GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

ISSN 1612-6793

Master's Thesis

submitted in partial fulfilment of the
requirements for the course "Applied Computer Science"

Nested CONSTRUCTs in SPARQL

Stefan Siemer

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

25. October 2019

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

First Supervisor:	Prof. Dr. Wolfgang May
Second Supervisor:	Prof. Dr. Carsten Damm

[illegible]

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 25. October 2019

Abstract

Nested CONSTRUCT queries provide the possibility of generating and shaping RDF data in the FROM clause of a SPARQL query. This feature can be used to create ground new triples, reuse and optimise queries or perform distributed evaluation and data integration. In this thesis, a modified version of the SPARQL language is considered which enables the use of nested CONSTRUCT queries. Furthermore, the corresponding implementation is presented for the Jena Framework. Afterwards, the Mondial database is used to show several use cases. In the end, the Win-Move-Game demonstrates the relevance of nested CONSTRUCT queries when used in combination with reasoning. All in all, it turns out that nested CONSTRUCT queries are a very useful feature in the SPARQL language.

Contents

List of Figures

List of Listings

List of Abbreviations

Chapter 1

Introduction

The SPARQL Protocol And RDF Query Language (SPARQL) is the standard query language for graph Knowledge Bases (KBs) based on the Resource Description Framework (RDF) [3]. RDF KBs are commonly used in Linked Open Data (LOD) [4] to provide publicly accessible data endpoints [5]. Many theoretical and practical approaches to further improve SPARQL are proposed in the literature. The paper [2] by Angles and Gutierrez introduces the possibility of several types of sub-queries. A sub-query is a fully-fledged query that is nested in one part of another query [2]. Similar features have various applications in other query languages, such as "views" in the Structured Query Language (SQL) [6]. Their advantages are supposed to be manifold. On the one hand, the handling of queries becomes much easier by introducing new possibilities like views and the reuse of existing queries. On the other hand, the features introduce new possibilities of rewriting queries and optimising the evaluation process on the technical level [2].

Roughly speaking, a SPARQL query is built with the three main components SELECT-FROM-WHERE. First of all, one of the query formats SELECT, DESCRIBE, ASK or CONSTRUCT determines the shape of the answer respectively as variable bindings, RDF, boolean or RDF [7]. After that, the input data is defined by FROM clauses. The graph pattern, which will be used in the evaluation process, is specified in the WHERE clause. Since SPARQL 1.1, nested SELECT queries are possible in the WHERE clause [7]. The FROM clause does not yet accept nested queries, even though similar languages like SQL contain this feature [2]. A nested query in the FROM clause of SPARQL should produce fresh RDF data and add the transformed data to the input. This functionality can be provided by the CONSTRUCT and the DESCRIBE query type. Those queries in the FROM clause will be referred to as "nested CONSTRUCTs".

A possible use case for nested CONSTRUCT queries is the creation of not explicitly known, new information. This information is then added to the current dataset for further processing. As a motivational example found in the literature [2], Listing 1.1 shows the construction of the new relationship "co-author" in a book-author KB. This new relation is afterwards used to gather information about all author pairs that have been co-authors in the past [2]. In comparison to that, Listing 1.2 represents the analogous query in SQL. The nested query sections are coloured green.

```sparql
SELECT ?Mail1 ?Mail2
FROM <file:mails.n3>
FROM {
        CONSTRUCT { ?Aut1 co-author ?Aut2}
        FROM <file:bib.n3>
        WHERE { ?Art bib:has-author ?Aut1.
                ?Art bib:has-author ?Aut2.
                FILTER ( !( ?Aut1 = ?Aut2))
              }
     }
WHERE { ?Per1 co-author ?Per2.
        ?Per1 foaf:mbox ?Mail1.
        ?Per1 foaf:mbox ?Mail2
      }
```

Listing 1.1: Motivational example in SPARQL [2].

```sql
SELECT  mbox1.Mail as Mail1,
        mbox2.Mail as Mail2
FROM mbox mbox1,
     mbox mbox2,
(
    SELECT  ha1.Aut as Aut1,
            ha2.Aut as Aut2
    FROM    has_author ha1,
            has_author ha2
    WHERE   ha1.Art=ha2.Art
        AND ha1.Aut!=ha2.Aut
) as co_author
WHERE   mbox1.Per=co_author.Aut1
        AND mbox2.Per=co_author.Aut2;
```

Listing 1.2: Motivational example in SQL.

In this thesis, an implementation of the extended SPARQL, supporting nested CONSTRUCTs, is presented. Firstly, some theoretical foundations of RDF, SPARQL and the Web Ontology Language (OWL) are described in Chapter 2. Furthermore, all used tools, frameworks and data are sketched and briefly explained. The following Chapters 3 and 4 are concerned with the new grammar and the integration of the desired changes in the current tools. Chapter 5 presents the new possible applications of nested CONSTRUCTs. The freshly added features will be used for query composition, modularisation, data integration and optimisation [2]. Further, Chapter 6 shows a comparison between pure RDF and RDF together with constructed RDF when used in combination with a reasoner. Finally, Chapter 7 summarises and draws a conclusion with possibilities for further work.

Chapter 2

Basics

This chapter provides a basic formal description of the theoretical foundations concerning RDF and SPARQL. Next to the official notation and semantics published by the World Wide Web Consortium (W3C) there will be accompanying examples. These examples shall give a brief impression on how the actual querying process works. Furthermore, a rough overview of the practical testing environment is presented.

2.1 Resource Description Framework

The following definitions and concepts of RDF are mainly derived from the official W3C recommendations on RDF 1.1 [8]. RDF formally describes a directed graph. The edge from one node to another specifies the binary relation between them [9].

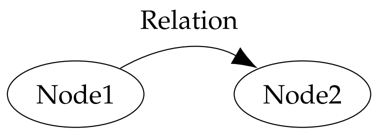

Figure 2.1: Graph representation of a binary relation.

Definition 2.1.1 (Internationalized Resource Identifier (IRI)). *Is a string uniquely identifying a resource. The set of all IRIs is denoted with I. Two IRIs $a, b \in I$ are equal, if and only if they are of the same size and for every i-th character in a and b holds $a_i = b_i$ [8].*

This property guarantees a flawless identification within the set of all IRIs. In a local data source different common words would be sufficient to describe resources. In the internet a prefix is needed to make the resource unique again, because two people could have used the same words for their resources. These prefixes must be a unique string, like a domain name.

Definition 2.1.2 (Literal). *Describes values like strings, numbers and dates. A literal commonly consists of a lexical value and an optional IRI for the data type. The IRI defines the data type of value, hence describing how the lexical value is to be interpreted. The set of all Literals is denoted L. Two literals are equal, if and only if the value and the data type IRI are the same [8].*

Usually built-in IRIs exist for the regular data types like integers and strings. There is also a broad selection of other official data types recommended by the W3C. If everyone uses these recommendations there will be no problem with correctly interpreting the values of other data sources.

Definition 2.1.3 (Blank Nodes (BNs)). *Are elements of an arbitrarily generated set, that is disjunct to I and L. The set of all BNs is denoted with B [8].*

BNs are usually created only locally as a temporal identifier.

Definition 2.1.4 (RDF Term). *Is an element of the set* $T := I \cup L \cup B$ *[8].*

Definition 2.1.5 (RDF Triple). *Is a triple* $(s, p, o) \in (I \cup B) \times I \times T$ *expressing the relation of the Subject* s *and Object* o *with the Predicate* p *[8].*

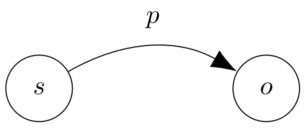

Figure 2.2: Graph representation of a RDF triple.

Definition 2.1.6 (RDF Graph). *G denotes a finite set of RDF triples. The nodes of G are the set of subjects and objects occurring in the triples. The predicates are the edges of G. Predicates can also occur as a node to allow statements about the predicate itself [8].*

The following sections consider the fictional RDF graphs G_1 and G_2 about integer properties as small examples. They both share the prefix db that gets expanded to $< intdb : \sharp >$. The exemplary resource $db : 1$ has the properties name "One" and the value 1.

$$\begin{aligned}
G_1 := \; & \{(db:1,\; db:name,\; "One"),\; (db:1,\; db:val,\; 1), \\
& (db:2,\; db:name,\; "Two"),\; (db:2,\; db:val,\; 2), \\
& (db:3,\; db:name,\; "Three"),\; (db:3,\; db:val,\; 3), \ldots\} \\
G_2 := \; & \{(db:2,\; db:has-property,\; db:isprime), \\
& (db:3,\; db:has-property,\; db:isprime), \ldots\}
\end{aligned}$$

The Notation 3 (N3) [10] is a common form of representing RDF graphs. This format uses white-spaces to delimit the single triple items and points for separating triples. Commas are used for shortening the same subject and predicate with different objects. Finally, semicolons are used to shortcut different predicates for the same subject. The Listings 2.1 and 2.2 show the graphs G_1 and G_2 in N3 respectively.

```
@prefix db: <intdb:#> .
db:1 db:name "One"; db:val 1.
db:2 db:name "Two"; db:val 2.
db:3 db:name "Three"; db:val 3.
db:4 db:name "Four"; db:val 4.
db:5 db:name "Five"; db:val 5.
```

Listing 2.1: G_1 in N3.

```
@prefix db: <intdb:#> .
db:2 db:has-property db:isprime.
db:3 db:has-property db:isprime.
db:5 db:has-property db:isprime.
```

Listing 2.2: G_2 in N3.

Definition 2.1.7 (RDF Graph Merge)**.** *For two RDF graphs G_1 and G_2 the merge is defined as $G_{new} := G_1 \cup G_2$ [8].*

The graphs G_1 and G_2 are sketched in Figure 2.3. Their merged graph is represented in Figure 2.4.

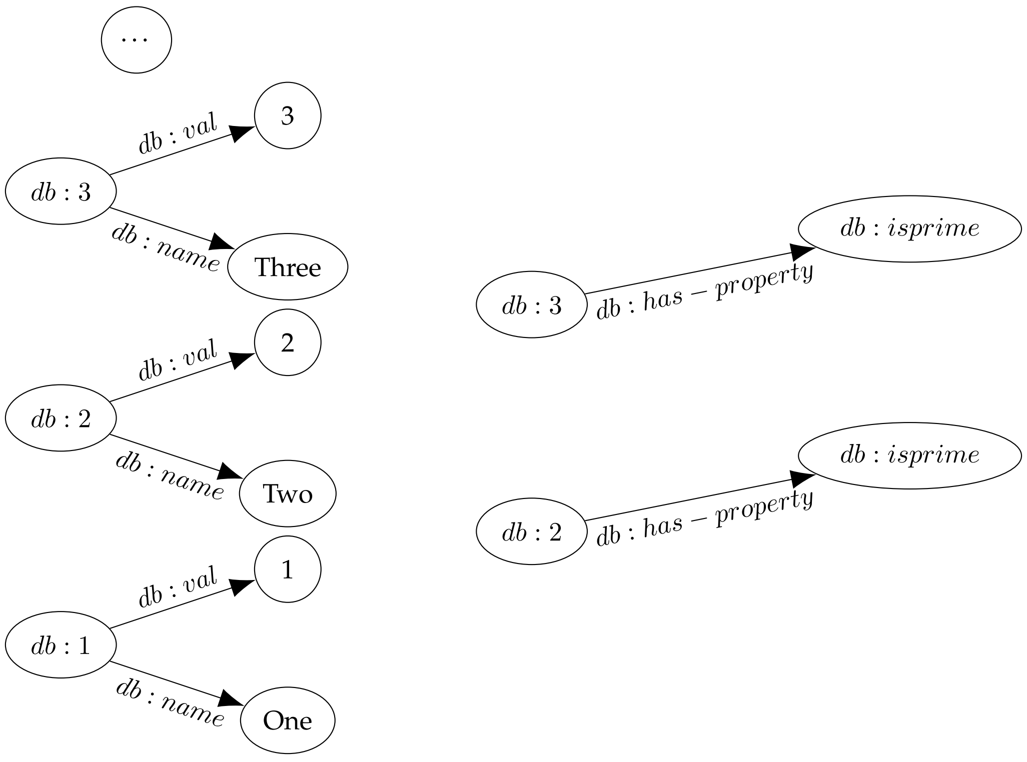

Figure 2.3: Graph representations of G_1 and G_2.

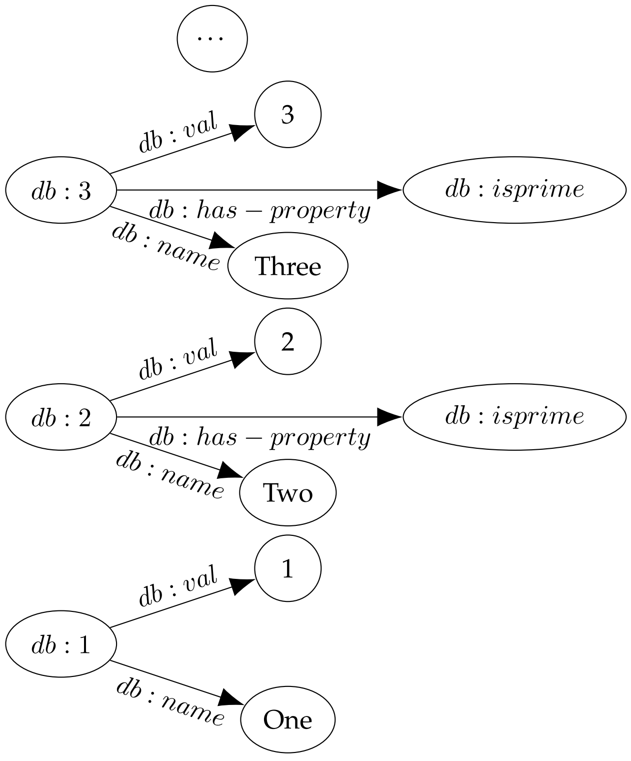

Figure 2.4: Graph representation of the merge G_{new} of G_1 and G_2.

Definition 2.1.8 (RDF Graph Isomorphism)**.** *The two RDF graphs G and $\hat{G}$ are isomorphic [8] , if there is a bijective function $\Phi : V(G) \to V(\hat{G})$, s.t. :*

1. *Φ maps BNs to BNs.*
2. $\Phi(l) = l\ , with\ l \in L \cap V(G)$
3. $\Phi(i) = i\ , with\ i \in I \cap V(G)$
4. $(s, p, o) \in G \iff (\Phi(s), p, \Phi(o)) \in \hat{G}$

This definition states that all graphs, being isomorphic by exchanging BNs, represent the same information [8].

2.2 SPARQL

Having the previous definitions for RDF graphs at hand, this section proceeds with the introduction of the corresponding query language SPARQL. The goal is to give an intuition on the actual evaluation process of SPARQL; starting from the initial string, which is representing the query, pursuing with the SPARQL algebra operators and their evaluation.

Definition 2.2.1 (SPARQL Query)**.** *A SPARQL abstract query is a tuple $Q := (E, D, QF)$ where [7]:*

- *E is a SPARQL algebra expression.*
- *D is a RDF-Dataset.*
- *QF is the query form.*

Commonly, E is also known as the WHERE segment of the query, D as the FROM and QF as the SELECT/CONSTRUCT/ASK/DESCRIBE segment.

2.2.1 SPARQL Datasets

The definition of a RDF dataset extends the notation for multiple RDF graphs. This notation will be used to introduce upcoming operations.

Definition 2.2.2 (RDF Dataset)**.** *A RDF dataset is a set:*

$$D := \{G, (\langle u_1 \rangle, G_1), (\langle u_2 \rangle, G_2), \ldots, (\langle u_n \rangle, G_n)\}$$

where G and G_i are RDF graphs, and each $\langle u_i \rangle$ is a different IRI representing the respective graph's name. G is called the default graph. [7]

Two datasets can be merged by performing a RDF graph merge on all corresponding graphs with the same name.

2.2.2 SPARQL Solution Mappings

With the definition of the underlying data, more evaluation-specific definitions are given in the following subsection.

Definition 2.2.3 (Query Variable)**.** *A query variable is an element of the set V, where V is infinite and disjoint from the RDF terms T [7].*

The introduction of the query variables V allows for the extension of RDF triples to triple patterns.

Definition 2.2.4 (Triple Pattern)**.** *A triple pattern is an element of the set [7]:*

$$(I \cup B \cup V) \times (I \cup V) \times (T \cup V)$$

Examples for triple pattern are:

$$(db:1,\ db:name,\ ?X) \quad (db:1,\ ?Y,\ ''One'') \quad (?Z,\ db:name,\ ''One'')$$

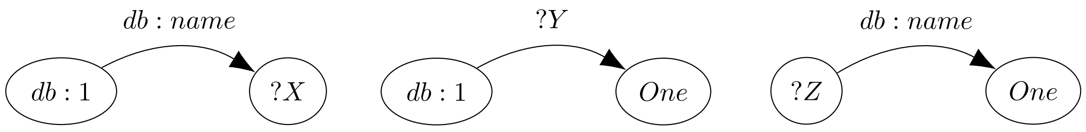

Figure 2.5: Graph representation of Triple Pattern.

Definition 2.2.5 (Basic Graph Pattern (BGP))**.** *A BGP is a set of triple patterns. This effectively forms a graph with the possibility of variables replacing subjects, predicates and objects. [7]*

Definition 2.2.6 (Solution Mapping)**.** *A solution mapping μ is a partial function:*

$$\mu : V_{pattern} \rightarrow T$$

The domain of μ, $dom(\mu)$ is the subset $V_{pattern}$ of V where μ is defined [7]. The solution mapping is only a partial function, because variables can be declared as optional.

The solution mapping gives possible bindings for variables in a pattern according to the criteria, presented in the next section about evaluation. If only certain variables are required, they can be individually selected by a specification in the SELECT clause, else the $\star$ operator can be used to request the complete solution mapping. Further, a solution mapping can be interpreted as a list instead of a set. This new list is called solution sequence and can be modified in order to shape the solutions [7].

Definition 2.2.7 (Solution Sequence Modifiers)**.** *The most common solution sequence modifiers for shaping the results are [7]:*

- *ORDER BY: puts the solutions in a specific order.*
- *Projection: chooses only certain variables with SELECT.*
- *DISTINCT: ensures unique solutions in the sequence by omitting duplicates.*
- *OFFSET: specifies a position from which solutions are to be considered (only useful with ORDER BY).*
- *LIMIT: restricts the number of solutions.*

2.2.3 SPARQL Algebra Evaluation

In order to give an intuition on the evaluation of algebra expressions, the BGP, FILTER and OPTIONAL operator are introduced in detail below. Therefore, the RDF examples from the Figures 2.1 and 2.2 are used.

BGP Evaluation

The basic operator of the SPARQL algebra is the BGP. Most of the other operators are based on the results of BGP evaluations. The results of the evaluation of algebra expressions are solution mappings with different properties.

Definition 2.2.8 (BGP Semantics)**.** *With $\mu(BGP)$ a solution mapping function that replaces all variables in BGP by suitable RDF terms:*

$$[\![BGP]\!]_D \coloneqq \{\mu : V_{BGP} \rightarrow T \mid \mu(BGP) \subseteq D\}$$

This means all possibilities of replacing variables in the BGP with RDF terms, such that the result is a subgraph of the dataset. [3].

A small example of a BGP in a SPARQL query is given in Listing 2.3. The corresponding algebra operator in abstract form, shown in Listing 2.4, evaluates to the solution mappings in Table 2.1.

```
PREFIX  :     <intdb:#>

SELECT  *
FROM <numbers.n3>
WHERE
  { ?A  :val  ?B }
```

Listing 2.3: Simple example of a BGP in SPARQL.

```
(prefix ((: <intdb:#>))
  (bgp (triple ?A :val ?B)))
```

Listing 2.4: Algebra form of the query in Listing 2.3.

	?A	?B
μ_1	intdb:#2	2
μ_2	intdb:#4	4
μ_3	intdb:#1	1
μ_4	intdb:#3	3
μ_5	intdb:#5	5

Table 2.1: Solution Mappings to the query in Listing 2.3.

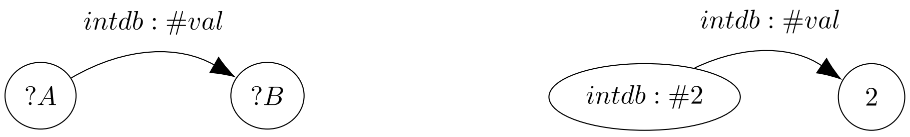

Figure 2.6: Graph representation of BGP on the left and $\mu_1(BGP)$ on the right.

FILTER Evaluation

The FILTER operator is one of the core operators of the SPARQL algebra [1,3]. It is used to restrict the set of possible solution mappings according to some parameters.

Definition 2.2.9 (FILTER Semantics)**.**

$$[\![BGP\ FILTER\ R]\!]_D := \{\mu \mid \mu \in [\![BGP]\!]_D,\ [\![R]\!]_\mu = true\}$$

This basically means all mappings from the BGP evaluation with restrictions R on the values of μ [3].

A small example of a FILTER in a SPARQL query is given in Listing 2.5. The corresponding algebra operator in abstract form, shown in Listing 2.6, evaluates to the solution mappings in Table 2.2.

```
PREFIX  :     <intdb:#>

SELECT  *
FROM <numbers.n3>
WHERE
  { ?A  :val  ?B
    FILTER ( ?B < 3 )
  }
```

Listing 2.5: Simple example of a FILTER in SPARQL.

```
(prefix ((: <intdb:#>))
  (filter (< ?B 3)
    (bgp (triple ?A :val ?B))))
```

Listing 2.6: Algebra form of the query in Listing 2.5.

?A	?B
intdb:#2	2
intdb:#1	1

Table 2.2: Solution Mappings to the query in Listing 2.5.

OPTIONAL Evaluation

The OPTIONAL operator is also one of the core operators of the SPARQL algebra [1,3]. It can be used to make parts of a BGP optional. This means that variables in these pattern are only matched if possible and omitted otherwise, thus making the set of solution mappings partial functions. The evaluation of the OPTIONAL operator is split into two parts. First, the BGP is evaluated once with the optional part and then without the optional part for all solution mappings not occurring in the previous case. Afterwards, both sets of solution mappings are combined with a set union operator. This procedure is commonly known as a `left join` in the database context.

A small example of an OPTIONAL in a SPARQL query is given in Listing 2.7. The corresponding algebra operator in abstract form, shown in Listing 2.8, evaluates to the solution mappings in Table 2.3.

```
PREFIX  :     <intdb:#>

SELECT  *
FROM <numbers.n3>
WHERE
  { ?A  :val  ?B
    OPTIONAL
      { ?A  :has-property  ?C }
  }
```

Listing 2.7: Simple example of an OPTIONAL in SPARQL.

```
(prefix ((: <intdb:#>))
  (leftjoin
    (bgp (triple ?A :val ?B))
    (bgp (triple ?A :has-property ?C))))
```

Listing 2.8: Algebra form of the query in Listing 2.7.

?A	?B	?C
intdb:#2	2	intdb:#isprime
intdb:#4	4	
intdb:#1	1	
intdb:#3	3	intdb:#isprime
intdb:#5	5	intdb:#isprime

Table 2.3: Solution Mappings to the OPTIONAL query in Listing 2.7.

2.2.4 SPARQL Nested Queries

The following subsection gives an introduction of nested queries and particularly nested CONSTRUCTs.

Definition 2.2.10 (Nested Query). *Let $Q = (E, D, QF)$ be a query. A query $\hat{Q}$ is nested in Q, if and only if $\hat{Q}$ occurs in Q as part of one of the expressions E, D or QF. Q is then referred to as the outer query and $\hat{Q}$ as the inner query. [2]*

Definition 2.2.11 (CONSTRUCT Query). *Is a query $Q = (E, D, CONSTRUCT)$. Instead of selecting variables, a CONSTRUCT query defines triple templates for the creation of new RDF triples. These templates may contain variables from the solution mappings of the WHERE evaluation.*

As an example, Listing 2.9 shows the creation of a new relation `isDoubleOf` between suitable numbers. The results of this query are shown in Figure 2.7.

```
PREFIX : <intdb:#>

CONSTRUCT {?B :isDoubleOf ?A}
FROM <file:numbers.n3>
WHERE {
        ?A :val ?Aval.
        ?B :val ?Bval.
        FILTER(?Aval * 2 = ?Bval)}
```

Listing 2.9: Simple example of a CONSTRUCT query.

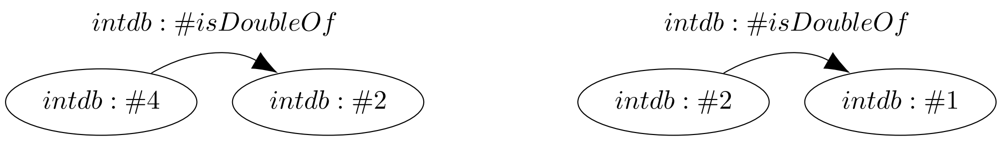

Figure 2.7: Graph representation of the results of Listing 2.9.

Definition 2.2.12 (DESCRIBE Query). *Is a query $Q = (E, D, DESCRIBE)$. It returns a RDF graph as information about a resource. This RDF graph may contain triples to a certain depth depending on its implementation with the original resource as root node.*

As an example, Listing 2.10 gives the information about the `<intdb:#2>` resource shown in Figure 2.8.

```
PREFIX : <intdb:#>

DESCRIBE ?A
FROM <file:numbers.n3>
WHERE { ?A :val 2}
```

Listing 2.10: Simple example of a DESCRIBE query.

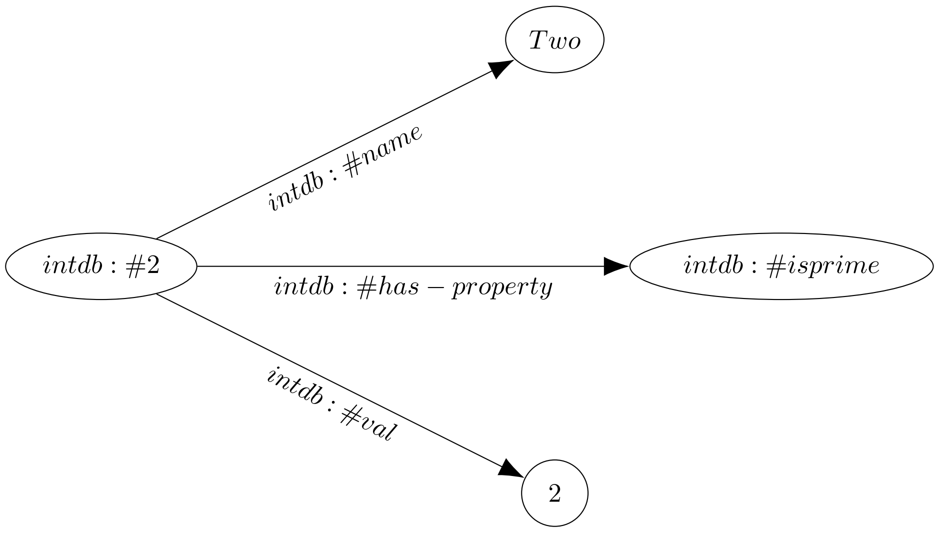

Figure 2.8: Graph representation of the result of Listing 2.10.

A nested CONSTRUCT is defined as a special case of nested queries.

Definition 2.2.13 (Nested CONSTRUCT Query). *A nested query $\hat{Q}$ is a nested CONSTRUCT query, if it produces RDF data within the the D segment of an outer query Q. Possible query types for nested CONSTRUCT queries are CONSTRUCT and DESCRIBE. The outer query merges the data from all nested CONSTRUCTs and RDF files to obtain its D [2].*

The possibility of correlated queries is omitted in this thesis, but can be considered in future work.

Definition 2.2.14 (Query Correlation). *The queries Q and $\hat{Q}$ are correlated, if and only if $\hat{Q}$ is nested in Q and variables occur in both graph pattern of Q and $\hat{Q}$. These variables are called correlated variables. [2]*

2.3 Ontologies & Reasoning

An ontology defines the vocabulary for concepts and categories in a specific context. In the context of data, it is mostly used to describe classes and relationships as well as relations between these classes and properties. The most common ontology languages for RDF KBs are OWL and RDF Schema (RDFS). These languages have reserved prefixes and are officially recommended by the W3C. RDFS is mainly concerned with the schema of classes, e.g. their hierarchy, domains and ranges of properties. OWL is used for modelling more complex knowledge like relationships between classes or class properties and their restrictions. With these tools at hand it is possible to model higher level knowledge like:

- All members of the class "Child" have exactly two parents (Cardinality restriction).
- All member of the class "Parent" must have at least on child. (Cardinality/Existential restriction)
- The class "Parent" is a subclass of "Ancestor" (Hierarchy).
- The classes "Mother" and "Father" have disjoint sets of individuals (Class relationship).

This formalisation is very important when it comes to reasoning. In general, reasoning describes the process of deriving implicit knowledge. Sticking to the examples above, that means that an individual of the class "Child" must have two corresponding parents, even if there are no individuals in the current state of the KB [9]. This logical assumption allows this knowledge to be modelled only implicitly.

2.3.1 Open World vs. Closed World

OWL is essential for reasoning over KBs, because a LOD KB is subject to the Open World Assumption (OWA). The OWA describes the assumption, that the information in a KB may always be incomplete. Contrary to this assumption, the Closed World Assumption (CWA) KBs are always assumed to be complete. As an example, a KB of family members is considered. In order to deduce the number of children of one individual under CWA, it is required to simply count the suitable relations. Under OWA the simple count of relations would be a lower bound, because it is possible that there are several children unknown to the KB. That is why additional information is required in order to deduce such knowledge, e.g. about the cardinality of this property [1,9].

2.3.2 OWL and Logic

Modelling complex knowledge is often based on a formal logic. Using a highly expressive language to represent the knowledge, enables logical reasoning on the KB. Usually a highly expressive language yields high computational complexity on the reasoning process or even undecidability. Tackling this problem, different restrictions in the underlying logic can be used to balance expressivity and efficient reasoning. The OWL is based on first-order logic and can be divided into three main categories. OWL Full is the most expressive of these, but undecidable. As a subset of OWL Full there is OWL Description Logic (DL) which is decidable and has a worst case computational complexity of NExpTime. A further subset of OWL DL is OWL Lite which is decidable and has a worst case computational complexity of ExpTime. The most used fragment in the literature and also used in this thesis is OWL DL [9].

The syntax of OWL is based on RDF and can therefore be represented as a directed graph. In order to distinguish this terminological knowledge from the assertional information about individuals, the literature splits the KB into the TBox and ABox respectively [1, 9].

2.3.3 OWL DL

OWL DL is a subcategory of OWL Full and is based on description logics. Description Logics are decidable fragments of first-order predicate logic. A logic is decidable, if for every inference problem in this language exists a terminating algorithm for deciding it. Different variations of DL are concerned with a favourable trade off between expressivity and scalability. In order to provide decidability, some language constructs of OWL Full must be forbidden. For example, the combined use of inverse and transitive properties is not allowed. Further restrictions can be found in the literature [9]. The most efficient OWL DL reasoning engines are based on tableaux algorithms. Tableaux algorithms solve very complex worst case complexity problems, but are often combined with heuristics in order to improve to a good average case performance [9].

2.3.4 Pellet

Pellet is as a OWL DL reasoner, written in Java. It is available in both a commercial and an open source version. The Semwebjar tool, presented in Section 2.5, uses the open sourcer version openllet. Its underlying description logic is commonly found as $\mathcal{SHOIQ}$ and since OWL 2 DL $\mathcal{SROIQ}$ [9].

Pellet supports conjunctive queries using SPARQL. Conjunctive queries extend the underlying data model to a point, where it supports OWL DL. These models provide different views on the combined ABox and TBox. One of the most basic difference to a normal model is the assertion of TBox data to every suitable instance of the ABox [9].

2.4 Apache Jena Framework

"A free and open source Java framework for building Semantic Web and Linked Data applications." [11]

The Apache Jena Framework provides the possibility to create and read RDF data. Furthermore, it can expose them as an endpoint, enables the work with different models like OWL and reasoning about data [11]. While all of these features are important in the LOD and Semantic Web development, the main component of Jena is ARQ. ARQ is the query engine that evaluates the SPARQL query language against RDF data and will be altered in this thesis [12]. Jena is an open source project from the Apache Foundation and provides full access to the source code on GitHub [13]. The documentation for using ARQ in applications can be found on the official website [14].

2.5 Semwebjar

The Semwebjar tool is developed and run by the Databases and Information Systems (DBIS) group [15] in Göttingen. It is a wrapping tool for the core ARQ engine of the Jena framework. This wrapper provides different possibilities for adding different reasoners, command line arguments and the deployment on an Apache Tomcat web server [16]. The handling and setup of the tool and the web server is described on the DBIS page as well as in the practical report [17, 18]. Implementations of nested CONSTRUCTs will mainly be tested with the Semwebjar tool.

2.6 JavaCC

"Java Compiler Compiler is the most popular parser generator for use with java." [19]

This tool reads in grammar specifications, written in a Java-like language, and writes out Java source code. From this source code a Java parser can be compiled, which parses matching expressions according to the specified grammar. Java classes and functions can be imported and used during the parsing process, e.g. to fill own containers and objects with respective information. A small example of the JavaCC language is given by the toy example from the official JavaCC documentation in Listing 2.11 and Listing 2.12 [19]. These code snippets represent the grammar with the two production rules:

$$Start \rightarrow aNc$$
$$N \rightarrow b \mid bc$$

Therefore, the grammar accepts the words "abc" and "abcc".

```
void Start() :
{}
{
  "a" N() "c"
}
```

Listing 2.11: Production rule of *Start*.

```
void N() :
{}
{
  "b" [ "c" ]
}
```

Listing 2.12: Production rule of *N*.

2.7 Mondial Database

Mondial [20] is a database especially designed for academic case studies and teaching purposes. It is used by the DBIS group [15] in various lectures like Semantic Web [1] or Database Foundations [6]. Mondial is nowadays available in XML, Datalog, F-Logic, RDF and in various relational formats. This thesis uses the RDF version. The database contains geographical entities and their properties, such as countries, cities, rivers, mountains and so on. As an excerpt, Listing 2.13 shows some information in the Mondial database about Albania in the N3 format.

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@base <http://www.semwebtech.org/mondial/10/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
 <countries/AL/>  rdf:type :Country ;
     :name "Albania"   ;
     :carCode 'AL' ;
     :area 28750 ;
     :capital <countries/AL/cities/Tirana/> ;
     :population 2821977 ;
     :hadPopulation [ a :PopulationCount; :year 1950; :value 1214489] , [ a :
         PopulationCount; :year 1960; :value 1618829] , [ a :PopulationCount; :year
         1970; :value 2138966] , [ a :PopulationCount; :year 1980; :value 2734776] , [ a
          :PopulationCount; :year 1990; :value 3446882] , [ a :PopulationCount; :year
         1997; :value 3249136] , [ a :PopulationCount; :year 2000; :value 3304948] , [ a
          :PopulationCount; :year 2001; :value 3069275] , [ a :PopulationCount; :year
         2011; :value 2821977] ;
     :populationGrowth 0.3 ;
     :infantMortality 13.19 ;
     :gdpTotal 12800 ;
     :gdpInd 12 ;
     :gdpServ 68.5 ;
     :gdpAgri 19.5 ;
     :inflation 1.7 ;
     :unemployment 16.9 ;
     :government "parliamentary democracy" ;
     :independenceDate '1912-11-28'^^xsd:date ;
     :wasDependentOf
              <politicalbodies/Ottoman+Empire/>  .
 <politicalbodies/Ottoman+Empire/> rdf:type :PoliticalBody  .
 ...
 ...
```

Listing 2.13: Information about Albania in the Mondial database

Chapter 3

SPARQL Grammar

In order to introduce an extended version of SPARQL, the specifications and implementation have to be adjusted accordingly. This chapter is concerned with the changes in the grammar as well as the corresponding elements of Jena.

3.1 Grammar Specification

This section describes all necessary changes in the grammar of SPARQL as it is described in the official W3C recommendation [7]. Since the full grammar is very large only the relevant snippets will be examined.

The excerpt below shows the production rules 13 to 16 of the original grammar beginning with the $DatasetClause$ [7]. Additionaly, `ConstructQuery` serves as a query type example. Every data providing statement, initialised by the FROM keyword, is parsed in this part of the grammar. When the keyword FROM is read, it is followed by either a default graph description or a named graph description. The respective branch is determined by the first following token. If there is the NAMED keyword, the next IRI token not only determines the graphs source, but also its name. In the other case, the IRI only describes a source of RDF data, which has to be stored in the default graph.

$$
\begin{array}{lll}
P \coloneqq \{ & & \\
 & \ldots & \\
 & ConstructQuery & ::= \text{“}CONSTRUCT\text{“}\ldots \\
 & DatasetClause & ::= \text{“}FROM\text{“} \quad (\, DefaultGraphClause \mid NamedGraphClause \,) \\
 & DefaultGraphClause & ::= SourceSelector \\
 & NamedGraphClause & ::= \text{“}NAMED\text{“} \quad SourceSelector \\
 & SourceSelector & ::= iri \\
 & \ldots & \\
 \} & &
\end{array}
$$

An altered version of the previously introduced grammar production rules is proposed in the specification below.

$$
\begin{array}{lll}
P \coloneqq \{ & & \\
 & \ldots & \\
 & ConstructQuery & ::= \text{“}CONSTRUCT\text{“} \quad (\, \text{“}\mathbf{REASONER}\text{“} \,)? \ \ldots \\
 & DatasetClause & ::= \text{“}FROM\text{“} \quad (\, DefaultGraphClause \mid NamedGraphClause \,) \\
 & DefaultGraphClause & ::= (\, SourceSelector \mid \mathbf{SubFrom} \,) \\
 & NamedGraphClause & ::= \text{“}NAMED\text{“} \quad SourceSelector \\
 & \mathbf{SubFrom} & ::= \text{“}\{\text{“} \quad (\, SubFromNoService \mid SubFromService \,) \quad \text{“}\}\text{“} \\
 & \mathbf{SubFromNoService} & ::= (\, ConstructQuery \mid DescribeQuery \,) \\
 & \mathbf{SubFromService} & ::= \text{“}SERVICE\text{“} \quad iri \quad (\, ConstructQuery \mid DescribeQuery \,) \\
 & SourceSelector & ::= iri \\
 & \ldots & \\
 \} & &
\end{array}
$$

The grammar now accepts nested CONSTRUCT and DESCRIBE queries initialised by a curly brace after the FROM clause. Furthermore, a remote CONSTRUCT/DESCRIBE query to SPARQL endpoints is inserted. This enables the addition of remote data to the local dataset. Therefore, the SERVICE keyword with an additional IRI, followed by the query, shall return the respective RDF graph. Additionally, the optional keyword REASONER is introduced after each query form indicating the support of the Pellet reasoner in the respective query. With this modified language specification, an appropriate parser can be build automatically. This new parser is allowed to parse the altered grammar and handle the events according to the definitions of the new features.

3.2 Grammar Implementation

A practical implementation of the specifications from the previous section requires the adjustment of `master.jj` and `SPARQLParserBase.java`. These files are placed in the `Grammar` and `src` folder of the ARQ-root respectively. While `master.jj` contains the specifications of the grammar in the javacc format, `SPARQLParserBase.java` provides necessary functions and data structures needed to handle the parsed statements. All the information of the query string will be stored in an internal query object, which is an instance of the class `Query.java`. Since the query object is not relevant for the parser in order to accept nested CONSTRUCTs, its alteration will be presented later in Chapter 4.

Beginning with the `master.jj` file, Listing 3.1 adds a choice option for the parser in the `DefaultGraphClause` branch. With `SubFrom` a new alternative branch is given, instead of reading an IRI for a RDF graph location.

```
void DefaultGraphClause() : { String iri ; }
{
  (iri = SourceSelector() { getQuery().addGraphURI(iri);} | SubFrom() )
}
```

Listing 3.1: Modification of `DefaultGraphClause` in `master.jj`.

`SubFrom` is chosen, if the token after the FROM is an opening curly brace "{", here `<LBRACE>`. When this constellation occurs, either a `<SERVICE>` token is read for the `SubFromService` branch or the `SubFromNoService` branch is chosen. Finally a closing curly brace "}", here `<RBRACE>`, must be read to close the nested CONSTRUCT. This is presented in Listing 3.2 below.

```
void SubFrom() : {}
{
  <LBRACE>
  ( SubFromNoService() | SubFromService() )
  <RBRACE>
}
```

Listing 3.2: `SubFrom` in `master.jj`.

If the parser selects the `SubFromNoService` branch (Listing 3.3), the nested CONSTRUCT has to be initiated with `startSubFromNoService` function from `SPARQLParserBase.java`. This function, described in Listing 3.5, makes sure the inner query will be stored in a separate query object. Then the nested query is parsed as a normal CONSTRUCT or DESCRIBE query. In the end, the `endSubFROMNoService` function returns the inner query, which is stored via the `addSubFromNoServiceQuery` function as a nested query of the outer query object.

```
void SubFromNoService() : {Token t ;}
{
  { startSubFromNoService(); }
  ( ConstructQuery() | DescribeQuery() )
  {Query q = endSubFromNoService(); getQuery().addSubFromNoServiceQuery(q);}
}
```

Listing 3.3: `SubFromNoService` in `master.jj`.

When the `SubFromService` branch (Listing 3.4) is chosen, the processing of the inner query is very similar to the `SubFromNoService`. Additionally to the regular parsing process of the inner query, the IRI of the SERVICE endpoint is parsed and stored together with the nested query.

```
void SubFromService() : {String n ;}
{
  <SERVICE>
  n = IRIREF()
  { startSubFromService(); }
  ( ConstructQuery() | DescribeQuery() )
  {Query q = endSubFromService(); getQuery().addSubFromServiceQuery(q,n);}
}
```

Listing 3.4: `SubFromService` in `master.jj`.

The parser is able to create a tree-like nesting of the queries within the query object by operating on a stack. An outer query is pushed to the stack, when the beginning of a nested query is parsed. Then a new query object with the same prologue ,i.e. prefixes and context, is created and used for further parsing. After finishing the parsing of the inner query, the inner query object is returned and the outer query is reattached by the pop operation of the stack. These functions are defined in `SPARQLParserBase.java` as shown in Listings 3.5 and 3.6. There is no functional difference in `startSubFromService` and `startSubFromNoService`. They have been split into separate functions for debugging purposes.

```
protected void startSubFromService()
{
    pushQuery();
    query = newSubQuery(getPrologue()) ;
}

protected void startSubFromNoService()
{
    pushQuery();
    query = newSubQuery(getPrologue()) ;
}
```

Listing 3.5: Start of a nested query in `SPARQLParserBase.java`.

```
protected Query endSubFromService()
{
    Query subQuery = query;
    popQuery();
    return subQuery;
}

protected Query endSubFromNoService()
{
    Query subQuery = query;
    popQuery();
    return subQuery;
}
```

Listing 3.6: End of a nested query in `SPARQLParserBase.java`.

In order to use the optional keyword REASONER as an indicator for an underlying Pellet model the new token is introduced in Listing 3.7. Afterwards, this token can be used to trigger the `setReasoner` function. This is exemplary shown in Listing 3.8 with a CONSTRUCT query.

```
TOKEN:
{
...
| < REASONER:  "REASONER" >
...
}
```

Listing 3.7: Introduction of REASONER Token in `master.jj`.

```
//Also add the reasoner to the SELECT, ASK, DESCRIBE queries
void ConstructQuery() : { Template t ;
                          QuadAcc acc = new QuadAcc() ; }
{
 <CONSTRUCT>
   { getQuery().setQueryConstructType() ; }
  ( <REASONER> { getQuery().setReasoner(true);} )?
  ...
```

Listing 3.8: REASONER Token usage example in a CONSTRUCT query (`master.jj`).

These changes are not covered by the default build script. Therefore, the new parser needs to be built manually with the `grammar` bash script next to the `master.jj` file in the `Grammar` folder. The target path for the resulting Java files needs to be checked before the execution.

Chapter 4

Jena Implementation

While the previous Chapter introduced the new grammar, this chapter presents the changes inside the Jena ARQ source code. Therefore, the approximate evaluation process of Jena with the idea of implementing nested CONSTRUCTs is sketched. Afterwards, the individual changes are documented in order to reproduce the implementation results of this thesis.

4.1 Dataflow in Jena

The Figure 4.1 illustrates the path of a query from parsing to execution. Orange containers represent the variable input files which determine the outcome of a query. `Foo.sparql` contains the string representation of a query. Depending on the parser specification in `master.jj`, there are different possibilities of parsing this query string, e.g. accepting or rejecting nested CONSTRUCTs. With references to dataset sources, for example a `Bar.rdf` file, a query object can be build by the `QueryFactory`. The `DatasetFactory` takes these dataset sources and creates a model. This model is used together with an algebra plan, derived from the query object, to create a `QueryExecution` object in the `QueryExecutionFactory`. As a new feature, the query object shall contain references to nested CONSTRUCT queries. These inner queries go through the same execution process as the outer query recursively and merge their resulting RDF graphs with the outer query's model as described in Definition 2.1.7. The new features are marked with red rectangles in the sketch of Figure 4.1.

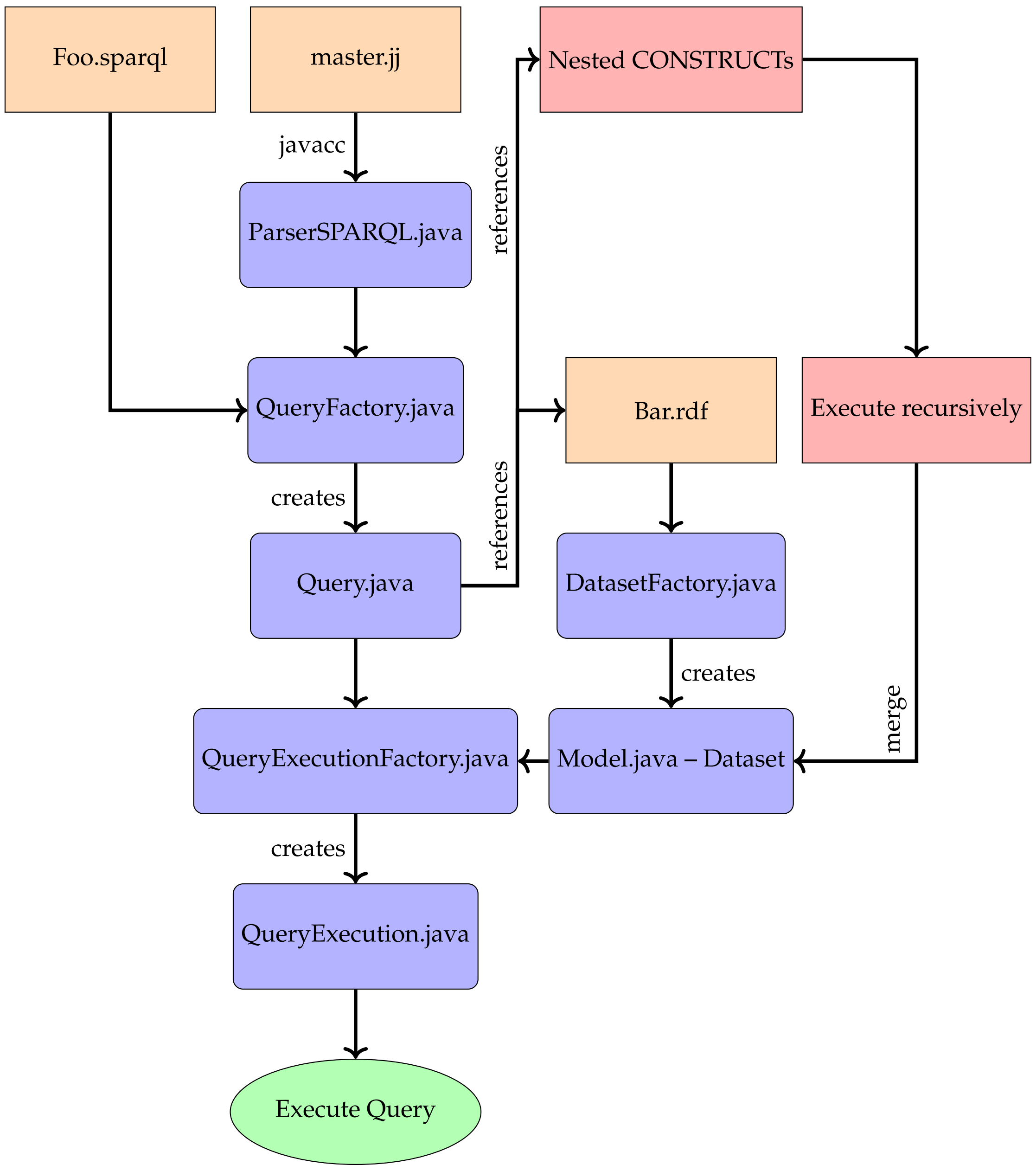

Figure 4.1: Sketch of the new data flow in a SPARQL query execution process.

From Figure 4.1 it can be concluded, that the relevant classes are located in between `Query.java` and the output of `DatasetFactory.java`. The next section specifies the necessary altering of these corresponding Java files.

4.2 Modification of Jena Classes

Scanning through the Java classes in between `Query.java` and `DatasetFactory.java` yields four classes to be modified. Firstly, `Query.java` is required to implement containers for the different nested CONSTRUCT variations and a flag to enable reasoner support. Afterwards, `DatasetUtils.java` uses a `DatasetDescription.java` object to create a dataset graph. This object is returned by a getter function of the query object. In the end, `QueryEngineBase.java` needs to be changed. In order to enable reasoning, an alternative underlying model needs to be accessible. Hence, these four classes are sufficient to be altered; therefore providing minimal code changes by using as much of the given structures and functions as possible. The most important changes are outlined in the following.

4.2.1 Modification of Query.java

A query object needs to be able to store all kinds of occurring nested CONSTRUCTs, i.e. CONSTRUCT and DESCRIBE queries as well as these two as a remote service query. The former two options are stored in lists serving as containers for the parsed nested queries. The latter have to be stored together with their corresponding service address. In the example presented in Listing 4.1 this is implemented with a `HashMap`.

```java
private List<Query> nestedConstructQueries = new ArrayList<>();
private List<Query> nestedDescribeQueries = new ArrayList<>();
private List<HashMap<String,Object>> nestedServiceConstructQueries = new ArrayList<>();
private List<HashMap<String,Object>> nestedServiceDescribeQueries = new ArrayList<>();
```

Listing 4.1: Lists for nested CONSTRUCT queries in the `Query` object.

In order to access the information of possible reasoner support, a simple boolean with getter and setter functions is added to the query object. This can be seen in Listing 4.2.

```java
protected boolean reasoner = false;
public void setReasoner(boolean b) { reasoner = b ; }
public boolean isReasoner()        { return reasoner ; }
```

Listing 4.2: REASONER boolean with setter and getter.

Inserting the nested queries into the intended lists is done by the functions `addSubFromNoServiceQuery` in Listing 4.3 and `addSubFromServiceQuery` in Listing 4.4. The first one shall be called by the parser in absence of the `SERVICE` keyword. It checks the queries for their respective query type and uses add operations for list insertion. The second function is called by the parser in presence of the `SERVICE` keyword in combination with an IRI. It inserts the IRI with the key “First” and the query with the key “Second” into a `HashMap`, which is then added into the respective list.

```java
public void addSubFromNoServiceQuery(Query q)
{
    if(q.isConstructType()) {
        if(nestedConstructQueries == null)
            nestedConstructQueries = new ArrayList<>();
        nestedConstructQueries.add(q);
    } else if (q.isDescribeType()) {
        if(nestedDescribeQueries == null)
            nestedDescribeQueries = new ArrayList<>();
        nestedDescribeQueries.add(q);
    } else
        throw new QueryException("Nested CONSTRUCT was not added:\n" + q.serialize()) ;
}
```

Listing 4.3: Adding nested CONSTRUCT queries to the Lists.

```java
public void addSubFromServiceQuery(Query q, String n)
{
    if(q.isConstructType()) {
        if(nestedServiceConstructQueries == null)
            nestedServiceConstructQueries = new ArrayList<>();
        HashMap<String,Object> tmpmap = new HashMap<>();
        tmpmap.put("First", q);
        tmpmap.put("Second", n);
        nestedServiceConstructQueries.add(tmpmap);
    } else if (q.isDescribeType()) {
        if(nestedServiceDescribeQueries == null)
            nestedServiceDescribeQueries = new ArrayList<>();
        HashMap<String,Object> tmpmap = new HashMap<>();
        tmpmap.put("First", q);
        tmpmap.put("Second", n);
        nestedServiceDescribeQueries.add(tmpmap);
    } else
        throw new QueryException("Nested service-CONSTRUCT was not added:\n" + q.
            ↪ serialize()) ;
}
```

Listing 4.4: Adding nested service-CONSTRUCT queries to the Lists.

Listing 4.5 shows the getter functions needed for further processing later on.

```java
public List<Query> getNestedConstructQueries(){
    return nestedConstructQueries;}
public boolean usesNestedConstructQuery(Query q){
    return nestedConstructQueries.contains(q);}
public List<Query> getNestedDescribeQueries(){
    return nestedDescribeQueries;}
public boolean usesNestedDescribeQuery(Query q){
    return nestedDescribeQueries.contains(q);}
public List<HashMap<String,Object>> getNestedServiceConstructQueries(){
    return nestedServiceConstructQueries;}
public List<HashMap<String,Object>> getNestedServiceDescribeQueries(){
    return nestedServiceDescribeQueries;}
```

Listing 4.5: Adding getter to make nested CONSTRUCTs available from the outside.

All information about the dataset, which is described within a query string, is compactly represented in a `DatasetDescription` object. This will be adjusted accordingly in Subsection 4.2.2. The query object is able to provide the information about the presence of such a dataset and creates a `DatasetDescription` object from its internal state. These two tasks are performed by the functions `hasDatasetDescription` and `getDatasetDescription`. While the first one checks for non-emptiness of all possible data sources, the second one adds all of them together and returns this as a `DatasetDescription`. The extended version of these two functions are presented in Listing 4.6 and Listing 4.7.

```
public boolean hasDatasetDescription()
{
    if ( getGraphURIs() != null && getGraphURIs().size() > 0 )
        return true ;
    if ( getNamedGraphURIs() != null && getNamedGraphURIs().size() > 0 )
        return true ;
    if ( getNestedConstructQueries() != null && getNestedConstructQueries().size() > 0 )
        return true;
    if ( getNestedDescribeQueries() != null && getNestedDescribeQueries().size() > 0 )
        return true;
    if ( getNestedServiceConstructQueries() != null && getNestedServiceConstructQueries
        ↪ ().size() > 0 )
        return true;
    if ( getNestedServiceDescribeQueries() != null && getNestedServiceDescribeQueries().
        ↪ size() > 0 )
        return true;
    return false ;
}
```

Listing 4.6: Update of `hasDatasetDescription`.

```
public DatasetDescription getDatasetDescription()
{
    if ( ! hasDatasetDescription() )
        return null;

    DatasetDescription description = new DatasetDescription() ;

    description.addAllDefaultGraphURIs(getGraphURIs()) ;
    description.addAllNamedGraphURIs(getNamedGraphURIs()) ;
    description.addAllDefaultConstructQuerys(getNestedConstructQueries());
    description.addAllDefaultDescribeQuerys(getNestedDescribeQueries());
    description.addAllDefaultServiceConstructQuerys(getNestedServiceConstructQueries());
    description.addAllDefaultServiceDescribeQuerys(getNestedServiceDescribeQueries());
    return description ;
}
```

Listing 4.7: Update of `getDatasetDescription`.

4.2.2 Modification of DatasetDescription.java

All the changes from Subsection 4.2.1, that lead to an altered `DatasetDescription` object, need to be accepted by this very object. Listing 4.8 shows all changes in this class. The new functions are built analogously to the already existing containers, hence providing the same accessibility in further procedures.

```java
public void addDefaultConstructQuery(Query q){ nestedConstructQueries.add(q) ;}
public void addAllDefaultConstructQuerys(Collection<Query> qs) { nestedConstructQueries.addAll(qs);}
public List<Query> getDefaultConstructQuerys() {return nestedConstructQueries ;}
public Iterator<Query> eachDefaultConstructQuery() {return nestedConstructQueries.iterator();}

public void addDefaultDescribeQuery(Query q){ nestedDescribeQueries.add(q) ;}
public void addAllDefaultDescribeQuerys(Collection<Query> qs) { nestedDescribeQueries.addAll(qs);}
public List<Query> getDefaultDescribeQuerys() {return nestedDescribeQueries ;}
public Iterator<Query> eachDefaultDescribeQuery() {return nestedDescribeQueries.iterator();}

public void addDefaultServiceConstructQuery(HashMap<String,Object> q){ nestedServiceConstructQueries.add(q) ;}
public void addAllDefaultServiceConstructQuerys(Collection<HashMap<String,Object>> qs) {  nestedServiceConstructQueries.addAll(qs);}
public List<HashMap<String,Object>> getDefaultServiceConstructQuerys() {return nestedServiceConstructQueries ;}
public Iterator<HashMap<String,Object>> eachDefaultServiceConstructQuery() {return nestedServiceConstructQueries.iterator();}

public void addDefaultServiceDescribeQuery(HashMap<String,Object> q){ nestedServiceDescribeQueries.add(q) ;}
public void addAllDefaultServiceDescribeQuerys(Collection<HashMap<String,Object>> qs) { nestedServiceDescribeQueries.addAll(qs);}
public List<HashMap<String,Object>> getDefaultServiceDescribeQuerys() {return nestedServiceDescribeQueries ;}
public Iterator<HashMap<String,Object>> eachDefaultServiceDescribeQuery() {return nestedServiceDescribeQueries.iterator();}
```

Listing 4.8: Extension of `DatasetDescription.java`.

4.2.3 Modification of DatasetUtils.java

In order to transform the information from the `DatasetDescription` into an actual internal `DatasetGraph`, the `DatasetUtils` class provides the relevant functions.

The `createDatasetGraph` function in Listing 4.9 is available with various signatures. They all unwrap their arguments in order to call the same function in the end. Having a `DatasetDescription` at hand, the appropriate function needs to also unwrap all the new information. Afterwards, the proximate functions have to be adjusted to the new parameters until the evaluating function `addInGraphsWorker`.

```java
public static DatasetGraph createDatasetGraph(DatasetDescription datasetDesc, String
    ↪ baseURI) {
    return createDatasetGraph(datasetDesc.getDefaultConstructQuerys(),datasetDesc.
        ↪ getDefaultDescribeQuerys(),datasetDesc.getDefaultServiceConstructQuerys(),
        ↪ datasetDesc.getDefaultServiceDescribeQuerys(), datasetDesc.
        ↪ getDefaultGraphURIs(), datasetDesc.getNamedGraphURIs(), baseURI) ;
}
```

Listing 4.9: Accepting extended `DatasetDescription` as parameter.

A new `createDatasetGraph` function with altered signature is required to pass the information to the `addInGraphs` function.

```java
public static DatasetGraph createDatasetGraph(List<Query> constQuerys, List<Query>
    ↪ descQuerys, List<HashMap<String,Object>> servConstQuerys, List<HashMap<String,
    ↪ Object>> servDescQuerys, List<String> uriList, List<String> namedSourceList,
    ↪ String baseURI) {
    DatasetGraph dsg = DatasetGraphFactory.createGeneral();
    addInGraphs(dsg, constQuerys, descQuerys, servConstQuerys, servDescQuerys, uriList,
        ↪ namedSourceList, baseURI);
    return dsg ;
}
```

Listing 4.10: Modification of `createDatasetGraph`.

Analogously to the previous existing functions a modified version of the `addInGraphs` function is added to call `addInGraphsWorker`.

```
public static void addInGraphs(DatasetGraph dsg, List<Query> constQuerys, List<Query>
    ↪ descQuerys, List<HashMap<String,Object>> servConstQuerys, List<HashMap<String,
    ↪ Object>> servDescQuerys, List<String> uriList, List<String> namedSourceList,
    ↪ String baseURI) {
    if ( ! dsg.supportsTransactions() )
        addInGraphsWorker(dsg, constQuerys, descQuerys, servConstQuerys, servDescQuerys,
            ↪ uriList, namedSourceList, baseURI) ;

    if ( dsg.isInTransaction() )
        addInGraphsWorker(dsg, constQuerys, descQuerys, servConstQuerys, servDescQuerys,
            ↪ uriList, namedSourceList, baseURI);

    Txn.executeWrite(dsg, ()->addInGraphsWorker(dsg, constQuerys, descQuerys,
        ↪ servConstQuerys, servDescQuerys, uriList, namedSourceList, baseURI)) ;
}
```

Listing 4.11: Modification of `addInGraphs`.

Finally, the function `addInGraphsWorker` processes all nested queries and merges their results together with the conventional data sources into the default graph of the query. For this purpose an appropriate `QueryExecution` object is constructed and executed. In the case of a remote query the `QueryEngineHTTP` is to be used. In the Listings 4.12 and 4.13 the local nested CONSTRUCTs and DESCRIBEs are processed. Listings 4.14 and 4.15 show the processing of remote nested CONSTRUCTs and DESCRIBEs.

```
if ( constQuerys != null && !constQuerys.isEmpty()) {
    for (Query q : constQuerys)
    {
        QueryExecution qexec = QueryExecutionFactory.create(q);
        Dataset constds = qexec.execConstructDataset();
        DatasetGraph cdsg = constds.asDatasetGraph();
        GraphUtil.addInto(dsg.getDefaultGraph(), cdsg.getDefaultGraph());
    }
}
```

Listing 4.12: Adding the nested CONSTRUCT results to the default graph.

```java
if ( descQuerys != null && !descQuerys.isEmpty()) {
    for (Query q : descQuerys)
    {
        QueryExecution qexec = QueryExecutionFactory.create(q);
        Model descmod = qexec.execDescribe();
        Dataset descds = DatasetFactory.create(descmod);
        DatasetGraph descg = descds.asDatasetGraph();
        GraphUtil.addInto(dsg.getDefaultGraph(), descg.getDefaultGraph());
    }
}
```

Listing 4.13: Adding the nested DESCRIBE results to the default graph.

```java
if ( servConstQuerys != null && !servConstQuerys.isEmpty()) {
    for (HashMap<String,Object> q : servConstQuerys)
    {
        QueryEngineHTTP remoteQE = (QueryEngineHTTP) QueryExecutionFactory.sparqlService
            ↪ ((String)q.get("Second"), (Query) q.get("First"));
        Dataset descds = remoteQE.execConstructDataset();
        DatasetGraph descg = descds.asDatasetGraph();
        GraphUtil.addInto(dsg.getDefaultGraph(), descg.getDefaultGraph());
    }
}
```

Listing 4.14: Adding the nested service-CONSTRUCT to the default graph.

```java
if ( servDescQuerys != null && !servDescQuerys.isEmpty()) {
    for (HashMap<String,Object> q : servDescQuerys)
    {
        QueryEngineHTTP remoteQE = (QueryEngineHTTP) QueryExecutionFactory.sparqlService
            ↪ ((String)q.get("Second"), (Query) q.get("First"));
        Model descmod = remoteQE.execDescribe();
        Dataset descds = DatasetFactory.create(descmod);
        DatasetGraph descg = descds.asDatasetGraph();
        GraphUtil.addInto(dsg.getDefaultGraph(), descg.getDefaultGraph());
    }
}
```

Listing 4.15: Adding the nested service-DESCRIBE to the default graph.

4.2.4 Modification of QueryEngineBase.java

Listing 4.16 shows the alteration of the `QueryEngineBase`. If the boolean `setReasoner` is `true`, the underlying model needs to be replaced with a Pellet ontology model. This is done by unwrapping the DatasetGraph to a model, replacing it by a `PelletReasoner` model and then rewrapping it to a DatasetGraph. In order to use the Pellet model the `openllet` needs to be imported.

```java
import openllet.jena.*;

protected QueryEngineBase(Query query, DatasetGraph dsg, Binding input, Context cxt) {
    this(dsg, input, cxt) ;
    this.query = query ;
    query.setResultVars() ;
    // Unoptimized so far.
    setOp(createOp(query)) ;

    DatasetGraph dsgtmp = prepareDataset(dsg, query);
    if(query.isReasoner()){
        OntModel om = ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC,
            ↪ DatasetFactory.wrap(dsgtmp).getDefaultModel());
        Dataset tmpds = DatasetFactory.wrap(om);
        dataset = tmpds.asDatasetGraph();
    } else{
        dataset = dsgtmp;
    }
}
```

Listing 4.16: Replacing the standard Model with a Pellet model.

Chapter 5

Analysis

After the implementation of nested CONSTRUCTs in the previous chapter 4, this chapter deals with their application. Section 5.1 will present some of the possible new features. A common theoretical concern about nested CONSTRUCTs will be examined in Section 5.2.

5.1 Analysis of Use Cases

Nested CONSTRUCTs can be used to create new values, write queries in a different way, integrate data from different sources and reduce the dataset down to a necessary minimum. The following subsections deal with these options in detail.

5.1.1 Creation of New Values

The first fundamental feature of nested CONSTRUCT queries is the creation of values that are not present in the KB. These new facts are crucial in some applications, as it will be shown in Chapter 6. In other situations they provide a different, more convenient way to handle the data similar to views in SQL [2]. An example for this is constructed with the Mondial KB (Chapter 2) in Listing 5.1. In order to process the average GDP per person of every country, an inner query calculates the values and adds them to the dataset. Using the results, the outer query can simply address this data like all other data in the KB.

```
...
FROM {
        CONSTRUCT {?C :gdpAverage ?GDPavg}
        FROM <file:mondial.rdf>
        WHERE   {
                 ?C a :Country.
                 ?C :gdpTotal ?GDPtot.
                 ?C :population ?Pop.
                 BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
                }
    }
WHERE { ...
```

Listing 5.1: Nested CONSTRUCT of average GDP per person.

5.1.2 Query Writing

Writing a query is an application-dependent task. One might use an explorative approach of examining the data and therefore reuse the results of the previous query in following queries. Furthermore, optimisation or a distributed evaluation of queries can be a driving factor in composing a query. Nested CONSTRUCTs enable the possibility to push information from the WHERE to the FROM clause. This can lead to an optimisation, e.g. by pushing down FILTER expressions as far as possible. [2]

Reuse of Queries

The reuse of existing CONSTRUCT/DESCRIBE queries have several benefits. For example, the workflow within data exploration becomes much easier. An already existing and verified query can simply be executed to generate relevant subgraphs for further exploration. Contrary to simply saving the subgraph and starting a new query on this new file, a nested CONSTRUCT can be used to calculate that subgraph just in time. Even though this approach recalculates the new graph every time, it also guarantees the actuality of the data and prevents endless copies. In combination with a remote access point the actuality of the data is especially crucial, because one does not have control over the data. As an example, the DESCRIBE query of rich countries in Listing 5.2 is reused in Listing 5.3.

The query below defines a country as rich if their average GDP per person is above 50000. It then returns a RDF graph consisting of all subgraphs spanned by each individual country, hence describing each country by all found information.

```
DESCRIBE ?C
WHERE    {?C a :Country.
          ?C :gdpTotal ?GDPtot.
          ?C :population ?Pop.
          BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
          FILTER (?GDPavg > 50000)}
}
```

Listing 5.2: DESCRIBE of rich countries.

To further explore this freshly generated graph of rich countries, it can be queried for other information. An example query could ask for respective names of these countries and, if available, the economic sector distribution. This is done in the query Listing 5.3 below. The green area is the reused query.

```
PREFIX : <http://www.semwebtech.org/mondial/10/meta#>
BASE <http://www.semwebtech.org/mondial/10/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?Country ?Agri ?Ind ?Serv
FROM     {
        SERVICE <http://www.semwebtech.org/mondial/10/sparql>
        DESCRIBE ?C
        WHERE    {?C a :Country.
                  ?C :gdpTotal ?GDPtot.
                  ?C :population ?Pop.
                  BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
                  FILTER (?GDPavg > 50000)}

        }

WHERE { ?C a :Country.
```

```
        ?C :name ?Country.
        OPTIONAL {?C :gdpInd ?Ind}.
        OPTIONAL {?C :gdpAgri ?Agri}.
        OPTIONAL {?C :gdpServ ?Serv}}
```

Listing 5.3: Reuse of rich countries.

The results of this query are presented in Table 5.1. The "Falkland Islands" do not have available data for the industrial and service sector, hence only appearing in the results because of the OPTIONAL keyword. All numbers represent the corresponding share of this sector in the overall GDP. This example shows the easy intuition of nested CONSTRUCTs for data exploration.

?Country	**?Agri**	**?Ind**	**?Serv**
"Falkland Islands"	95	NA	NA
"Qatar"	0.1	72.2	27.7
"San Marino"	0.1	39.2	60.7
"Liechtenstein"	8	37	55
"Switzerland"	0.7	26.8	72.5
"Canada"	1.7	28.4	69.9
"Monaco"	0	10	90
"Australia"	3.8	27.4	68.7
"Norway"	1.2	42.3	56.5
"Denmark"	1.5	21.7	76.8
"Andorra"	14	79	6
"Sweden"	2	31.3	66.8
"Bermuda"	0.7	5.7	93.5
"United States"	1.1	19.5	79.4
"Singapore"	0	29.4	70.6
"Jersey"	2	2	96
"Kuwait"	0.3	50.6	49.1
"Luxembourg"	0.3	13.3	86.4
"Macao"	0	6.5	93.5

Table 5.1: Solutions to query 5.3.

Distributed evaluation

With nested CONSTRUCT queries, it is easy to merge query results from two different endpoints for efficient data integration. It is possible to let the endpoints evaluate the queries and only do further processing on the smaller resulting graphs. Additionally, these resulting graphs can be shaped into the own conventional graph patterns. This might be a big advantage if an application has a pre-existing outer query and is supposed to add in another data source. Case examples for this are online portals gathering information from multiple webpages. The scenario in Listing 5.4 assumes that two partial versions of the Mondial database exist. One provides the population of countries, the other one provides the total GDP. In order to calculate the average GDP for each country, one needs to retrieve triples for population as well as the total GDP from the respective database. Afterwards, all the required data is in the active graph of the outer query and can be used for returning the average GDP per person.

```
PREFIX mon: <http://www.semwebtech.org/mondial/10/meta#>
BASE <http://www.semwebtech.org/mondial/10/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?Country ?GDPavg
FROM    {
         SERVICE <http://www.semwebtech.org/mondial/10/sparql>
         CONSTRUCT {?C mon:name ?Name. ?C mon:gdpTotal ?GDPtot}
         WHERE   {
                 ?C a mon:Country.
                 ?C mon:gdpTotal ?GDPtot.
                 ?C mon:name ?Name
                }
        }
FROM    {
         SERVICE <http://localhost:8080/mondial-lod/sparql>
         CONSTRUCT {?C mon:name ?Name. ?C mon:population ?Pop}
         WHERE   {
                 ?C a mon:Country.
                 ?C mon:population ?Pop.
                 ?C mon:name ?Name
                }
        }
```

```
WHERE    {
          ?C mon:name ?Country.
          ?C mon:population ?Pop.
          ?C mon:gdpTotal ?GDPtot
          BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
         }
```

Listing 5.4: Distributed query for average GDP per person.

Optimisation Considerations

Usually there are different measures to be considered for describing the optimisation of a query. First of all, there is the usual computational complexity of queries described in "Complexity of SPARQL" [21]. Since there is nothing changed in this area, the focus will be set on the exchange of information and needed storage space. Therefore, the amount of triples sent between two or more parties is considered in order to solve a given problem.

The execution time of remote queries is omitted, because it is not a very reliable measurement. This is due to the extreme dependency on external changing variables, e.g. the internet connection or the number of concurrent queries on the server. Additionally to these influences, there is no defined standard for the answer format of SPARQL endpoints. Some of them refuse to answer complex queries due to a fixed maximum calculation time [5] or do not handle special queries, like DESCRIBE, at all. Nonetheless, nested CONSTRUCTs enable the rewriting of a query and therefore allow theoretical optimisation possibilities. Simple changes in the inner and outer query have significant influences on the evaluation process. This can be used to adapt queries to specific needs without changing the results and hence optimising it.

The following example queries show the impact of simply pushing parts of the graph pattern from the outer to the inner query. Therefore, a constructed scenario shall examine the type of government in countries with an average GDP per person above 50000. The relevant part of this query is the FILTER on countries with a GDP per person of more than 50000. Governmental data will only serve as a representative of all other information in the graph that is directly connected to the country. The nested CONSTRUCT query is always marked with a green area.

The first query simply requests the whole graph from the remote endpoint and does all the evaluation on the local machine.

```
PREFIX mon: <http://www.semwebtech.org/mondial/10/meta#>
BASE <http://www.semwebtech.org/mondial/10/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?Country ?GDPavg ?Gov
FROM    {
         SERVICE <http://www.semwebtech.org/mondial/10/sparql>
         CONSTRUCT {?C ?P ?O}
         WHERE   {
                 ?C ?P ?O
                }
        }
WHERE   {
         ?C a mon:Country.
         ?C mon:name ?Country.
         ?C mon:gdpTotal ?GDPtot.
         ?C mon:population ?Pop.
         BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
         FILTER (?GDPavg > 50000).
         ?C mon:government ?Gov
        }
```

Listing 5.5: Get all triples and FILTER afterwards.

The nested CONSTRUCT is resulting in 194551 triples to be sent from the endpoint to the user. As a small shift towards more filtering in the inner query, only country data shall be retrieved from the endpoint. This limits the data to be sent and lowers the calculation to be done on the local machine.

```
PREFIX mon: <http://www.semwebtech.org/mondial/10/meta#>
BASE <http://www.semwebtech.org/mondial/10/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?Country ?GDPavg ?Gov
FROM     {
         SERVICE <http://www.semwebtech.org/mondial/10/sparql>
         CONSTRUCT {?C ?P ?O}
         WHERE   {
                 ?C a mon:Country.
                 ?C ?P ?O
                 }
        }
WHERE    {
         ?C mon:name ?Country.
         ?C mon:gdpTotal ?GDPtot.
         ?C mon:population ?Pop.
         BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
         FILTER (?GDPavg > 50000).
         ?C mon:government ?Gov
        }
```

Listing 5.6: Get only the triples from country nodes.

For this request, the nested query was only returning 22016 triples to the user. Finally, only the required data from suitable countries, fitting the conditions, are requested from the endpoint. This further lowers the amount of sent triples to a minimum for this task. In order to query the relevant properties, they are simply added as a template to the nested CONSTRUCT query and accessed in the outer query. Therefore, the inner query in Listing 5.7 sends only 57 triples to the user.

```
PREFIX mon: <http://www.semwebtech.org/mondial/10/meta#>
BASE <http://www.semwebtech.org/mondial/10/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?Country ?GDPP ?Gov
FROM    {
         SERVICE <http://www.semwebtech.org/mondial/10/sparql>
         CONSTRUCT {?C mon:name ?Name. ?C mon:government ?Gov .
                    ?C mon:gdpAverage ?GDPavg}
         WHERE   {
                 ?C a mon:Country.
                 ?C mon:gdpTotal ?GDPtot.
                 ?C mon:population ?Pop.
                 ?C mon:name ?Name.
                 ?C mon:government ?Gov.
                 BIND( (?GDPtot*1000000) / ?Pop AS ?GDPavg).
                 FILTER (?GDPavg > 50000).
                }
        }
WHERE   {
         ?C mon:name ?Country.
         ?C mon:gdpAverage ?GDPP.
         ?C mon:government ?Gov
        }
```

Listing 5.7: Get only triples from countries fitting the condition.

5.2 Composability of CONSTRUCT Queries

One objective of query languages, like SPARQL, is to be as simple as possible. This means that features that can be expressed by other features of the language, shall not be included in the language [2]. Nested CONSTRUCTs have not already been added to the language, because an unproven conjecture states, that they are expressible by SELECT subqueries [3]. In particular, this means that for an outer query Q_1 and an inner nested CONSTRUCT query Q_2 with the dataset D, there exists a query Q in the language, such that:

$$[\![Q_2]\!]_{[\![Q_1]\!]_D} = [\![Q]\!]_D$$

In the paper [3] a rewriting method for this problem is proposed. Even though a rewriting method exists, it is non-intuitive, technical, and has an exponential blowup in the depth of nestings. Furthermore, the possibility of reasoning in the nested CONSTRUCT is not considered in this paper. It is shown in Chapter 6 that nested CONSTRUCTs can be a very useful feature in combination with reasoning. All in all, the authors call out nested CONSTRUCTs as a “desired feature” [3].

Chapter 6

Closing the Open World

The previous chapter has shown the possibilities of nested CONSTRUCTs in altering the ABox of a KB. Since a TBox is also a RDF graph, nested CONSTRUCTs can also be used to alter the terminological and schema knowledge of the KB. This extended or manipulated knowledge can then be used for upcoming reasoning tasks. Furthermore, the possibility of using a Pellet model in a nested CONSTRUCT enables freshly derived data in the ABox as well as the TBox for further processing. Hence, this feature provides the opportunity of querying data, that can not be derived by a single round of reasoning or is dependent on reasoned knowledge beforehand. Since this process is locally tackling the properties of the OWA, it will be referred to as "Closing the Open World".

6.1 Win-Move-Game

Mondial is too big for the reasoner and thus for the "Closing the Open World" scenario. That is why the Win-Move-Game is considered below. This game is commonly used in CWA settings for its complex logical semantics. The lecture slides [1] introduce this game as a classical problem case in the open world setting, because it requires an appropriate closure. While former solution approaches needed to wrap multiple queries with Java, this thesis will present the use of nested CONSTRUCTs in order to provide a solution in pure SPARQL.

The rules of the two player game are very simple. Considering a graph with directed edges; a move is a transition from one node to another via a chosen edge. The game ends if one player can not make any further moves and therefore loses. In the Figure 6.1, a graph representation of an instance of this game is given [1]. Listing 6.1 gives the corresponding graph description in RDF.

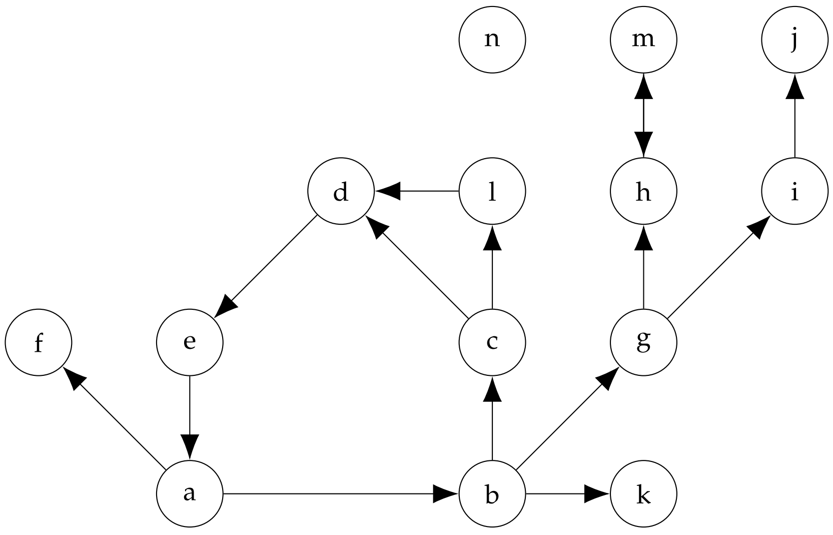

Figure 6.1: Graph representation of a Win-Move-Game example [1].

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.

:Node a owl:Class; owl:equivalentClass
  [ a owl:Class; owl:oneOf (:a :b :c :d :e :f :g :h :i :j :k :l :m :n)
     ↪ ].
:edge a owl:ObjectProperty; rdfs:domain :Node; rdfs:range :Node.
:a a :Node; :edge :b, :f.
:b a :Node; :edge :c, :g, :k.
:c a :Node; :edge :d, :l.
:d a :Node; :edge :e.
:e a :Node; :edge :a.
:f a :Node.
:g a :Node; :edge :i, :h.
:h a :Node; :edge :m.
:i a :Node; :edge :j.
:j a :Node.
:k a :Node.
:l a :Node; :edge :d.
:m a :Node; :edge :h.
:n a :Node.
```

Listing 6.1: RDF representation of the Win-Move-Game instance.

By simply looking at the graph it becomes obvious that certain nodes are always losing and others are always winning. Standing on the node "j" will always yield a loss, since there is no further edge to be chosen. The node "i" on the other hand will always guarantee a win, because you can only choose one edge which will put the enemy player on a losing node. Furthermore, a constellation of the nodes "m" and "h" clearly forces an endless loop and is therefore a drawn state. In fact, there are two general rules for determining if a node is in a winning or a losing state. All remaining nodes are drawing nodes by logical exclusion. Winning nodes can be described as the possibility to put the enemy on a losing node. Assuming perfect players, this simply correlates with the existence of an edge to a losing node. A losing node has only edges that lead to a winning state for the enemy. In other words, for all outgoing transitions the next node will be a winning node. These rules can be described with OWL DL and are presented in Listing 6.2.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.

:WinNode a owl:Class; owl:intersectionOf ( :Node
  [a owl:Restriction; owl:onProperty :edge; owl:someValuesFrom :
     ↪ LoseNode]).
:LoseNode a owl:Class; owl:intersectionOf ( :Node
  [a owl:Restriction; owl:onProperty :edge; owl:allValuesFrom :WinNode
     ↪ ]).
```

Listing 6.2: OWL description of WinNodes and LoseNodes.

With this description two problems are arising. On the one hand, drawing nodes can only be deduced by knowing all winning and losing nodes. On the other hand and far more important, the OWA makes the reasoning over these rules problematic. There might be further edges that are not present in the current KB. As an example, node "j" could have a non-prominent edge, which leads to an unknown node; hence, making the losing node a winning node. To tackle this problem, the set of edges in the KB is assumed to be complete, effectively making this property closed. In order to extend the ABox and TBox appropriately, the current number of outgoing edges is to be counted for each node. For expressing this in OWL, subclasses of the class node need to be added to the KB which have this number as a restriction on the cardinality of their edge property. In the end, each node needs to be assigned to its corresponding subclass. For example, the node "j" belongs to the class of `0edgeNode` and node "a" belongs to `2edgeNode`. Listing 6.3 shows a corresponding CONSTRUCT query that builds this closure.

```
CONSTRUCT { ?newclass a owl:Class; rdfs:subClassOf <foo://bla#Node>;
   ↪ owl:equivalentClass [ a owl:Restriction ; owl:onProperty <foo://
   ↪ bla#edge>; owl:cardinality ?num] . ?O a ?newclass }
FROM <file:winmove-axioms.n3>
FROM <file:winmove-ex-graph.n3>
WHERE {
        {
                SELECT ?O (count(distinct ?X) AS ?num)
                WHERE { ?O a <foo://bla#Node> .
                        OPTIONAL { ?O  <foo://bla#edge> ?X}}
                GROUP BY ?O ?N
        } .
        BIND (URI(concat('foo://bla#', str(?num), 'edgeNode')) AS ?
           ↪ newclass) }
```

Listing 6.3: Constructing a Win-Move-Closure with a CONSTRUCT query.

Having this CONSTRUCT query at hand, it can be used to extend the current KB, such that a reasoner can determine winning, losing and drawn nodes. For this purpose, a new query with REASONER support is set up. This query uses the above query as a nested CONSTRUCT and extends the KB further by adding a draw node property to every node that cannot be classified as either a winning or losing node. Afterwards, the KB contains all information to classify each node as either a winning, losing or drawn node. In order to query this information, another wrapping SELECT query is used with REASONER support, because it has to deduce the winning and losing nodes again. The full query can be seen in Listing 6.4.

```
PREFIX : <foo://bla#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT REASONER ?N ?NT
FROM {
        CONSTRUCT REASONER { ?N a :DrawNode . ?S a ?NT }
        FROM {
                CONSTRUCT { ?newclass a owl:Class; rdfs:subClassOf <foo
                   ↪ ://bla#Node>; owl:equivalentClass [ a owl:
                   ↪ Restriction ; owl:onProperty <foo://bla#edge>;
                   ↪ owl:cardinality ?num] . ?O a ?newclass . ?Sub ?
                   ↪ Pre ?Obj }
                FROM <file:winmove-axioms.n3>
                FROM <file:winmove-ex-graph.n3>
                WHERE { { ?Sub ?Pre ?Obj }
                        UNION
                        {
                                SELECT ?O (count(distinct ?X) AS ?num)
                                WHERE { ?O a <foo://bla#Node> .
                                        OPTIONAL { ?O  <foo://bla#edge>
                                           ↪  ?X}}
                                GROUP BY ?O ?N
                        } .
                        BIND (URI(concat('foo://bla#', str(?num), '
                           ↪ edgeNode')) AS ?newclass) }
                }
        WHERE { {?N a :Node .
                FILTER NOT EXISTS { ?N a :WinNode } .
                FILTER NOT EXISTS { ?N a :LoseNode } .}
                UNION
                { ?S a ?NT} }
     }
WHERE { ?N a :Node ; a ?NT }
ORDER BY ?N
```

Listing 6.4: SPARQL query to close and evaluate the Win-Move-Game in one query.

The solution of this query is intuitively represented by the coloured graph in Figure 6.2. Red nodes stand for losing nodes, green nodes are winning nodes and yellow ones represent drawn nodes.

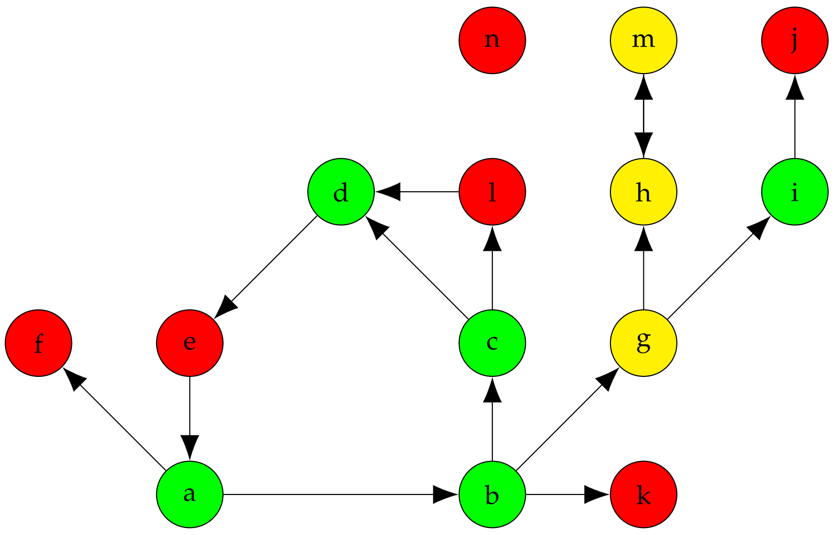

Figure 6.2: Graph representation of the Win-Move-Game solution.

Chapter 7

Conclusion

This thesis was mainly concerned with a functioning implementation of the nested CONSTRUCT feature in SPARQL. The new feature is functional and did not show any shortcomings in the tested scenarios and yields a nice practical addition to the language. Nested CONSTRUCTs are especially suitable for data integration purposes, because they provide non materialised views and simplify the reshaping of remote data. Furthermore, they enable the usage of the latest data and distribute the main parts of the calculation to the remote endpoints. They are able to construct a closure in case of insufficient information. Therefore, nested CONSTRUCTs are a necessity for reasoning under the CWA with SPARQL.

On the downside, these features are only implemented in the testing environment of the Semwebjar tool. Therefore, the practical use, outside of teaching and experimenting, is limited to endpoints providing the necessary infrastructure. This infrastructure includes the possibility of handling and returning the inevitable query types CONSTRUCT and DESCRIBE.

All in all, the thesis is a nice starting point for further experiments with nested CONSTRUCT queries. In addition, the work process can be used as a guideline to add possible new features in a similar fashion.

7.1 Future Work

Potential follow up work comprises mainly some theoretical extra considerations. One topic could be the analysis of correlated variables, namely the sharing of a variable between inner and outer query. Furthermore, the grammar could be extended to load nested CONSTRUCTs from a file; hence keeping the complete query structured and as minimal as possible. Moreover, the paper [3] proposes the possibility of updating KBs with nested CONSTRUCTs, in case of their availability.

Bibliography

[1] W. May, “Semantic Web Lecture”. `https://www.dbis.informatik.uni-goettingen.de/Teaching/SemWeb-WS1819/`, 2018/2019. [Online; accessed 21.10.2019].

[2] R. Angles and C. Gutierrez, “Subqueries in SPARQL”, *CEUR Workshop Proceedings*, vol. 749, 01 2011.

[3] A. Polleres, J. Reutter, and E. V. Kostylev, “Nested constructs vs. sub-selects in SPARQL”, *Alberto Mendelzon International Workshop on Foundations of Data Management*, vol. 10, 2016.

[4] T. Berners-Lee, “Linked Data”. `https://www.w3.org/DesignIssues/LinkedData.html`, 2006. [Online; accessed 21.10.2019].

[5] M. Heinemann, “Linked Open Data and its Evaluation”. `https://www.dbis.informatik.uni-goettingen.de/Teaching/Theses/theses-list.html`, 2019. [Masterthesis].

[6] W. May, “Introduction to Databases”. `https://www.dbis.informatik.uni-goettingen.de/Teaching/DB-WS1819/`, 2018/2019. [Online; accessed 21.10.2019].

[7] S. Harris and A. Seaborne, “SPARQL 1.1 query language”, W3C recommendation, W3C, Mar. 2013. `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`.

[8] R. Cyganiak, D. Wood, and M. Lanthaler, “RDF 1.1 concepts and abstract syntax”, W3C recommendation, W3C, Feb. 2014. `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`.

[9] P. Hitzler, M. Krötzsch, and S. Rudolph, *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 1st ed., 2009.

[10] T. Berners-Lee and D. Connolly, “Notation3 (N3): A readable RDF syntax”. `https://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/`, 2011. [Online; accessed 21.10.2019].

[11] The Apache Software Foundation, “Apache Jena”. `https://jena.apache.org/index.html`, 2019. [Online; accessed 21.10.2019].

[12] The Apache Software Foundation, "ARQ". `https://jena.apache.org/documentation/query/index.html`, 2019. [Online; accessed 21.10.2019].

[13] The Apache Software Foundation, "Apache Jena Releases". `https://jena.apache.org/download/index.cgi`, 2019. [Online; accessed 21.10.2019].

[14] The Apache Software Foundation, "ARQ - Application API". `https://jena.apache.org/documentation/query/app_api.html`, 2019. [Online; accessed 21.10.2019].

[15] W. May, "Databases and Information Systems". `https://www.dbis.informatik.uni-goettingen.de/`, 2019. [Online; accessed 21.10.2019].

[16] The Apache Software Foundation, "Apache Tomcat". `https://tomcat.apache.org/`, 2019. [Online; accessed 21.10.2019].

[17] W. May, "Playground page for the XML Course". `http://www.stud.informatik.uni-goettingen.de/xml-lecture/#tomcat`, 2019. [Online; accessed 21.10.2019].

[18] S. Siemer, "Exploring the Apache Jena Framework", 2019. Practical preliminary work of this thesis.

[19] J. Team, "JavaCC - The Java Parser Generator". `https://javacc.org/`, 2019. [Online; accessed 21.10.2019].

[20] W. May, "Information extraction and integration with FLORID: The MONDIAL case study", Tech. Rep. 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`.

[21] J. Perez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL", 2006.