VC: Notation Attributes]
[VC: One Notation Per Element Type]
[VC: No Notation on Empty Element]
[VC: No Duplicate Tokens]
[59] Enumeration ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')' [VC:
    Enumeration]
[VC: No Duplicate Tokens]
[60] DefaultDecl ::= '#REQUIRED' | '#IMPLIED'
| (('#FIXED' S)? AttValue)  [VC: Required Attribute]
[VC: Attribute Default Value Syntactically Correct]
[WFC: No < in Attribute Values]
[VC: Fixed Attribute Default]
[WFC: No External Entity References]
[61] conditionalSect ::=  includeSect | ignoreSect
[62] includeSect ::= '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'  [VC:
     Proper Conditional Section/PE Nesting]
[63] ignoreSect ::= '<![' S? 'IGNORE' S? '[' ignoreSectContents* ']]>'
    [VC: Proper Conditional Section/PE Nesting]
[64] ignoreSectContents ::=  Ignore ('<![' ignoreSectContents ']]>'
    Ignore)*
[65] Ignore ::=  Char* - (Char* ('<![' | ']]>') Char*)
[66] CharRef ::= '&#' [0-9]+ ';'
| '&#x' [0-9a-fA-F]+ ';'  [WFC: Legal Character]
[67] Reference ::=  EntityRef | CharRef
[68] EntityRef ::= '&' Name ';' [WFC: Entity Declared]
```

```
[VC: Entity Declared]
[WFC: Parsed Entity]
[WFC: No Recursion]
[69] PEReference ::= '%' Name ';' [VC: Entity Declared]
[WFC: No Recursion]
[WFC: In DTD]
[70] EntityDecl ::=  GEDecl | PEDecl
[71] GEDecl ::= '<!ENTITY' S Name S EntityDef S? '>'
[72] PEDecl ::= '<!ENTITY' S '%' S Name S PEDef S? '>'
[73] EntityDef ::=  EntityValue | (ExternalID NDataDecl?)
[74] PEDef ::=  EntityValue | ExternalID
[75] ExternalID ::= 'SYSTEM' S SystemLiteral
| 'PUBLIC' S PubidLiteral S SystemLiteral
[76] NDataDecl ::=  S 'NDATA' S Name  [VC: Notation Declared]
[77] TextDecl ::= '<?xml' VersionInfo? EncodingDecl S? '?>'
[78] extParsedEnt ::=  TextDecl? content
[80] EncodingDecl ::=  S 'encoding' Eq ('"' EncName '"' | "'" EncName
    "'" )
[81] EncName ::= [A-Za-z] ([A-Za-z0-9._] | '-')*  /* Encoding name
    contains only Latin characters */
[82] NotationDecl ::= '<!NOTATION' S Name S (ExternalID | PublicID) S?
    '>'  [VC: Unique Notation Name]
[83] PublicID ::= 'PUBLIC' S PubidLiteral
```

A.4 XML Namespaces Grammar

XML Namespaces [Layman et al., 1999], as being an important part of the XML world, are given in this section.

```
[1] NSAttName ::= PrefixedAttName | DefaultAttName
[2] PrefixedAttName ::= 'xmlns:' NCName [NSC: Reserved Prefixes and
    Namespace Names]
[3] DefaultAttName ::= 'xmlns'
[4] NCName ::= Name - (Char* ':' Char*) /* An XML Name, minus the ":" */
[5] NCNameChar Orphaned
[6] NCNameStartChar Orphaned
[7] QName ::= PrefixedName | UnprefixedName
[8] PrefixedName ::=  Prefix ':' LocalPart
[9] UnprefixedName ::=  LocalPart
[10] Prefix ::= NCName
[11] LocalPart ::= NCName
[12] STag ::= '<' QName (S Attribute)* S? '>' [NSC: Prefix Declared]
[13] ETag ::= '</' QName S? '>' [NSC: Prefix Declared]
[14] EmptyElemTag ::= '<' QName (S Attribute)* S? '/>'  [NSC: Prefix
    Declared]
```

```
[15] Attribute ::= NSAttName Eq AttValue | QName Eq AttValue  [NSC:
    Prefix Declared]
[NSC: No Prefix Undeclaring]
[NSC: Attributes Unique]
[16] doctypedecl ::= '<!DOCTYPE' S QName (S ExternalID)? S? ('[' (
    markupdecl | PEReference | S)* ']' S?)? '>'
[17] elementdecl ::= '<!ELEMENT' S QName S contentspec S? '>'
[18] cp ::= (QName | choice | seq) ('?' | '*' | '+')?
[19] Mixed ::= '(' S? '#PCDATA' (S? '|' S? QName)* S? ')*' | '(' S? '#
    PCDATA' S? ')'
[20] AttlistDecl ::= '<!ATTLIST' S QName AttDef* S? '>'
[21] AttDef ::= S (QName | NSAttName) S AttType S DefaultDecl
```

A.5 Turtle Grammar

Turtle [Beckett and Berners-Lee, 2008] is a more concise and intuitive RDF serialisation syntax than RDF/XML.

```
[1] turtleDoc ::= statement*
[2] statement ::= directive '.' | triples '.' | ws+
[3] directive ::= prefixID | base
[4] prefixID  ::= '@prefix' ws+ prefixName? ':' uriref
[5] base  ::= '@base' ws+ uriref
[6] triples ::= subject predicateObjectList
[7] predicateObjectList ::= verb objectList ( ';' verb objectList )* (
    ';')?
[8] objectList  ::= object ( ',' object)*
[9] verb  ::= predicate | 'a'
[10]  comment ::= '#' ( [^#xA#xD] )*
[11]  subject ::= resource | blank
[12]  predicate ::= resource
[13]  object  ::= resource | blank | literal
[14]  literal ::= quotedString ( '@' language )? | datatypeString |
    integer | double | decimal | boolean
[15]  datatypeString  ::= quotedString '^^' resource
[16]  integer ::= ('-' | '+') ? [0-9]+
[17]  double  ::= ('-' | '+') ? ( [0-9]+ '.' [0-9]* exponent | '.'
    ([0-9])+ exponent | ([0-9])+ exponent )
[18]  decimal ::= ('-' | '+')? ( [0-9]+ '.' [0-9]* | '.' ([0-9])+ |
    ([0-9])+ )
[19]  exponent  ::= [eE] ('-' | '+')? [0-9]+
[20]  boolean ::= 'true' | 'false'
[21]  blank ::= nodeID | '[]' | '[' predicateObjectList ']' | collection
[22]  itemList  ::= object+
[23]  collection  ::= '(' itemList? ')'
[24]  ws  ::= #x9 | #xA | #xD | #x20 | comment
```

```
[25]  resource  ::= uriref | qname
[26]  nodeID  ::= '_:' name
[27]  qname ::= prefixName? ':' name?
[28]  uriref  ::= '<' relativeURI '>'
[29]  language  ::= [a-z]+ ('-' [a-z0-9]+ )*
[30]  nameStartChar ::= [A-Z] | "_" | [a-z] | [#x00C0-#x00D6] | [#x00D8
    -#x00F6] | [#x00F8-#x02FF] | [#x0370-#x037D] | [#x037F-#x1FFF] | [#
    x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] | [#x3001-#xD7FF]
    | [#xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]
[31]  nameChar  ::= nameStartChar | '-' | [0-9] | #x00B7 | [#x0300-#
    x036F] | [#x203F-#x2040]
[32]  name  ::= nameStartChar nameChar*
[33]  prefixName  ::= ( nameStartChar - '_' ) nameChar*
[34]  relativeURI ::= ucharacter*
[35]  quotedString  ::= string | longString
[36]  string  ::= #x22 scharacter* #x22
[37]  longString  ::= #x22 #x22 #x22 lcharacter* #x22 #x22 #x22
[38]  character ::=  '\u' hex hex hex hex |
'\U' hex hex hex hex hex hex hex hex |
'\\' |
[#x20-#x5B] | [#x5D-#x10FFFF]
[39]  echaracter  ::= character | '\t' | '\n' | '\r'
[40]  hex ::= [#x30-#x39] | [#x41-#x46]
[41]  ucharacter  ::=  ( character - #x3E ) | '\>'
[42]  scharacter  ::=  ( echaracter - #x22 ) | '\"'
[43]  lcharacter  ::= echaracter | '\"' | #x9 | #xA | #xD
```

Appendix B

Evaluation Queries

The queries for the evaluation in Chapter 6 are presented in this appendix. Three sets of queries were used: the original XMark [Schmidt et al., 2002] queries shown in Section B.1, the XMark queries translated to XSPARQL shown in Section B.2 and the optimised XSPARQL queries 8, 9 and 10 presented in Section B.3.

In the following list we give the original explanation[1] for every XMark query from query #1 to query #20. The query explanations apply for the queries of all following sections.

1. Return the name of the person with ID `person0`.
2. Return the initial increases of all open auctions.
3. Return the IDs of all open auctions whose current increase is at least twice as high as the initial increase.
4. List the reserves of those open auctions where a certain person issued a bid before another person.
5. How many sold items cost more than 40?
6. How many items are listed on all continents?
7. How many pieces of prose are in our database?
8. List the names of persons and the number of items they bought. (joins person, closed_auction)
9. List the names of persons and the names of the items they bought in Europe. (joins person, closed_auction, item)

[1] as given by the official XMark queries at `http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt`

10. List all persons according to their interest; use French markup in the result.
11. For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.
12. For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.
13. List the names of items registered in Australia along with their descriptions.
14. Return the names of all items whose description contains the word 'gold'.
15. Print the keywords in emphasis in annotations of closed auctions.
16. Return the IDs of those auctions that have one or more keywords in emphasis. (cf. Q15)
17. Which persons don't have a homepage?
18. Convert the currency of the reserve of all open auctions to another currency.
19. Give an alphabetically ordered list of all items along with their location.
20. Group customers by their income and output the cardinality of each group.

B.1 Original XMark Benchmark Queries

The first set of benchmark queries are the original XMark benchmark queries[2]. The only thing changed, is the way to access the XML source document: instead of using the *fn:doc* function, the query URL is passed to the query as external variable. This measure simplified benchmark execution.

```
1 declare variable $graph external;
2
3 let $auction := doc($graph) return
4 for $b in $auction/site/people/person[@id = "person0"] return $b/name/
    text()
```

Listing B.1: XMark Query 1

[2] http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt

```
declare variable $graph external;

let $auction := doc($graph) return
for $b in $auction/site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>
```

Listing B.2: XMark Query 2

```
declare variable $graph external;

let $auction := doc($graph) return
for $b in $auction/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2 <= $b/bidder[last()
    ]/increase/text()
return
  <increase
  first="{$b/bidder[1]/increase/text()}"
  last="{$b/bidder[last()]/increase/text()}"/>
```

Listing B.3: XMark Query 3

```
declare variable $graph external;

let $auction := doc($graph) return
for $b in $auction/site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

Listing B.4: XMark Query 4

```
declare variable $graph external;

let $auction := doc($graph) return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

Listing B.5: XMark Query 5

```
declare variable $graph external;

let $auction := doc($graph) return
for $b in $auction//site/regions return count($b//item)
```

Listing B.6: XMark Query 6

```
declare variable $graph external;

let $auction := doc($graph) return
for $p in $auction/site
return
  count($p//description) + count($p//annotation) + count($p//
      emailaddress)
```

Listing B.7: XMark Query 7

```
declare variable $graph external;

let $auction := doc($graph) return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Listing B.8: XMark Query 8

```
declare variable $graph external;

let $auction := doc($graph) return
let $ca := $auction/site/closed_auctions/closed_auction return
let
    $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
    let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2
    return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

Listing B.9: XMark Query 9

```
declare variable $graph external;

let $auction := doc($graph) return
for $i in
  distinct-values($auction/site/people/person/profile/interest/@category
      )
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

Listing B.10: XMark Query 10

```
declare variable $graph external;

let $auction := doc($graph) return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{$p/name/text()}">{count($l)}</items>
```

Listing B.11: XMark Query 11

```
1
2  declare variable $graph external;
3
4  let $auction := doc($graph) return
5  for $p in $auction/site/people/person
6  let $l :=
7    for $i in $auction/site/open_auctions/open_auction/initial
8    where $p/profile/@income > 5000 * exactly-one($i/text())
9    return $i
10 where $p/profile/@income > 50000
11 return <items person="{$p/profile/@income}">{count($l)}</items>
```

Listing B.12: XMark Query 12

```
1  declare variable $graph external;
2
3  let $auction := doc($graph) return
4  for $i in $auction/site/regions/australia/item
5  return <item name="{$i/name/text()}">{$i/description}</item>
```

Listing B.13: XMark Query 13

```
1  declare variable $graph external;
2
3  let $auction := doc($graph) return
4  for $i in $auction/site//item
5  where contains(string(exactly-one($i/description)), "gold")
6  return $i/name/text()
```

Listing B.14: XMark Query 14

```
1
2  let $auction := doc($graph) return
3  for $a in
4    $auction/site/closed_auctions/closed_auction/annotation/description/
         parlist/
5     listitem/
6     parlist/
7     listitem/
8     text/
9     emph/
10    keyword/
11    text()
12 return <text>{$a}</text>
```

Listing B.15: XMark Query 15

```
declare variable $graph external;

let $auction := doc($graph) return
for $a in $auction/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/parlist/listitem/text/
          emph/
       keyword/
       text()
    )
  )
return <person id="{$a/seller/@person}"/>
```

Listing B.16: XMark Query 16

```
declare variable $graph external;

let $auction := doc($graph) return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

Listing B.17: XMark Query 17

```
declare namespace local = "http://www.foobar.org";
declare variable $graph external;

declare function local:convert($v as xs:decimal?) as xs:decimal?
{
  2.20371 * $v (: convert Dfl to Euro :)
};

let $auction := doc($graph) return
for $i in $auction/site/open_auctions/open_auction
return local:convert(zero-or-one($i/reserve))
```

Listing B.18: XMark Query 18

```
declare variable $graph external;

let $auction := doc($graph) return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending empty greatest
return <item name="{$k}">{$b/location/text()}</item>
```

Listing B.19: XMark Query 19

```
declare variable $graph external;

let $auction := doc($graph) return
<result>
  <preferred>
    {count($auction/site/people/person/profile[@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count($auction/site/people/person/profile[@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```

Listing B.20: XMark Query 20

B.2 XSPARQL Benchmark Queries

This section contains the queries of the XMark XQuery benchmark suite [Schmidt et al., 2002] translated to XSPARQL.

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare variable $graph external;

for *
from $graph
where { $person a foaf:Person;
                :id "person0";
                foaf:name $name }
return $name
```

Listing B.21: XSPARQL Query 1

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

for $increase
from $graph
where { $auction a :OpenAuction .
        $bid a :Bid;
             :onAuction $auction;
             :order 1;
             :increase $increase }
return <increase>{$increase}</increase>
```

Listing B.22: XSPARQL Query 2

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

for $increase $increase2
from $graph
where { $auction a :OpenAuction .
        [] a :Bid;
             :onAuction $auction;
             :order 1 ;
             :increase $increase.
        [] a :Bid;
             :onAuction $auction;
             :order $order ;
             :increase $increase2;
        filter($increase * 2 <= $increase2 )
        optional {
          [] a :Bid ; :onAuction $auction ; :order $o2 .
          filter($o2 > $order)
        }
        filter(!bound($o2))
}
return   <increase
  first="{$increase}"
  last="{$increase2}"/>
```

Listing B.23: XSPARQL Query 3

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare variable $graph external;

for $reserve
from $graph
where { $b1 a :Bid ;
            :bidder [ :id "person20" ] ;
            :order $o1 ;
            :onAuction $a .
        $b2 a :Bid ;
            :bidder [ :id "person51" ] ;
            :order $o2 ;
            :onAuction $a .
        filter($o1 < $o2)
        $a  a :OpenAuction ;
            :reserve $reserve .
}
return <history>{$reserve}</history>
```

Listing B.24: XSPARQL Query 4

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

let $x := for $price
   from $graph
   where {
     [ a :ClosedAuction ;
       :price $price ] .
     filter($price >= 40)
   }
   return $price
return count($x)
```

Listing B.25: XSPARQL Query 5

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

count(
for *
from $graph
where {
$item  a :Item ;
       :locatedIn [] .
}
return $item
)
```

Listing B.26: XSPARQL Query 6

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare variable $graph external;

count (
for *
from $graph
where {
 { $x :description [] .}
 union
 { $x foaf:mbox [] . }
}
return $x
)
```

Listing B.27: XSPARQL Query 7

```
1  prefix : <http://xsparql.deri.org/data/>
2  prefix foaf: <http://xmlns.com/foaf/0.1/>
3  declare variable $graph external;
4
5  for $id $name
6  from $graph
7  where {
8  $person a foaf:Person ;
9          :id $id ;
10         foaf:name $name .
11 }
12 return <item person="{$name}">
13   {
14     let $x := for * from $graph
15       where {
16         $ca a :ClosedAuction ;
17             :buyer [ :id $id ] .
18       }
19       return $ca
20     return count($x)
21   }
22 </item>
```

Listing B.28: XSPARQL Query 8

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare variable $graph external;

for $id $name
from $graph
where {
  [] a foaf:Person ;
    foaf:name $name ;
    :id $id.
}
return
  <person name="{$name}">
  {
    for *
    from $graph
    where {
       [] :buyer [:id $id];
          a :ClosedAuction ;
          :itemRef [
            :locatedIn [ a :Region ;
                         :name "europe" ] ;
            :name $itemname
          ] .
    }
    return <item>{$itemname}</item>
  }
  </person>
```

Listing B.29: XSPARQL Query 9

```
1  prefix : <http://xsparql.deri.org/data/>
2  prefix foaf: <http://xmlns.com/foaf/0.1/>
3  declare variable $graph external;
4
5  for distinct $catid
6    from $graph
7    where {
8      $category a :Category ;
9                :id $catid.
10   }
11
12 return
13 <categorie><id>{$catid}</id>
14 {
15   for *
16   from $graph
17   where {
18     $profile a :PersonalProfile ;
19        :ofPerson $person ;
20        :interest [ a :Category ;
21                    :id $catid  ] .
22
23     optional { $profile :gender $gender . }
24     optional { $profile :age $age . }
25     optional { $profile :education $education . }
26     optional { $profile :income $income . }
27
28     optional { $person foaf:name $name . }
29     optional { $person foaf:mbox $email . }
30     optional { $person foaf:homepage $homepage . }
31   }
32   return
33     <personne>
34       <statistiques>
35         <sexe>{$gender}</sexe>
36         <age>{$age}</age>
37         <education>{$education}</education>
38         <revenu>{$income}</revenu>
39       </statistiques>
40       <coordonnees>
41         <nom></nom>
42         <rue></rue>
43         <ville></ville>
44         <pays></pays>
45         <reseau>
46           <courrier>{$email}</courrier>
47           <pagePerso>{$homepage}</pagePerso>
```

```
48          </reseau>
49        </coordonnees>
50        <cartePaiement></cartePaiement>
51      </personne>
52  }
53  </categorie>
```

Listing B.30: XSPARQL Query 10

```
1  prefix : <http://xsparql.deri.org/data/>
2  prefix foaf: <http://xmlns.com/foaf/0.1/>
3  declare variable $graph external;
4
5  for * from $graph
6  where {
7    [] :ofPerson $person ; :income $income .
8    $person a foaf:Person ; foaf:name $name .
9  }
10 let $l :=
11   for * from $graph
12   where {
13     [] a :OpenAuction ; :initialValue $initialValue .
14     filter($income > 5000 * $initialValue)
15   }
16   return $initialValue
17 return <items name="{$name}">{count($l)}</items>
```

Listing B.31: XSPARQL Query 11

```
1
2  prefix : <http://xsparql.deri.org/data/>
3  declare variable $graph external;
4
5  for * from $graph
6  where {
7    [] a :PersonalProfile; :income $income .
8    filter($income > 50000)
9  }
10 let $l :=
11    for * from $graph
12    where {
13       [] a :OpenAuction ; :initialValue $initialValue .
14       filter($income > 5000 * $initialValue)
15    }
16    return $initialValue
17 return <items person="{$income}">{count($l)}</items>
```

Listing B.32: XSPARQL Query 12

```
1  prefix : <http://xsparql.deri.org/data/>
2  declare variable $graph external;
3
4  #let $auction := doc("auction.xml") return
5  #for $i in $auction/site/regions/australia/item
6  #return <item name="{$i/name/text()}">{$i/description}</item>
7
8  for *
9  from $graph
10 where {
11   [] a :Item ;
12      :locatedIn [ a :Region ; :name "australia" ] ;
13      :name $name ;
14      :description $desc .
15 }
16 return <item name="{$name}">{$desc}</item>
```

Listing B.33: XSPARQL Query 13

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

for $name from $graph
where {
  $i a :Item ; :description $d .
  filter regex($d, "gold")
  $i :name $name .
}
return $name
```

Listing B.34: XSPARQL Query 14

```
           because saxon:parse creates XML in the empty namespace :)

prefix t: <http://xsparql.deri.org/data/>
declare namespace saxon="http://saxon.sf.net/";
declare variable $graph external;

for $desc
from $graph
where {
[] t:auction [a t:ClosedAuction]; t:description $desc
}
return
 for $b in saxon:parse(fn:data($desc))
 for $a in $b/text/emph/keyword/text()
 return <text>{$a}</text>
```

Listing B.35: XSPARQL Query 15

```
prefix t: <http://xsparql.deri.org/data/>
declare variable $graph external;

for $desc $id
from $graph
where {
[] t:auction [a t:ClosedAuction; t:seller [t:id $id]]; t:description
    $desc
}
for $b in saxon:parse(fn:data($desc))
for $a in $b/text/emph/keyword/text()[1]
return <person id="{$id}"/>
```

Listing B.36: XSPARQL Query 16

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare variable $graph external;

for $name
from $graph
where {
  $person a foaf:Person ;
          foaf:name $name .
  optional {
    $person foaf:homepage $homepage.
  }
  filter(!bound($homepage)).
}
return <person name="{$name}"/>
```

Listing B.37: XSPARQL Query 17

```
prefix : <http://xsparql.deri.org/data/>

declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal) as xs:decimal
{
  2.20371 * $v (: Convert Dfl to Euro :)
};
declare variable $graph external;
for *
from $graph
where {
  [] a :OpenAuction ; :reserve $reserve .
}
return local:convert($reserve)
```

Listing B.38: XSPARQL Query 18

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

#let $auction := doc("auction.xml") return
#for $b in $auction/site/regions//item
#let $k := $b/name/text()
#order by zero-or-one($b/location) ascending empty greatest
#return <item name="{$k}">{$b/location/text()}</item>

for * from $graph
where {
[] a :Item ;
   :name $name ;
   :location $location .
}
order by $name
return <item name="{$name}">{$location}</item>
```

Listing B.39: XSPARQL Query 19

```
prefix : <http://xsparql.deri.org/data/>
declare variable $graph external;

<result>
  <preferred>
    { let $l := for * from $graph
        where { [] :income $income . filter($income >= 100000) }
        return $income
      return count($l)
    }
  </preferred>
  <standard>
    { let $l := for * from $graph
        where { [] :income $income . filter($income >= 30000 && $income
            < 100000) }
        return $income
      return count($l)
    }
  </standard>
  <challenge>
    { let $l := for * from $graph
        where { [] :income $income . filter($income < 30000) }
        return $income
      return count($l)
    }
  </challenge>
  <na>
    {
      let $l := for * from $graph
        where { $pp a :PersonalProfile . optional{ $pp :income $income .
             } filter(!bound($income)) }
        return $income
      return count($l)
    }
  </na>
</result>
```

Listing B.40: XSPARQL Query 20

B.3 XDEP Optimised Queries

The last set of queries were built by applying our optimisation approach of Chapter 5 to the relevant queries of the last section. The queries here are given as pure XQuery queries to show the rewriting.

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/
       xsparql.xquery" at "http://xsparql.deri.org/demo/xquery/xsparql-
       types.xquery";
2  declare namespace _javaSaxon = "java:org.deri.sparql.Sparql";
3  (: declare default element namespace "http://xsparql.deri.org/data/"; :)
4  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
5  declare variable $graph external;
6
7  let $_aux_results4 := _xsparql:_sparql( fn:concat("PREFIX foaf: <http://
       xmlns.com/foaf/0.1/>
8    PREFIX : <http://xsparql.deri.org/data/>
9    ", " SELECT ", "$ca $id", " ", "from", _xsparql:_rdf_term( _xsparql:
         _binding_term( $graph ) ), " WHERE { ", " { ", "$ca", " ", " a ",
         " ", ":ClosedAuction", " ; ", ":buyer", " ", "[", ":id", " $id ]",
          " . ", " } ", " } ") )
10
11 (: XSPARQL FOR from 8:4 :)
12 let $_aux_results0 := _xsparql:_sparql( fn:concat("PREFIX foaf: <http://
       xmlns.com/foaf/0.1/>
13 PREFIX : <http://xsparql.deri.org/data/>
14 ", " SELECT ", "$id ", "$name ", "from", _xsparql:_rdf_term( _xsparql:
       _binding_term( $graph ) ), " WHERE { ", " { ", "$person", " ", " a "
       , " ", "foaf:Person", " ; ", ":id", " ", "$id", " ; ", "foaf:name",
       " ", "$name", " . ", " } ", " } ") )
15 for $_aux_result0 at $_aux_result0_pos in _xsparql:_sparqlResults(
       $_aux_results0 )
16
17 (: SPARQL variable $id from 8:4 :)
18 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
19
20 (: SPARQL variable $name from 8:8 :)
21 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
22 return
23   <item person="{$name}">
24     {let $x :=
25   for $_aux_result4 at $_aux_result4_pos in _xsparql:_sparqlResults(
         $_aux_results4 )
26   where $id = _xsparql:_resultNode( $_aux_result4, "id" )
27   return
28
29   (: SPARQL variable $ca from 0:0 :)
```

```
30    let $ca := _xsparql:_resultNode( $_aux_result4, "ca" )
31
32    (: dependent variable $id :)
33    return
34      fn:data( $ca )
35    return
36      count( $x )}
37    </item>
```

Listing B.41: Xdep optimised query 8

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/
       xsparql.xquery" at "http://xsparql.deri.org/demo/xquery/xsparql-
       types.xquery";
2  declare namespace _javaSaxon = "java:org.deri.sparql.Sparql";
3  (: declare default element namespace "http://xsparql.deri.org/data/"; :)
4  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
5  declare variable $graph external;
6
7    let $_aux_results4 := _xsparql:_sparql( fn:concat("PREFIX foaf: <http
         ://xmlns.com/foaf/0.1/>
8    PREFIX : <http://xsparql.deri.org/data/>
9    ", " SELECT ", "$itemname $id", " ", "from", _xsparql:_rdf_term(
         _xsparql:_binding_term( $graph ) ), " WHERE { ", " { ", "[]", " ",
          ":buyer", " ", "[", ":id", " $id ]", " ; ", " a ", " ", ":
         ClosedAuction", " ; ", ":itemRef", " ", "[", ":locatedIn", " ", "[
         ", " a ", " ", ":Region", " ; ", ":name", " ", """europe""", "]",
         " ; ", ":name", " ", "$itemname", "]", " . ", " } ", " } ") )
10
11
12 (: XSPARQL FOR from 21:4 :)
13 let $_aux_results0 := _xsparql:_sparql( fn:concat("PREFIX foaf: <http://
       xmlns.com/foaf/0.1/>
14 PREFIX : <http://xsparql.deri.org/data/>
15 ", " SELECT ", "$id ", "$name ", "from", _xsparql:_rdf_term( _xsparql:
       _binding_term( $graph ) ), " WHERE { ", " { ", "[]", " ", " a ", " "
       , "foaf:Person", " ; ", "foaf:name", " ", "$name", " ; ", ":id", " "
       , "$id", " . ", " } ", " } ") )
16 for $_aux_result0 at $_aux_result0_pos in _xsparql:_sparqlResults(
       $_aux_results0 )
17
18 (: SPARQL variable $id from 21:4 :)
19 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
20
21 (: SPARQL variable $name from 21:8 :)
22 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
23 return
24   <person name="{$name}">
```

```
25      {
26    for $_aux_result4 at $_aux_result4_pos in _xsparql:_sparqlResults(
          $_aux_results4 )
27    where $id = _xsparql:_resultNode( $_aux_result4, "id" )
28    return
29
30    (: SPARQL variable $itemname from 0:0 :)
31    let $itemname := _xsparql:_resultNode( $_aux_result4, "itemname" )
32
33    (: dependent variable $id :)
34    return
35      <item>{fn:data( $itemname )}</item>}
36      </person>
```

Listing B.42: XDEP optimised query 9

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/
       xsparql.xquery" at "http://xsparql.deri.org/demo/xquery/xsparql-
       types.xquery";
2  declare namespace _javaSaxon = "java:org.deri.sparql.Sparql";
3  (: declare default element namespace "http://xsparql.deri.org/data/"; :)
4  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
5  declare variable $graph external;
6
7    let $_aux_results4 := _xsparql:_sparql( fn:concat("PREFIX foaf: <http
         ://xmlns.com/foaf/0.1/>
8    PREFIX : <http://xsparql.deri.org/data/>
9    ", " SELECT ", "$name ", "$income ", "$gender ", "$email ", " ", "
         $homepage ", "$age ", "$person ", "$profile ", "$education $catid
         ", "from", _xsparql:_rdf_term( _xsparql:_binding_term( $graph ) ),
          " WHERE { ", " { ", "$profile", " ", " a ", " ", ":
         PersonalProfile", " ; ", ":ofPerson", " ", "$person", " ; ", ":
         interest", " ", "[", " a ", " ", ":Category", " ; ", ":id", "
         $catid ]", " . ", " optional ", " { ", "$profile", " ", ":gender",
          " ", "$gender", " . ", " } ", " optional ", " { ", "$profile", "
         ", ":age", " ", "$age", " . ", " } ", " optional ", " { ", "
         $profile", " ", ":education", " ", "$education", " . ", " } ", "
         optional ", " { ", "$profile", " ", ":income", " ", "$income", " .
          ", " } ", " optional ", " { ", "$person", " ", "foaf:name", " ",
         "$name", " . ", " } ", " optional ", " { ", "$person", " ", "foaf:
         mbox", " ", "$email", " . ", " } ", " optional ", " { ", "$person"
         , " ", "foaf:homepage", " ", "$homepage", " . ", " } ", " } ", " }
          ") )
10
11 (: XSPARQL FOR from 8:4 :)
12 let $_aux_results0 := _xsparql:_sparql( fn:concat("PREFIX foaf: <http://
       xmlns.com/foaf/0.1/>
13 PREFIX : <http://xsparql.deri.org/data/>
```

```
14 ", " SELECT ", " DISTINCT ", "$catid ", "from", _xsparql:_rdf_term(
       _xsparql:_binding_term( $graph ) ), " WHERE { ", " { ", "$category",
        " ", " a ", " ", ":Category", " ; ", ":id", " ", "$catid", " . ", "
        } ", " } ") )
15 for $_aux_result0 at $_aux_result0_pos in _xsparql:_sparqlResults(
       $_aux_results0 )
16
17 (: SPARQL variable $catid from 8:13 :)
18 let $catid := _xsparql:_resultNode( $_aux_result0, "catid" )
19 return
20   <categorie><id>{fn:data( $catid )}</id>
21   {
22   (: XSPARQL FOR from 0:0 :)
23   for $_aux_result4 at $_aux_result4_pos in _xsparql:_sparqlResults(
         $_aux_results4 )
24   where $catid = _xsparql:_resultNode( $_aux_result4, "catid" )
25   return
26
27   (: SPARQL variable $name from 0:0 :)
28   let $name := _xsparql:_resultNode( $_aux_result4, "name" )
29
30   (: SPARQL variable $income from 0:0 :)
31   let $income := _xsparql:_resultNode( $_aux_result4, "income" )
32
33   (: SPARQL variable $gender from 0:0 :)
34   let $gender := _xsparql:_resultNode( $_aux_result4, "gender" )
35
36   (: SPARQL variable $email from 0:0 :)
37   let $email := _xsparql:_resultNode( $_aux_result4, "email" )
38
39   (: dependent variable $catid :)
40
41   (: SPARQL variable $homepage from 0:0 :)
42   let $homepage := _xsparql:_resultNode( $_aux_result4, "homepage" )
43
44   (: SPARQL variable $age from 0:0 :)
45   let $age := _xsparql:_resultNode( $_aux_result4, "age" )
46
47   (: SPARQL variable $person from 0:0 :)
48   let $person := _xsparql:_resultNode( $_aux_result4, "person" )
49
50   (: SPARQL variable $profile from 0:0 :)
51   let $profile := _xsparql:_resultNode( $_aux_result4, "profile" )
52
53   (: SPARQL variable $education from 0:0 :)
54   let $education := _xsparql:_resultNode( $_aux_result4, "education" )
55   return
```

```
56      <personne>
57            <statistiques>
58              <sexe>{fn:data( $gender )}</sexe>
59              <age>{fn:data( $age )}</age>
60              <education>{fn:data( $education )}</education>
61              <revenu>{fn:data( $income )}</revenu>
62            </statistiques>
63            <coordonnees>
64              <nom></nom>
65              <rue></rue>
66              <ville></ville>
67              <pays></pays>
68              <reseau>
69                <courrier>{fn:data( $email )}</courrier>
70                <pagePerso>{fn:data( $homepage )}</pagePerso>
71              </reseau>
72            </coordonnees>
73            <cartePaiement></cartePaiement>
74          </personne>}
75    </categorie>
```

Listing B.43: XDEP optimised query 10

Appendix C

Benchmark Results

This appendix presents the results of the evaluation of Chapter 6. We show the mean query evaluation times of the three experiments of Chapter 6 in seconds rounded to one decimal. Whenever there is no result given (experiment #2, queries #8–#12, for datasets #4 and #5) the query evaluation time exceeded the timeout value of our test framework, which was fixed to 10 hours, i. e., 36 000 seconds.

Table C.1 shows the query evaluation times for the first experiment, that is the evaluation of the 20 XMark XQuery queries.

Table C.2 shows the query evaluation times for the second experiment, that is the evaluation of the standard XSPARQL rewriting of the 20 translated XSPARQL queries.

Table C.3 contains the benchmark results of the Xdep optimised XSPARQL queries. Since only three of the available 20 queries are compatible with Xdep (queries #8–#10) we provide query evaluation results for these queries only.

Query #	Dataset #1	Dataset #2	Dataset #3	Dataset #4	Dataset #5
1	1.8	2.0	2.4	4.6	7.9
2	1.7	2.0	2.4	4.6	8.0
3	1.9	2.1	2.6	4.8	8.1
4	1.8	2.1	2.6	4.8	8.2
5	1.7	1.9	2.4	4.6	7.8
6	1.8	2.0	2.3	4.5	7.7
7	1.8	2.0	2.4	4.6	7.9
8	2.0	2.2	2.8	5.1	9.2
9	2.0	2.3	2.9	5.8	9.8
10	2.5	3.2	3.9	6.4	11.1
11	2.7	4.1	11.0	54.5	198.2
12	2.5	3.2	6.1	22.7	75.0
13	1.8	2.0	2.5	4.7	8.0
14	1.8	2.0	2.5	4.8	8.5
15	1.7	2.0	2.4	4.5	7.7
16	1.8	2.0	2.4	4.5	7.7
17	1.8	2.1	2.5	4.7	8.1
18	1.8	2.1	2.5	4.7	8.3
19	2.0	2.4	3.0	5.6	9.2
20	1.8	2.1	2.6	4.9	8.5

Table C.1: Benchmark results of experiment #1

Query #	Dataset #1	Dataset #2	Dataset #3	Dataset #4	Dataset #5
1	2.6	3.6	7.7	16.8	42.4
2	11.9	45.2	175.6	1 043.5	3 651.2
3	12.1	45.5	179.9	1 022.9	3 679.7
4	2.6	3.6	6.7	18.6	40.7
5	2.3	3.5	6.2	16.5	34.1
6	2.9	4.6	8.1	21.1	37.6
7	4.1	6.2	9.7	22.2	42.5
8	1 478.7	6 225.3	25 904.6	—	—
9	1 459.3	6 133.2	25 659.0	—	—
10	85.0	336.1	1 455.2	—	—
11	641.5	2 527.1	10 892.0	—	—
12	314.5	1 161.9	4 895.3	—	—
13	2.6	4.1	7.1	20.2	36.4
14	2.4	3.8	6.5	16.5	34.6
15	2.9	4.5	8.2	20.9	40.4
16	2.9	4.8	8.6	21.7	39.7
17	2.7	4.4	7.8	18.8	36.3
18	2.6	4.0	7.0	17.6	36.6
19	3.5	5.8	9.6	21.6	39.5
20	6.3	11.7	23.6	64.1	141.4

Table C.2: Benchmark results of experiment #2

Query #	Dataset #1	Dataset #2	Dataset #3	Dataset #4	Dataset #5
8	20.2	55.5	189.1	1 114.4	4 501.8
9	12.3	26.7	74.2	383.3	1 433.8
10	36.2	107.7	400.4	2 330.9	9 414.9

Table C.3: Benchmark results of experiment #3

Appendix D

Semantic Properties

In this appendix we will give several propositions and corresponding proofs or proof sketches describing semantic properties of XSPARQL++ as well as the correspondence of the semantics of the XDEP optimisation with the original XSPARQL++ semantics.

D.1 Semantic Properties of XSPARQL++

XSPARQL subsumes XQuery syntax by extending XQuery with SPARQL grammar productions. The semantics of XSPARQL is defined on top of XQuery by the introduction of new normalisation rules as well as rules for static typing and dynamic evaluation. But evaluating the *SparqlForClause* still relies on the SPARQL semantics. Since only these new rules where introduced, semantics is unchanged for standard XQuery queries.

Proposition D.1. *Let XQ be an XQuery query, then the evaluation result of XQ under XQuery semantics is equivalent to the evaluation result of XQ under XSPARQL++ semantics.*

Proof (Sketch). XSPARQL semantics is defined on top of XQuery semantics. The rules introduced for XSPARQL fall in one of two classes: (1) Rules handling constructs *not* contained in XQuery (such as *SparqlForClause* or *ConstructClause*) (2) Rules redefining XQuery syntax.

First case: Queries containing non-XQuery syntax are out of the scope of the XQuery semantics and are therefore irrelevant for this proof.

Second Case: The only rules falling in this class are the normalisation rules for FLWOR expressions, more specifically *ForClause*s and *LetClause*s. The normalisation rules *LetClause*s are copied from original XQuery (see Rule 2.25). After splitting *ForClause*s to *SimpleForClause*s, exactly as original XQuery, the ones without positional variables are always decorated with

positional variables (see Rule 2.24). Therefore this rule ensures that every *ForClause* contains a positional variable. But since the new positional variable name contains an underscore as first character it is ensured that it is indeed free. Because it is free, original semantics is not changed. □

Corollary D.1. *XSPARQL++ is a conservative extension of XQuery.*

Proof. From Proposition D.1 we know that any XQuery query is also a valid XSPARQL query. Therefore XSPARQL++ is an extension of XQuery conserving semantics. □

The proof to show that XSPARQL++ is a conservative extension of SPARQL is very similar to the one given in [Akhtar et al., 2007].

Proposition D.2. *Let* $SQ = (\$x_1, \ldots, \$x_n, DS, P, M)$ *be a SPARQL query of the form* `select` $\$x_1 \cdots \$x_n$ ***DWM***, *where DS denotes the RDF dataset corresponding to the DatasetClause* ***D***, *G the default graph of DS, P the graph pattern of the WhereClause* ***W***, *and* ***M*** *the solution modifiers. If* $eval(DS(G), SQ) = \Omega_1$ *and*

$$dynEnv \vdash \texttt{for}\ \$x_1 \cdots \$x_n\ \texttt{from}\ D(G)\ \texttt{where}\ \{P\}M\ \texttt{return}\ (\$x_1, \ldots, \$x_n) \Rightarrow \Omega_2.$$

Then, $\Omega_1 \equiv \Omega_2$ *modulo representation*[1].

Proof (Sketch). In Rule 3.6 the *SparqlForClause* of an XSPARQL query is evaluated by the function *fs:sparql*. Since all the variables in P are free (since the query is not nested and no XSPARQL prolog is given, there exists no possibility for a variable to be bound) they are not replaced. The function *fs:sparql*, by definition, evaluates a SPARQL query given as a list of variables, a *GroupGraphPattern* and the *SolutionModifier*, while the dataset of the query is retrieved from the *activeDataset* environment component. Therefore, and because the *ReturnClause* extracts Ω_2, the result is only a representational variant of Ω_1. □

Corollary D.2. *Let SQ be a SPARQL* `construct` *query, then the evaluation result of SQ under SPARQL semantics is equivalent to the evaluation result of SQ under XSPARQL++ semantics.*

Proof (Sketch). A `construct` queries is rewritten to a *SparqlForClause* with a *ConstructClause* as *ReturnClause*. Equivalent evaluation of the *SparqlForClause* is shown in the proof of Proposition D.2. And since XSPARQL's *ConstructClause* semantics reproduce the SPARQL semantics exactly in this regard, `construct` queries are evaluated equivalently to SPARQL semantics. □

[1] meaning that Ω_1 as well as Ω_2 represent the same sequence of (partial) variable bindings while ordering may be relevant

Corollary D.3. *The XSPARQL++ semantics is a conservative extension of the semantics of the SPARQL SelectQuery and the SPARQL ConstructQuery.*

Proof. Since Proposition D.2 and Corollary D.2 show that a SPARQL `select` query and a SPARQL `construct` query is evaluated equivalently under both, the XSPARQL and the SPARQL semantics, it is obvious that XSPARQL++ extends SPARQL while conserving semantics. □

D.2 Correspondence between XSPARQL++ and XDEP

Since XDEP is defined in two different variants, the proof is split in two parts, each handling one of the variants.

Proposition D.3. *Given an XSPARQL query XSQFS containing a SparqlForClause embedded in the ReturnClause of a ForClause, then,* $[\![XSQFS]\!]_{ExprXDEP} = [\![XSQFS]\!]_{Expr'}$.

Proof (Sketch). The rewriting for the outer *ForClause* and for the inner *ReturnClause* is the same for XSPARQL++ and for XDEP.

The first difference (step 1) between XSPARQL++ and XDEP is the inner *SparqlForClause* being evaluated before the outer *ForClause*. Since the result of this *SparqlForClause* is assigned to a new, i. e., previously unbound, variable, no existing environment component relevant is changed in a way that would influence the evaluation result. Therefore this change retains semantics.

The second difference (step 2) is the dependent join implementation. Instead of replacing the dependent variable in the *SparqlForClause* with its value, SPARQL solution mappings are gathered by joining the dependent variable with the solution mappings stored in the variable of the first step. Since the join compares the string value of the dependent variable and additionally considers blank nodes, the same solution mappings are gathered. Eventually the values of the non-dependent variables are assigned to XQuery variables.

Since this results in the same environment components for the *ReturnClause*, the semantics of XSPARQL++ is retained. □

Proposition D.4. *Given an XSPARQL query XSQSS containing a SparqlForClause embedded in the ReturnClause of another SparqlForClause, then,* $[\![XSQSS]\!]_{ExprXDEP} = [\![XSQSS]\!]_{Expr'}$.

Proof (Sketch). The rewriting for the outer *SparqlForClause* and the inner *ReturnClause* is the same for XSPARQL++ and for XDEP.

Since the results of the inner *SparqlForClause* are assigned to a new, i. e., previously unbound, variable, no existing environment is changed in a way

that would influence the evaluation result. Therefore this change retains semantics.

Proposition D.3 shows that the environment components are the same under XSPARQL++ semantics and Xdep when using a single dependent variable. In case of an outer *SparqlForClause* there can exist more than one dependent variables. Since the *fs:join* function selects only solution mappings for which all the dependent variables are the same (*the same* in this context includes the case where the dependent variable was bound to a blank node in the outer *SparqlForClause*), the XQuery environment components are again the same. □

Proposition D.5. *Given an XSPARQL query XSQ then,* $[\![XSQ]\!]_{ExprXDEP} = [\![XSQ]\!]_{Expr'}$.

Proof. Since the semantics of XSPARQL++ is retained by Xdep in the two subcases, the XQuery *ForClause* (see Proposition D.3) and the *SparqlForClause* (see Proposition D.4), and because these are the only cases for which Xdep is defined, Xdep alone retains the XSPARQL++ semantics. □

Bibliography

Every cited publication contains a reference the chapter(s) it was cited at.

Serge Abiteboul. Querying Semi-Structured Data. In Foto Afrati and Phokion Kolaitis, editors, *Database Theory - ICDT '97*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin, January 1997. doi: 10.1007/3-540-62222-5_33. 1.1

Ben Adida and Mark Birbeck. RDFa Primer. Bridging the Human and Data Webs. W3C Working Group Node, W3C, October 2008. URL `http://www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014/`. 2.2.1

Loredana Afanasiev and Maarten Marx. An analysis of XQuery benchmarks. *Information Systems*, 33(2):155–181, April 2008. ISSN 0306-4379. doi: 10.1016/j.is.2007.05.002. URL `http://www.sciencedirect.com/science/article/B6V0G-4NT9GJR-1/2/e60ed0e8bc466c687c09712900111a6d`. 6.1, 6.2

Waseem Akhtar, Jacek Kopecký, Thomas Krennwallner, and Axel Polleres. XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. Technical Report, DERI, IDA Business Park, Lower Dangan, Galway, Ireland, December 2007. URL `http://www.deri.ie/fileadmin/documents/DERI-TR-2007-12-14.pdf`. 1.3, 2.5.2, D.1

Waseem Akhtar, Jacek Kopecký, Thomas Krennwallner, and Axel Polleres. XSPARQL: Traveling between the XML and RDF worlds - and Avoiding the XSLT pilgrimage. In *5th European Semantic Web Conference (ESWC2008)*, volume 5021, pages 432–447, June 2008. doi: 10.1007/978-3-540-68234-9. URL `http://www.springerlink.com/content/qx68n5h481364805/`. A.2.1

Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. In Amit Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2008*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin / Heidelberg, 2008. doi: 10.1007/978-3-540-88564-1_8. 5.4

Steve Battle. Gloze: XML to RDF and back again. In *Proceedings of the First Jena User Conference*, 2006. doi: 10.1.1.88.8929. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.8929. 1.5

David Beckett. RDF/XML Syntax Specification (Revised). W3C Recommendation, W3C, February 2004. URL http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. 2.2.1, 2.2.1

David Beckett and Tim Berners-Lee. Turtle – Terse RDF Triple Languages. W3C Team Submission, W3C, January 2008. URL http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/. 2.2.1, A.5

David Beckett and Jeen Broekstra. SPARQL Query Results XML Format. W3C Recommendation, W3C, January 2008. URL http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/. 2.4.3, 2.5.2.1, 3.3, 4.2, 4.3.1, 4.3.2.1, 6.2

Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C Recommendation, W3C, January 2007. URL http://www.w3.org/TR/2007/REC-xpath20-20070123/. 1.1

T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, Internet Engineering Task Force, August 1998. URL http://www.rfc-editor.org/rfc/rfc2396.txt. 1.1

Tim Berners-Lee. Notation 3. Design Issue, W3C, 1998. URL http://www.w3.org/DesignIssues/Notation3. 2.2.1

Tim Berners-Lee. Linked Data. Design Issue, W3C, 2006. URL http://www.w3.org/DesignIssues/LinkedData.html. 1.1

Nikos Bikakis, Nektarios Gioldasis, Chrisa Tsinaraki, and Stavros Christodoulakis. Querying XML Data with SPARQL. In Sourav S. Bhowmick, Josef Küng, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 372–381. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-03573-9_32. 1.5

Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal On Semantic Web and Information Systems*, 5(2):1–24, 2009. ISSN 1552-6283. URL http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/. 7.1

Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, W3C, January 2007. URL http://www.w3.org/TR/2007/REC-xquery-20070123/. (document), 1.2, 1.3, 12, 2.3, 2.5.1.2, 3.6.6, A.2, A.2.2

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, W3C, November 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/. (document), 1, 1.1, 8, A.1, A.3

Peter Buneman. Semistructured Data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database systems*, pages 117–121, 1997. ISBN 0-89791-910-6. doi: http://doi.acm.org/10.1145/263661.263675. URL http://portal.acm.org/citation.cfm?id=263661.263675. 1.1

Jeremy J. Carroll and Patrick Stickler. TriX: RDF Triples in XML. Technical Report HPL-2004-56, HP Labs, 2004. URL http://www.hpl.hp.com/techreports/2004/HPL-2004-56.html. 1.5

John Cowan and Richard Tobin. XML Information Set. W3C Recommendation, W3C, February 2004. URL http://www.w3.org/TR/2004/REC-xml-infoset-20040204. 1.1

DBLP. DBLP Computer Science Bibliography. http://dblp.uni-trier.de/. URL http://dblp.uni-trier.de/. Online, accessed 7 July 2010. 3.1

D. V. Davy Van Deursen, Chris Poppe, Gäetan Martens, Erik Mannens, and Rik Van de Walle. XML to RDF Conversion: A Generic Approach. In *Proceedings of the 2008 International Conference on Automated solutions for Cross Media Content and Multi-channel Distribution*, pages 138–144, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3406-0. 1.5

Berrueta Diego, Labra Jose E., and Herman Ivan. XSLT+SPARQL : Scripting the Semantic Web with SPARQL embedded into XSLT stylesheets. In Chris Bizer, Sören Auer, Gunnar Aastrand Grimmes, and Tom Heath, editors, *4th Workshop on Scripting for the Semantic Web*, Tenerife, June 2008. 1.5

Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation, W3C, January 2007. URL http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/. (document), 1.2, 2.2, 2.3.3, 2.3.3.2, 2.24, 3.3, 3.3

Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf, Hannes Staffler, and Stefan Zugal. Translating XPath Queries into SPARQL Queries. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, volume 4805 of *Lecture Notes in Computer Science*, pages 9–10. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-76888-3_5. 1.5

M. Duerst and M. Signard. Internationalized Resource Identifiers (IRIs). RFC 3987, Internet Engineering Task Force, January 2005. URL `http://www.rfc-editor.org/rfc/rfc3987.txt`. `http://www.rfc-editor.org/rfc/rfc3987.txt`. 1.1

Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XSLT 2.0 and XQuery 1.0 Serialization. W3C Recommendation, W3C, January 2007. URL `http://www.w3.org/TR/2007/REC-xslt-xquery-serialization-20070123/`. 1.2

Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation, W3C, January 2007. URL `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/`. 1.1, 1.2, 3.3

FOAF. The Friend of a Friend (FOAF) project. `http://www.foaf-project.org/`. URL `http://www.foaf-project.org/`. Online, accessed 6 July 2010. 2.2

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall International, second edition, June 2008. ISBN 9780131873254. 5.1

Sven Groppe, Jinghua Groppe, Volker Linnemann, Dirk Kukulenz, Nils Hoeller, and Christoph Reinke. Embedding SPARQL into XQuery/XSLT. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 2271–2278. ACM, 2008. ISBN 978-1-59593-753-7. doi: 10.1145/1363686.1364228. 1.5

Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Working Draft, W3C, June 2010. URL `http://www.w3.org/TR/2010/WD-sparql11-query-20100601/`. 1.2, 1

JFlex. JFlex – The Fast Scanner Generator for Java. `http://www.jflex.de/`. URL `http://www.jflex.de/`. Online, accessed 7 July 2010. 4.2.1.1

Joseki. Joseki – A SPARQL Server for Jena. `http://www.joseki.org/`. URL `http://www.joseki.org/`. Online, accessed 7 July 2010. 4.2

Michael Kay. The SAXON XSLT and XQuery Processor. http://saxon.sourceforge.net/. URL http://saxon.sourceforge.net/. Online, accessed 4 October 2010. 4.2

Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C Recommendation, W3C, January 2007. URL http://www.w3.org/TR/2007/REC-xslt20-20070123/. 1.1, 1.2, 1.3

Alfons Kemper and Andre Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg, 2004. ISBN 978-3486576900. 5.1

Graham Klyne and Jeremy Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, W3C, February 2004. URL http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. 1

Thomas Krennwallner, Nuno Lopes, and Axel Polleres. XSPARQL: Semantics. W3C Member Submission, W3C, January 2009. URL http://www.w3.org/Submission/2009/SUBM-xsparql-semantics-20090120/. 2.5, 5.3.2

Andrew Layman, Dave Hollander, and Tim Bray. Namespaces in XML. W3C Recommendation, W3C, January 1999. URL http://www.w3.org/TR/1999/REC-xml-names-19990114. 1.1, A.4

Wolfgang Lehner and Harald Schöning. *XQuery. Grundlagen und fortgeschrittene Methoden.* Dpunkt Verlag, 2004. URL http://ebooks.ulb.tu-darmstadt.de/180/. 5.2

Linked Data. Linked Data – Connect Distributed Data across the Web. http://linkeddata.org/. URL http://linkeddata.org/. Online, accessed 16 September 2010. 1.1

Nuno Lopes, Thomas Krennwallner, Axel Polleres, Waseem Akhtar, and Stéphane Corlosquet. XSPARQL: Implementation and Test-cases. W3C Member Submission, W3C, January 2009. URL http://www.w3.org/Submission/2009/SUBM-xsparql-implementation-20090120/. 2.5, 4

Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, W3C, January 2007. URL http://www.w3.org/TR/2007/REC-xpath-functions-20070123/. 1.2, 2.5.1.2

Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, W3C, February 2004. URL http://www.w3.org/TR/rdf-primer/. http://www.w3.org/TR/rdf-primer/. 1.1

Mauro San Martín and Claudio Gutierrez. Representing, Querying and Transforming Social Networks with RDF/SPARQL. In *The Semantic Web: Research and Applications*, volume 5554 of *Lecture Notes in Computer Science*, pages 293–307. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-02121-3_24. 3.1

Jim Melton and Subramanian Muralidhar. XML Syntax for XQuery 1.0 (XQueryX). W3C Recommendation, W3C, January 2007. URL `http://www.w3.org/TR/2007/REC-xqueryx-20070123`. 1.2

Nilo Mitra and Yves Lafon. SOAP Version 1.2 Part 0:Primer. W3C Recommendation, W3C, jun 2007. URL `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`. 1.1

Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, Illustrated edition, May 2007. ISBN 0978739256. URL `http://www.pragprog.com/titles/tpantlr/`. 4.2.1.1, 4.3.1

Alexandre Passant, Jacek Kopecký, Stéphane Corlosquet, Diego Berrueta, Davide Palmisano, and Axel Polleres. XSPARQL: Use cases. W3C Member Submission, W3C, January 2009. URL `http://www.w3.org/Submission/2009/SUBM-xsparql-use-cases-20090120/`. 2.5, 3.2.1

Steve Pemberton. XHTML™ 1.0 The Extensible HyperText Markup Language. W3C Recommendation, W3C, August 2002. URL `http://www.w3.org/TR/2002/REC-xhtml1-20020801/`. 1.1

Axel Polleres, Thomas Krennwallner, Nuno Lopes, Jacek Kopecký, and Stefan Decker. XSPARQL Language Specification. W3C Member Submission, W3C, January 2009. URL `http://www.w3.org/Submission/2009/SUBM-xsparql-language-specification-20090120/`. (document), 1, 1.1, 2.5, 4, 2.5.1, 4.4.2

Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, W3C, January 2008. URL `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`. 1.2, 2.2.1, 2.9, 3.5.3, 3.7, A.2, A.2.3

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *5th International Semantic Web Conference*. ISWC, 2006. URL `http://www.dcc.uchile.cl/~cgutierr/papers/sparql.pdf`. 2.1, 2.4.1.1, 2.2, 2.3, 2.4.2, 2.4, 2.5, 2.6, 2.7, 2.8

Simon Schenk, Paul Gearon, and Alexandre Passant. SPARQL 1.1 Update. W3C Working Draft, W3C, October 2010. URL http://www.w3.org/TR/2010/WD-sparql11-update-20101014/. 2

Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002. 6.1, B, B.2

Dhavalkumar Thakker, Taha Osman, Shakti Gohil, and Phil Lakin. A Pragmatic Approach to Semantic Repositories Benchmarking. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *The Semantic Web: Research and Applications*, volume 6088 of *Lecture Notes in Computer Science*, pages 379–393. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-13486-9_26. 7.1

Priscilla Walmsley. *XQuery*. O'Reilly Media, 2007. 5.2

Priscilla Walmsley and David C. Fallside. XML Schema Part 0: Primer Second Edition. W3C Recommendation, W3C, October 2004. URL http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/. 1.1

Norman Walsh. RDF Twig: Accessing RDF Graphs in XSLT, 2003. URL http://rdftwig.sourceforge.net/. Presented at Extreme Markup Languages (XML) 2003, Montreal, Canada. 1.5