Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik

Query Optimization Techniques For Scaling Up To Data Variety

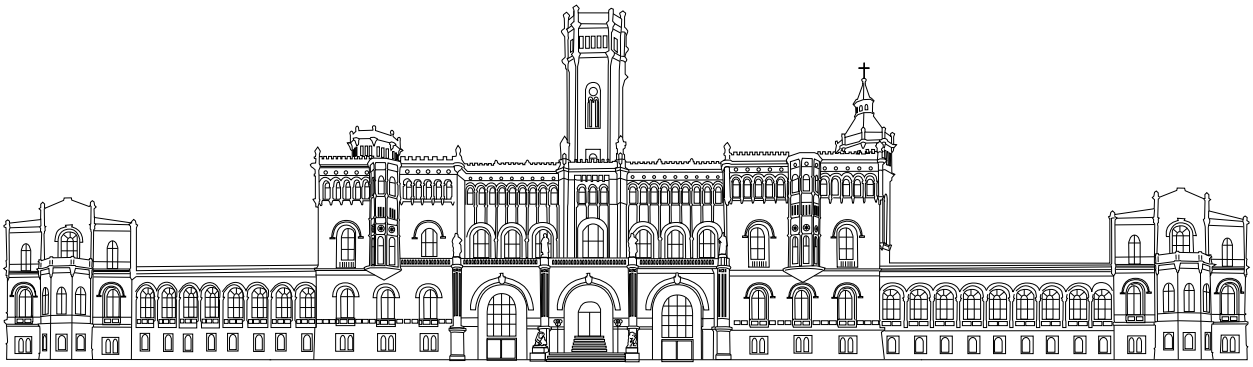
A thesis submitted in fulfillment of the requirements for the degree of
Master of Science in Computer Science

BY

Philipp Daniel Rohde
Matriculation number: 2886190
E-mail: philipp.rohde@stud.uni-hannover.de

First evaluator: Prof. Dr. Sören Auer
Second evaluator: Prof. Dr. Maria-Esther Vidal
Supervisor: M.Sc. Kemele M. Endris

July 4, 2019



Declaration of Authorship

I, Philipp Daniel Rohde, declare that this thesis titled, 'Query Optimization Techniques For Scaling Up To Data Variety' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Philipp Daniel Rohde

Signature: _____

Date: _____

"Information learned is more valuable than information given."

— Al Mualim, video game *Assassin's Creed* (2007)

Acknowledgements

First, I would like to thank Prof. Dr. Sören Auer and Prof. Dr. Maria-Esther Vidal for giving me the opportunity to develop my thesis at TIB. I am very grateful for all the guidance, immense knowledge, motivation, and support from Prof. Dr. Maria-Esther Vidal. Without all her help, the development of this thesis would have been impossible.

I would like to show my gratitude towards my supervisor, Kemele M. Endris, whose support and patience were crucial for the successful implementation of this thesis.

I would also like to thank my fellow scientists from the *Scientific Data Management Group*, especially Samaneh Jozashoori and Farah Karim, for all the discussions, proposals, and support on scientific working. I really enjoy being a member of the team.

Thanks to my friends for being patient with me during the development of my thesis. Special thanks to my girlfriend, Fiona Eitner, for her invaluable support and efforts to relax me. I am glad she helped me keeping a good work-life balance.

Finally, I want to give notice that this thesis extends my work already published in the paper 'Ontario: Federated Query Processing against a Semantic Data Lake' [12] at DEXA 2019.

Philipp D. Rohde

Abstract

Even though Data Lakes are efficient in terms of data storage, they increase the complexity of query processing; this can lead to expensive query execution. Hence, novel techniques for generating query execution plans are demanded. Those techniques have to be able to exploit the main characteristics of Data Lakes. Ontario is a federated query engine capable of processing queries over heterogeneous data sources. Ontario uses source descriptions based on RDF Molecule Templates, i.e., an abstract description of the properties belonging to the entities in the unified schema of the data in the Data Lake. This thesis proposes new heuristics tailored to the problem of query processing over heterogeneous data sources including heuristics specifically designed for certain data models. The proposed heuristics are integrated into the Ontario query optimizer. Ontario is compared to state-of-the-art RDF query engines in order to study the overhead introduced by considering heterogeneity during query processing. The results of the empirical evaluation suggest that there is no significant overhead when considering heterogeneity. Furthermore, the baseline version of Ontario is compared to two different sets of additional heuristics, i.e., heuristics specifically designed for certain data models and heuristics that do not consider the data model. The analysis of the obtained experimental results shows that source-specific heuristics are able to improve query performance. Ontario optimization techniques are able to generate effective and efficient query plans that can be executed over heterogeneous data sources in a Data Lake.

Keywords: Federated Query Processing, Federated Query Engine, Semantic Data Lake, Heterogeneous Data Sources, RDF Molecule Template

Contents

List of Figures

List of Tables

Acronyms

GaV Global-as-View

GLaV Global-Local-as-View

LaV Local-as-View

LSLOD Life Science Linked Open Data

QEP Query Execution Plan

RDF Resource Description Framework

RDF-MT RDF Molecule Template

RDFS RDF Schema

SDL Semantic Data Lake

SSQ star-shaped sub-query

URI Universal Resource Identifier

Chapter 1

Introduction

In recent years, an enormous amount of heterogeneous data has become available through various platforms. The need for efficient techniques to manage and query big and heterogeneous data has gained attention. The variety and volume of data integrated from different data sources have to be handled efficiently and effectively by Big Data systems. The Semantic Web community invested considerable efforts in transforming tabular data into linked data and interlinking these data sets with existing data sets in the *Linked Open Data cloud* [40], e.g., Linked Open Drug Data (LODD) [39], and Bio2RDF [5]. However, lifting tabular data to linked data is costly and assumes a stable schema and data model. *Data Lakes* have been proposed to provide a scalable and flexible knowledge discovery, analysis, and reporting. Data Lakes are composed of heterogeneous data sources in their original data format. Managing a Data Lake reduces the costs of identifying, storing, cleansing, and integrating data substantially and promotes flexibility in data analysis. Nevertheless, Data Lakes introduce complexity during query processing. Contrary to existing federated query engines, federated query processing over Data Lakes demands the integration and semantic description of data to be collected from heterogeneous sources. Thus, selecting relevant sources for a specific query, creating an efficient query execution plan considering the data source types, and combining partial results retrieved from these sources are the main challenges in query processing over Data Lakes.

This thesis addresses the problem of federated query processing over *Semantic Data Lakes* and proposes heuristics to be considered by Ontario, a query engine able to efficiently operate on heterogeneous data sources. Ontario implements novel query processing methods for source selection, query decomposition, and query planing that are capable of exploiting knowledge about the sources and the query to generate plans over a Semantic Data Lake. Ontario uses RDF Molecule Templates [13], i.e.,

```
SELECT DISTINCT ?drug ?disName ?drugformula ?sename
WHERE {
    t1  ?drug     dailymed:activeIngredient          dailymed:Simvastatin .
    t2  ?drug     dailymed:genericDrug               ?dbdrug .
    t3  ?drug     dailymed:possibleDiseaseTarget     ?disease .
    t4  ?drug     owl:sameAs                         ?sadrug .
    t5  ?disease  rdfs:label                         ?disName
    t6  ?sadrug   sider:sideEffect                   ?seffect .
    t7  ?seffect  sider:sideEffectName               ?sename .
    t8  ?dbdrug   drugbank:chemicalFormula           ?drugformula
}
```

(a) SPARQL Query: Find targets, and side effects of drugs with active ingredient Simvastatin.

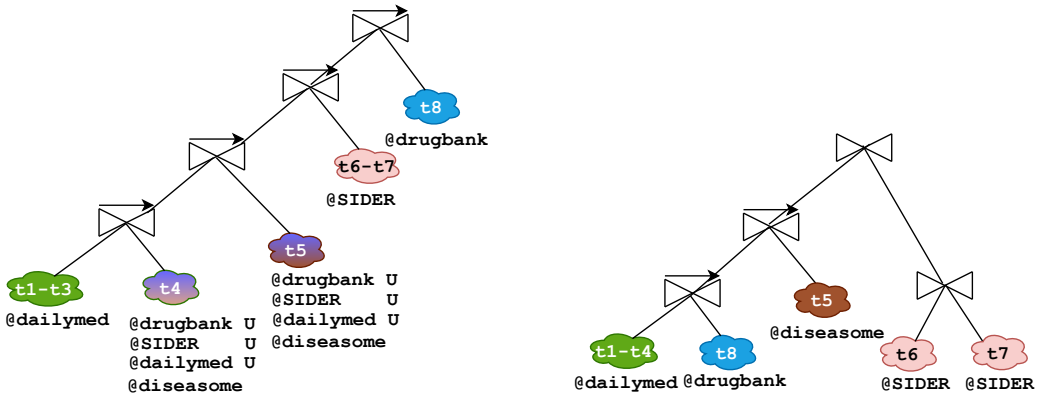
(b) Data Sources in a Data Lake

Figure 1.1: **Motivating Example**. (a) A SPARQL query composed of four star-shaped sub-queries accessing four data sources, Dailymed, Diseasome, SIDER, and DrugBank. (b) Data Sources: Dailymed (RDF in Virtuoso), Diseasome (Local JSON File), SIDER (TSV in HDFS), DrugBank (XML in MySQL) (taken from [12])

abstract descriptions of the properties of entities in an RDF data set in order to identify star-shaped sub-queries of an input query. In contrast to state-of-the-art approaches, Ontario classifies sub-queries according to their type of instantiations and joins. Additionally, star-shaped sub-queries are annotated with the data engines where they will be executed. Ontario exploits this knowledge to generate efficient query execution plans. The performance of Ontario is studied over the LSLOD benchmark [21] and compared to existing federated query engines, i.e., FedX [41], ANAPSID [2], and MULDER [13], and different configurations of the data sources, i.e., only RDF data sets, only RDB data sets, and a Semantic Data Lake composed of RDF and RDB data sets. Next, the work of this thesis is prompted with the motivating example from the preceding paper [12].

1.1 Motivating Example

The motivating example and its figures are taken from the paper [12] which includes my preceding work on the topic. In the biomedical domain, complex questions frequently need to be answered with multiple data sources of different data models. Especially in this domain, flexible data management and integration techniques are required due to the variety of tools and formats data is collected, generated, and processed with. To provide a unified view over these heterogeneous data sources, mapping rules are utilized to describe the required transformations from raw data



(a) FedX Query Plan (b) MULDER Query Plan

Figure 1.2: **Motivating Example: Query Plans**. (a) FedX creates a left-linear plan and uses nested loop joins (arrows on top of join) (b) MULDER identifies a bushy-tree for star-shaped groups. (taken from [12])

into the unified schema. These mappings enable the translation of queries from the unified schema into queries against the sources using native access interfaces.

This thesis is motivated by comparing the performance of federated SPARQL query engines over a federation of data sources that provide a SPARQL-based access interface. For instance, the SPARQL query in Figure 1.1a requires to collect the name of possible drug targets, chemical formula, and side effects of drugs labeled by the FDA that have the active substance Simvastatin. To answer this query, four data sources of different data formats (Figure 1.1b) need to be accessed. Dailymed publishes FDA label information about marketed drugs in the United States; Diseasome makes available a network of disorders and disease genes; DrugBank reports information about drugs and drug targets, and SIDER presents information on drug side effects. The example query comprises eight triple patterns that are identified with $t1$ to $t8$ in Figure 1.1a. The triple patterns $t1$ - $t4$ can be answered by Dailymed, while Diseasome is able to answer triple pattern $t5$. Further, SIDER can answer $t6$ and $t7$, $t8$ is answered by DrugBank. The data access services for each data set is implemented by different back-ends and provide varying capabilities. For instance, the endpoint service for SIDER and Diseasome are Spark-based query processors that translate queries from SPARQL to SQL, where the raw data need to be loaded in memory to evaluate the query in these data sources. Similarly, the endpoint for DrugBank translates SPARQL to SQL and executes the translated query in MySQL, which provides efficient indexing and query optimization for relational data.

Federated query engines, FedX [41] and MULDER [13], provide a unified view over a set of data sources that respect SPARQL protocol. They rely on source descriptions to select relevant sources for the given query and finding an efficient query execution plan. For instance, FedX contacts the data sources to decide where each triple pattern will be executed, while MULDER requires RDF Molecule Templates (RDF-MTs) to be collected in advance. FedX decomposes the example query into five sub-queries; $t1$ - $t3$, $t6$ - $t7$, and $t8$ that are sent to Dailymed, SIDER, and DrugBank, respectively, and $t4$ and $t5$ sent to all four data sources. FedX creates a left linear tree plan with nested loop joins, an operator that pushes the join operation down to the data sources by binding the join variables of the right operand with values extracted from the left operand, as shown in Figure 1.2a. The FedX planner assumes the underlying data model is RDF and triples are materialized in a triple store that is optimized for this data model. However, since the data sources have different data models and capabilities, pushing down join operations to the sources results in higher execution time, *20 minutes*, and incomplete results. To the contrary, MULDER decomposes the query into five sub-queries; $t1$ - $t4$, $t8$, and $t5$ are sent to Dailymed, DrugBank, and Diseasome, respectively, while $t6$ and $t7$ are executed in SIDER. MULDER creates a bushy tree plan with nested loop join and GJoin [2] operators (Figure 1.2b). Based on the selectivity of the operands the type of operator is decided. Like FedX, MULDER assumes RDF as the underlying data model and uniform querying capabilities of the data sources in the federation. Based on these assumptions, MULDER selects a nested loop join for the first two joins, between sub-queries $t1$ - $t4$, $t8$, and $t5$. Despite, MULDER creates an efficient bushy tree plan that helps parallelizing the query execution. The selection of the join operator ignores the data source capabilities and underlying data model, which results in a high execution time, *4.6 minutes*. This thesis devises optimization techniques guided by heuristics that enable the creation of source-dependent query plans in order to reduce query execution time. First, the Ontario query optimizer resorts to data source descriptions based on RDF Molecule Templates to select the relevant sources for the query. Then, the query is decomposed into star-shaped sub-queries that can be executed in the selected sources. Finally, a plan that composes the sub-queries is generated; physical operators are selected in order to minimize execution time and maximize answer completeness. State-of-the-art federated query engines are not able to generate efficient query execution plans for a federation of heterogeneous data sources since they are designed and optimized for a single data model; Ontario overcomes this downside by considering the types and capabilities of the sources.

1.2 Contributions

State-of-the-art federated query engines are designed to generate optimized query execution plans for federations of data sources in a common data model, e.g., RDF. Lifting a data set to RDF might not be feasible due to the high time complexity. Therefore, the data sources in a Semantic Data Lake are kept in their original data model. Mapping definitions like RML mappings can be used to add semantics to non-RDF data sources, e.g., relational databases. However, considering heterogeneity adds complexity to query processing due to the varying capabilities of the different data sources. Using RDF-MT based source descriptions enables effective generation of query execution plans. The heuristics used in existing query engines are tailored for a specific data model or considered general since they do not take into account special capabilities of a data model. This thesis shows that using general heuristics in a Semantic Data Lake is not sufficient for optimizing the query execution plans. Therefore, two sets of heuristics are proposed; one set of heuristics contains general heuristics only, and the second set includes source specific heuristics as well. Source specific heuristics are tailored for a specific data model that are likely to increase query performance in a heterogeneous federation. The empirical evaluation shows that source specific heuristics are able to improve the query performance and that some queries benefit from the execution in a Semantic Data Lake compared to a federation of RDF graphs.

1.3 Chapters

This thesis is structured as follows: Section 2 describes the basic concepts that underlie the development of this thesis. Related work is discussed in Section 3. The proposed approach is presented in Section 4. Section 5 examines the implementation of the approach. The experiments are described and their results analyzed in Section 6. Finally, Section 7 concludes and presents further directions of research.

Chapter 2

Background

This chapter introduces the main topics needed to understand the development of this thesis. In the first part, Semantic Web technologies like the Resource Description Framework (RDF), a query language for RDF data, extensions of RDF and further related concepts are discussed. Afterwards, data integration systems including architecture and schema mapping are presented. The third part explains physical database design which covers indexing and schema normalization. Finally, federated query processing and query optimization are examined.

2.1 Semantic Web Technologies

The *Semantic Web* extends the *World Wide Web* through standards established by the *World Wide Web Consortium* (W3C). These standards describe common data formats and protocols for data exchange. The Semantic Web aims to enhance the World Wide Web with semantics, i.e., a web of data readable by humans and machines. Tim Berners-Lee describes his original vision of the Semantic Web as in the following:

> "I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy, and our daily lives will be handled by machines talking to machines, leaving humans to provide the inspiration and intuition. The intelligent 'agents' people have touted for ages will finally materialize. "
>
> — Tim Berners-Lee, *Weaving the Web* (1999)

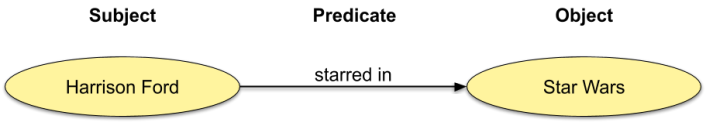


Figure 2.1: **Abstract Representation of an RDF Triple**

2.1.1 Resource Description Framework

The *Resource Description Framework* (RDF) [26] is a graph-based data model and W3C standard for publishing and exchanging data over the web. The graph is created from the basic building block of RDF, the RDF triple. Each triple consists of three parts: **(i)** subject - an entity or resource, **(ii)** object - an entity or resource, and **(iii)** predicate - a relation between subject and object. All parts of the triple are represented using *Universal Resource Identifiers* (URI). Only objects can be represented as a literal instead of a URI to use data formats like string, integer or date. An RDF graph is a directed graph with labeled edges [3], the nodes represent the subjects and objects while the predicates are the labels of the edges. Figure 2.1 illustrates an abstract representation of an RDF triple.

RDF uses an own vocabulary to specify the entities and their relations. Several serialization formats for RDF have been created such as N-Triples [9], RDF/XML [15], and Turtle [36]. Listing 2.1 shows the use of the RDF vocabulary to represent the example from Figure 2.1 in the Turtle format. Two resources (Actor and Movie) and one relation (starred_in) are defined. Then the fact that *Harrison Ford* starred in *Star Wars* is added. The prefix `rdf` describes the RDF vocabulary and `base` is the base for the relative URIs used for describing the data. The semicolon is used for adding another predicate and object to the same subject.

Listing 2.1: **Example of RDF/Turtle format**

```
@base <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-systax-ns#> .
<Actor> rdf:type rdf:Resource .
<starred_in> rdf:type rdf:Property .
<Movie> rdf:type rdf:Resource .
<Star_Wars> rdf:type <Movie> .
<Harrison_Ford> rdf:type <Actor> ;
    <starred_in> <Star_Wars> .
```

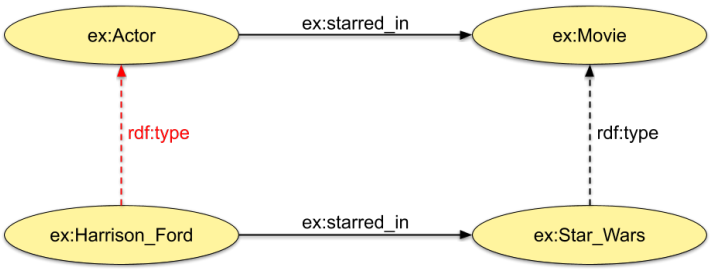


Figure 2.2: **RDFS Inference Example.** Inferred properties shown in red.

2.1.2 RDF Schema

RDF Schema (RDFS) [7] is an extension of RDF. RDFS introduces new vocabularies to define more complex relations between resources, e.g., defining hierarchies of classes with `rdfs:subClassOf` and specifying the range and domain of properties with `rdfs:domain` and `rdfs:range`, respectively. These extensions can be used for inference which is one of the advantages of RDFS.

The RDF graph in Figure 2.2 represents three triple statements **(i)** an actor stars in a movie, **(ii)** *Star Wars* is a movie, and **(iii)** *Harrison Ford* starred in *Star Wars*. The property `ex:starred_in` is modeled with the domain `ex:Actor` and the range `ex:Movie`. Applying RDFS inference results in the implicit knowledge of *Harrison Ford* being an actor. Imagine `ex:Actor` being a subclass of `ex:Human`. Using the inference rules [3], *Harrison Ford* is also a human.

2.1.3 SPARQL

SPARQL Protocol And RDF Query Language (SPARQL) [37] is a W3C standard query language for RDF. It can be used to extract, define, and manipulate the data. SPARQL is based on the Turtle serialization and *basic graph pattern* matching. A SPARQL query consists of triple patterns, an RDF triple that contains variables at any arbitrary place. A set of triple patterns is called basic graph pattern (BGP). The main query clauses supported are **(i)** `SELECT` for retrieving the results of the query, **(ii)** `CONSTRUCT` for constructing an RDF graph from the query, **(iii)** `ASK` for checking if the BGP exists in the data set, and **(iv)** `DESCRIBE` for descriptions of the data [3]. SPARQL queries may contain operators, e.g., `AND` (denoted by `.`), `UNION`, and `OPTIONAL`, to connect BGPs as well as `FILTER` to filter from the output the instantiations of variables that meet a certain condition. Listing 2.2 shows the example SPARQL query *"Get all movies Harrison Ford starred in"*. This query is composed of three triple patterns, e.g., `ex:Harrison_Ford ex:starred_in ?movie`

Listing 2.2: **Example of a SPARQL Query**

```
PREFIX ex: <http://www.example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?movie ?year WHERE {
       ex:Harrison_Ford ex:starred_in ?movie .
       ?movie rdf:type ex:Movie .
       ?movie ex:year ?year .
}
```

corresponds to *Harrison Ford* starring in `?movie` which is restricted to a movie in the next triple pattern.

On one hand the evaluation of a SPARQL query over an RDF graph using only the `AND` and `FILTER` operators is possible in polynomial time, on the other hand adding `UNION` or `OPTIONAL` operators to the query increases the complexity of query evaluation to NP-complete [3]. The complexity can be reduced to coNP-complete by rewriting the query to be in *union normal form* and being *well-designed*. A graph pattern is in union normal form if it is of the form:

$$(P_1 \text{ UNION } P_2 \text{ UNION } ... \text{ UNION } P_n)\,, \tag{1}$$

where each P_i $(1 \leq i \leq n)$ do not contain the `UNION` operator. A graph pattern without `UNION` and using only a subset of the variables for `FILTER` expressions is well-designed if for every sub-pattern $P' = (P_1 \text{ OPT } P_2)$ variables occurring in P' and P_2 also occur in P_1 [3].

2.1.4 RML Mappings

RDF Mapping Language (RML) mappings [11] are used to semantify data in other formats than RDF, i.e., they can be used to model domain-level knowledge for data derived from heterogeneous sources. For example tabular data about movies like in Table 2.1 can be enriched with semantics using an RML mapping like the one in Listing 2.3, assuming the data is stored in a TSV file named `movies.tsv`.

An RML mapping consists of different mappings, in the example only one mapping `:movies` is defined. Each mapping has a source description that specifies the source path and type. An iterator has to be defined in order to be able to access the data correctly. This iterator varies for different data types as well as the actual data representation. In addition to the source description the mapping contains a reference to the subject class of the mapping (`subjectMap`). The predicates and objects belonging to the subject are specified using `predicateObjectMap`s. This map includes

title	**year**	**studioName**
Star Wars	1977	Fox
Raiders of the Lost Ark	1981	Paramount
Stargate	1994	Centropolis

Table 2.1: **Minimal Relational Data on Movies**

the URI of the predicate and the predicate range. If the object is a literal, a reference to the corresponding column is made, e.g., `rml:reference "year"`. Else the object is a resource and a template is used to create the URI of the object from the column value, e.g., `rml:template "<http://www.example.org/{studioName}>"` evaluates to `<http://www.example.org/Fox>` if the value is `Fox`.

Listing 2.3: **Example RML Mapping for Movies**

```
@prefix : <http://www.example.org/> .
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix rml: <http://semweb.mmlab.be/ns/rml#> .
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .

:movies
    rml:logicalSource [
        rml:source "movies.tsv";
        rml:referenceFormulation ql:TSV;
        rml:iterator "*"
    ];
    rr:subjectMap [
      rr:template "http://www.example.org/{title}";
      rr:class <http://www.example.org/Movie>
    ];
    rr:predicateObjectMap [
        rr:predicate <http://www.example.org/year>;
        rr:objectMap [
            rml:reference "year"
        ]
    ];
    rr:predicateObjectMap [
        rr:predicate <http://www.example.org/at_studio>;
        rr:objectMap [
            rml:template "<http://www.example.org/{studioName}"
        ]
    ];
```

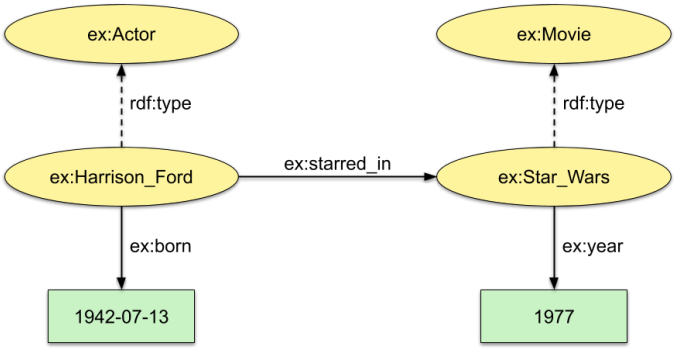
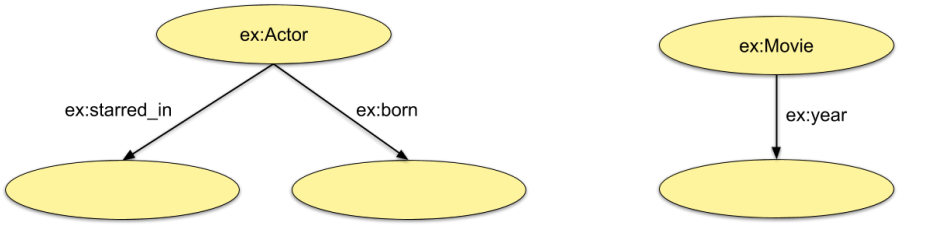


Figure 2.3: **Example RDF Graph for Actor and Movie Classes**



(a) RDF-MT for Actors (b) RDF-MT for Movies

Figure 2.4: **RDF Molecule Templates from Figure 2.3**

2.1.5 RDF Molecule Templates

RDF Molecule Templates (RDF-MTs) are an abstract description of entities belonging to the same RDF class [13]. An RDF-MT of an entity consists of a subject representing the RDF class and a number of properties that belong to the subject. Figure 2.3 shows an example RDF graph with two classes (Actor and Movie). From this RDF graph two RDF-MTs can be extracted. Each RDF-MT contains the predicates belonging to the subject. As RDF-MTs are metadata representing an RDF class the predicates are connected to blank nodes. The example from Figure 2.3 contains three predicates, namely `ex:starred_in`, `ex:born`, and `ex:year`. Two of them belong to the class `ex:Actor` and the remaining one describing the year of release belongs to the class `ex:Movie`. The resulting RDF-MTs are presented in Figure 2.4.

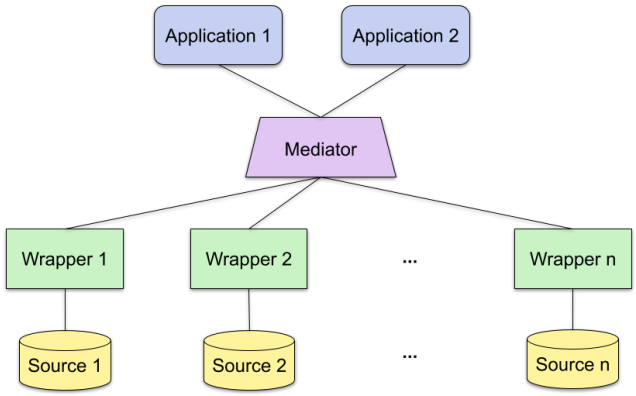


Figure 2.5: **Mediator/Wrapper Architecture**

2.2 Data Integration Systems

Data integration combines the data from different sources and provides a unified view of them to the user. This process is required for example when two companies need to merge their databases or in scientific research when combining research results from different repositories. Data integration has been extensively studied since the work on rewriting queries using views in the mid-1990's [17]. This section covers the *Mediator/Wrapper architecture*, schema mapping approaches and two different data integration systems, i.e., the classical *Data Warehouse* system and the one this thesis is based on, the *Semantic Data Lake*.

2.2.1 Mediator/Wrapper Architecture

The *Mediator/Wrapper architecture* was proposed in the early 90's to aid decision-making applications [48]. Since then it has become a very popular architecture for data integration systems despite the fact that wrapper development involves a lot of work. The architecture is presented visually in Figure 2.5. Each data source has an associated wrapper that provides information about the source, e.g., query processing capabilities. The mediator collects the information provided by the wrappers and performs query execution using the wrappers to access the data sources. The mediator then integrates the results from the wrappers to one unified result. The mediators have to evolve with the changes of the data sources in use. Therefore, Ericsson [14] comes up with a system that is able to deal with unavailable sources by checking for their health status. Wiederhold [47] suggests to limit the scope of a mediator to commensurate semantics.

2.2.2 Schema Mapping Approaches

Data integration systems provide a uniform query interface to the possibly heterogeneous data sources that are integrated. Therefore, a *global schema* is introduced that is used by the users to post their queries. Hence, it is not necessary for the users to know the schemas of all data sources.

There are two main ways to create schema mappings, namely *Global-as-View* (GaV) [27] and *Local-as-View*. The GaV approach models the global schema as a view over the local schemas of the data sources. On the contrary, LaV models the content of a data source as a view over the global schema. There is also a hybrid approach known as *Global-Local-as-View* (GLaV) that combines the two aforementioned approaches, i.e., a view over the data sources is defined as a view over the global schema [17]. A view definition maps the global and local relations of a schema. For example, if the global movie relation holds the title, year, and length of a movie and one of the sources has a relation of movies from 1991 with title, length and director the view definition would map the titles and lengths, setting the year in the global schema to 1991 and omitting the information about the director. The main advantage of the LaV approach is the easy integration of new data sources since the global schema does not need to be changed. Therefore, the query rewriting in the LaV approach has quite a high complexity.

Rewriting a query posed over the global schema into a query on the individual data sources in a system using the GaV approach is straightforward. The query will simply be unfolded using the view definitions, i.e., every relation from the global schema mentioned in the query will be replaced by its view definition. As the view definition contains the relations from the local data sources the unfolded query is a query over the individual data sources. Query rewriting in the GaV approach is linear in the number of global relations used.

In the LaV approach the query reformulation is not as easy and matches to the problem of query rewriting using views [17]. One possibility to rewrite a query in the LaV approach is the bucket algorithm [20]. This algorithm uses a bucket per each literal of the query and adds the views to the bucket that contain the literal. The buckets need to be combined (join of the views) in an efficient way. During the process partial containment mappings are generated and checked for incompatibilities. If a partial containment mapping covers all literals of the query the containment mapping and plan for the query is found. The bucket algorithm is sound but not complete, there are plans the algorithm does not find. Let V be the set of views, $n = \max(|V|)$, and $|q|$ the number of buckets. The complexity of the bucket algorithm is as follows:

$$(|V| \cdot n)^{|q|} \tag{2}$$

2.2.3 Data Warehouse

A traditional data integration system is the *Data Warehouse*. Usually a Data Warehouse is an integrated database within a company and used for data analysis in decision support. A Data Warehouse stores all the data from different databases and sites of the company in a centralized manner. For the user the Data Warehouse looks like an ordinary database, but data manipulation queries generally are forbidden since they would lead to the Data Warehouse being inconsistent with the sources [16]. Data warehousing, therefore, it implements materialized data integration. Data Warehouses follow the *extract-transform-load* (ETL) paradigm. The needed data will be updated from the sources periodically (extract, load) and transformed to match the *warehouse schema* (transform). The data access strategy used is *data on-write*, i.e., the data is organized in a static schema before it is processed [4]. Queries will be run over the Data Warehouse only and, therefore, complex queries for data analysis do not interfere with the everyday workload. Since the data is updated periodically one issue with data warehousing is the up-to-dateness of the data.

2.2.4 Semantic Data Lake

Data Lakes are often regarded as the contrary of Data Warehouses. The data in a Data Lake will not be transformed into a common data format, but stay in its original one. Therefore, a Data Lake implements *virtual data integration*. Also the data access strategy is different. Data Lakes use *data on-read*, i.e., the schema of the data will be checked when the data is used. Data Lakes are designed to store and access heterogeneous data. However dealing with this data is time-consuming and inefficient due to the different data models, instance structures and file formats [4].

To overcome the aforementioned problem the Data Lake can be semantified. A semantified Data Lake is called *Semantic Data Lake* (SDL). To add semantics to the Data Lake the data sets are equiped with mappings (cf. Subsection 2.1.4) to vocabularies, ontologies or knowledge graphs. This allows the user to deal with the heterogeneous data as if it was in one common format.

The data can then be queried in a unique high-level declarative query language. The query processing over Semantic Data Lakes has to face the following challenges: **(i)** *Query Decomposition:* decompose the query into sub-queries and generate an execution plan, **(ii)** *Source Selection:* select relevant sources for the sub-queries using the predefined mappings, and **(iii)** *Result Extraction:* the sub-queries are translated into the query language of the selected data sources, executed and the results are combined according to the generated plan, in a way that the original query will be answered correctly.

2.3 Physical Database Design

When setting up a relational database some physical design aspects have to be considered. For the matter of this thesis *indexing* and *schema normalization* are most important. First, indexing is discussed, followed by normalization.

2.3.1 Indexing

An index is any data structure that given the value of one or more attributes of a relation as input returns the records with this value quickly [16], i.e., finding all matching records with checking only a fraction of all possible records of the relation. Therefore, query execution can be sped up with appropriate indexes since they may improve the time needed to find all records of a relation meeting a certain condition. Indexing has been extensively studied in the past and many different kinds of indexes are implemented in relational database management systems. Most commonly known is the *primary index* as the index over the *primary key* of a relation. Indexes over candidate keys are called *unique indexes* since they do not contain any duplicates. As those indexes may not be sufficient to speed up the queries frequently send to the database, indexing attributes repeatedly appearing in the `WHERE` clause should be considered. This type of indexes is known as *secondary index*. Indexes may contain more than one attribute and are called *composite index* if they are combined of at least two attributes. When the ordering of data records is the same as the ordering of data entries in an index, the index is called *clustered index* [38].

The index structure used should be decided depending on the kind of comparisons made on that attribute. The simplest form of an index might be a sorted file with an associated *index file* containing key-pointer pairs where a pointer refers to a record in the data file. This type of index structure is used for primary indexes. The two most commonly used structures for indexes are *B-trees* and *hash tables*. A B-tree organizes its blocks in a tree structure. The search keys are stored in a sorted order in the leafs of the tree and copied from the data file. The last pointer in a leaf points to the next right leaf, the block with the next higher search keys. B-trees are best for range searches like `year > 1970 and year < 1980` [38]. Hash tables however are best for exact matches like `year = 1977` [38]. The search key will be hashed using a *hash function* which determines the bucket the search key will be stored in. Retrieving the records matching a search key K is as simple as returning the records linked to the bucket $h(K)$ where h is the hash function. Depending on the number of records the hash table can be kept in memory or needs to be stored on a secondary storage. Hash tables are an alternative choice for primary indexes.

Inserting, updating or deleting a record from a relation leads to an update of the indexes on this relation. Therefore, creating an index is always a trade-off between query execution time, storage use, and data manipulation time. Ramakrishnan and Gehrke [38] present some guidelines for index creation. **(i)** Attributes should only be indexed if a query benefits from it and indexes that speed up more than one query should be preferred. **(ii)** Attributes occurring in a `WHERE` clause are candidates for indexing. **(iii)** Composite indexes should be considered if a `WHERE` clause includes conditions on more than one attribute or if they enable index-only evaluation, i.e., accessing the relation can be avoided, for important queries. **(iv)** At most one index per relation can be clustered. Since clustering has a significant impact on performance the choice of clustered indexes is important. Range queries are most likely to benefit from clustering. Indexes enabling index-only evaluation need not to be clustered. **(v)** B+ trees are usually preferable over hash indexes, but a hash index is better if the index is intended to support index nested loop joins or only very important equality queries and no range queries use the index. **(vi)** The impact of an index on the updates in the workload has to be studied. If maintaining the index slows down frequent update operations, dropping the index should be considered. However, an index may speed up certain update operations. For example, an index on the ID of a movie could speed up an update of the length of that movie specified by the ID. **(vii)** An attribute should not be indexed if there exists a value for this attribute that is present in more than 15% of the records.

2.3.2 Normalization

Creating a relational database schema without paying sufficient attention may lead to problems. The main problem is *redundancy*, i.e., the same information is repeated in more than one tuple. Examples are `year` and `length` of a movie in Table 2.2. When having redundancies one may face *update anomalies* and *deletion anomalies* [16].

title	**year**	**length**	**studioName**	**starName**
Star Wars	1977	124	Fox	Carrie Fisher
Star Wars	1977	124	Fox	Mark Hamill
Star Wars	1977	124	Fox	Harrison Ford
Raiders of the Lost Ark	1981	115	Paramount	Harrison Ford
Raiders of the Lost Ark	1981	115	Paramound	Karen Allen
Stargate	1994	130	Centropolis	Kurt Russell

Table 2.2: **Exhibiting Anomalies in Relational Data**

The former occur when the redundant attribute is updated in just one of the tuples but not in the others. For example, if *Raiders of the Lost Ark* is really 116 minutes long and one carelessly updates only the first tuple of Table 2.2. The latter anomaly describes deleting the last tuple from the redundant attribute which leads to the loss of the information from the other attributes. This would happen if one deletes *Kurt Russel* from the set of stars from the movie *Stargate* in Table 2.2. There would be no more stars for this movie and all other information would be lost too.

Therefore, relations should be decomposed to avoid anomalies. Decomposing a relation can be understood as splitting the relation into multiple relations. Each of these relations is a subset of the original relation. By combining the information from all these relations the original information can be restored [44]. Normalization is achieved by decomposing the relations. There are several *normal forms* describing the grade of normalization.

Definition 1 (First Normal Form). *A relation is in First Normal Form (1NF) if it contains only atomic values.*

The values of a relation have to be atomic, i.e., they can not be further decomposed. Date [10] suggests that 'the notion of atomicity has no *absolute meaning*'. Therefore, one may consider a value atomic for some purpose but not for another.

Definition 2 (Second Normal Form). *A relation is in Second Normal Form (2NF) if there are no partial dependencies.*

A relation should not have a functional dependency that is only partially dependent from the candidate key. Hence, this rule is violated by a relation having $\{A\}$ as the set of key candidates and a functional dependency $AB \rightarrow C$ because there is no key candidate that includes A and B.

Definition 3 (Third Normal Form). *A relation is in Third Normal Form (3NF) if there are no transitive dependencies.*

Every relation in 3NF does not have functional dependencies which are transitive dependent from a candidate key, i.e., if $\{A\}$ is the set of candidate keys and the relation has the functional dependencies $A \rightarrow B$ and $B \rightarrow C$ this rule is violated due to the transitivity of the dependencies $(A \rightarrow B \rightarrow C)$.

Definition 4 (Boyce-Codd Normal Form). *A relation is in Boyce-Codd Normal Form (BCNF) if for each non-trivial functional dependency $(X \rightarrow Y)$, X is a super key.*

BCNF is more strict than 3NF and ensures that no redundancy can be detected with information about the functional dependencies alone [38]. BCNF enforces that

each functional dependency is dependent on the full key. A relation with the candidate key $\{A\}$ and a functional dependency $B \rightarrow C$ violates this rule. However for the same relation the functional dependency $AB \rightarrow C$ is valid because AB is a superkey, i.e., a superset of a candidate key.

Definition 5 (Fourth Normal Form). *A relation is in Fourth Normal Form (4NF) if there are no multi-valued dependencies.*

Any table in BCNF that has no multi-valued dependencies is also in 4NF. The multi-valued dependency $A \twoheadrightarrow B$ holds for a relation $R(A, B, C)$ if for the tuples (a, b, c) and (a, d, e) in R also the tuples (a, b, e) and (a, d, c) exist in R.

Definition 6 (Fifth Normal Form). *A relation is in Fifth Normal Form (5NF) if every non-trivial join dependency is implied by the candidate keys.*

Definition 7 (Sixth Normal Form). *A relation is in Sixth Normal Form (6NF) if there are no non-trivial join dependencies.*

If a relation schema S is decomposed in relations R_1 to R_n, the decomposition will be a lossless-join decomposition if the relations on S are restricted to a join dependency on S, denoted as $*(R_1, R_2, ..., R_n)$. A join dependency is trivial, if one of the R_i is S itself. The join dependency is implied by the candidate key(s) if and only if each of the relations R_1 to R_n is a superkey for S.

There are algorithms for transforming the relation schema to the different normal forms. For this thesis the algorithm from Bernstein [6] for transforming the schema to 3NF is used. The procedure of this transforming is depicted in Algorithm 2.1.

Algorithm 2.1 Synthesis Procedure by Bernstein [6]

1: let F be the set of functional dependencies, find a canonical cover G for F
2: partition G into groups such that all of the FDs in each group have identical left sides
3: merge groups of FDs if they are dependent from one another
4: construct a relation for each of these groups
5: if none of the relations contains a candidate key, add a new relation with the candidate key

2.4 Federated Query Processing

The databases in a federated system are autonomous, possibly heterogeneous, and maybe geographically decentralized [35]. The most often used architecture in federated settings is the *Mediator/Wrapper architecture* discussed in Subsection 2.2.1.

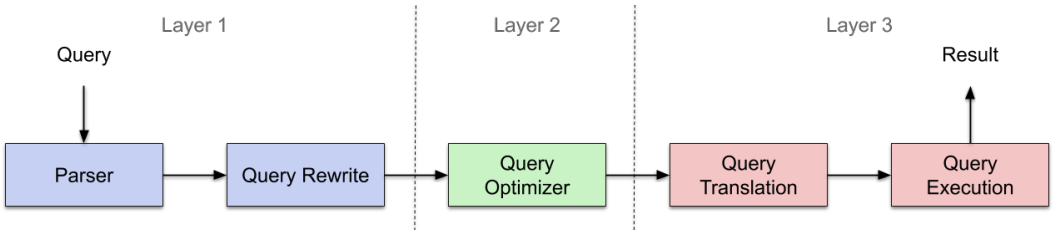


Figure 2.6: **Query Processing Pipeline**

Query processing in a federation can be split in three layers as indicated by different colors in Figure 2.6. The first two layers, located at the mediator, transform the input query to a globally optimized query execution plan (QEP) [35]. The first layer parses the query and afterwards uses the global schema to rewrite the query into a query on local relations. As mentioned before there are mainly two different approaches on how to define the global schema (cf. Subsection 2.2.2). The actual rewriting technique depends on which approach was chosen for the global schema. Rewriting the query also includes decomposing the query into sub-queries that are executable by the sources and identifying the relevant sources that contribute to the answer of these sub-queries. The second layer performs global optimization of the query and some execution. This layer considers the allocation of the local relations and the query processing capabilities of the different data sources. Query optimization itself is further discussed in Subsection 2.4.1. The output is an optimized QEP with sub-queries and operations that can be executed by the data sources. The third layer is located at the wrappers and performs query translation and execution. It also returns the obtained results to the mediator that performs result integration. Each wrapper needs the local export schema of the associated source to translate the sub-query into the language used by the data source. Translating may also include data conversion, e.g., if the global schema represents distances in meters, but a data source in the federation uses inches. The query is executed after the translation and the local results are transformed to the common format.

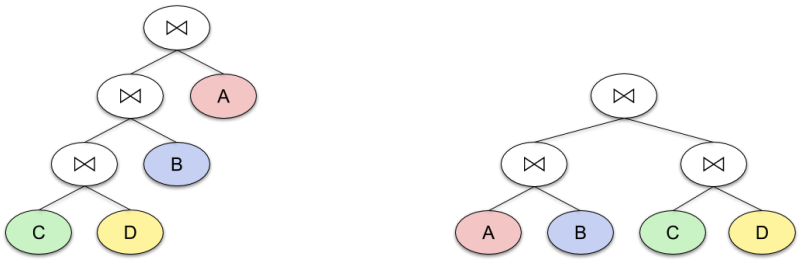
2.4.1 Query Optimization

The objective of the query optimizer is to find a good QEP. This involves two basic steps. First, enumerating alternative plans, typically only a subset of all possible QEPs is considered due to the very large number of possible plans. Second, estimating the cost of the enumerated plans and choosing the plan with the lowest cost [38]. The query optimization can be done at two different times relative to the

query execution: **(i)** *statically* before the execution of the query, and **(ii)** *dynamically* during the execution [35]. Since static query optimization is done before the query execution, the sizes of intermediate results are unknown and have to be estimated. Errors in these estimations can lead to a suboptimal QEP. Following this approach, it is possible to cache the QEP for a query in order to save the optimization time on subsequent requests for the same query. Dynamic query optimization is progressed at query execution time. Therefore, at any point of the execution the decision on which operator to use next can be based on the actual size of the result previously produced. The main advantage of dynamic query optimization is that the actual sizes of intermediate results are available and, therefore, minimizing the probability of creating a bad plan. However, the shortcoming of this approach is that the expensive task of query optimization has to be repeated for each execution of the query.

The most popular algorithm to enumerate the plans is *dynamic programming (DP)*. [42] contains an overview of alternative enumeration algorithms. DP creates very good plans if the cost model is accurate enough. The downside of this approach is the exponential time and space complexity and, therefore, it is not feasible for complex queries. An adaptive extension called *iterative dynamic programming* is proposed in [25] to overcome this disadvantage. It produces plans as good as DP for simple queries and good plans for queries too complex for DP. However, iterative dynamic programming is not suitable for very complex queries with dozens of relations. There are also *greedy* algorithms, but following the nature of the *greedy paradigm* they might get caught in a local optimum. [31] proposes a hypergraph-based approach to query optimization. Each sub-query will be handled as a node in the hypergraph. Nodes that share a join variable are merged into a new hypernode. Those hypernodes represent joins of sub-queries.

DP works in a bottom-up manner, that is starting with sub-plans that only involve one table. The best *access plan(s)* are considered for the next step. Then all two-way *join plans* using the access plans are evaluated. This includes evaluating different implementations of the join based on the query. Since using an implementation that returns an ordered result may be the better alternative if the data is needed to be sorted for a later operation. Afterwards, all three-way join plans are computed using the plans from the previous steps. If the query involves n tables, the algorithm continues in this fashion until all n-way join plans are enumerated. Inferior plans are pruned as early as possible. A plan can be pruned if there is an alternative plan that does at least the same amount of work with lower costs. When an inferior plan is pruned it is no longer be considered by the following steps of the plan generation. Thus, the complexity is significantly reduced.



(a) Left-Deep Tree (b) Bushy Tree

Figure 2.7: **Example of Left-Deep and Bushy Trees**

The search space can be further reduced if bushy trees are not considered but left-deep trees only. Figure 2.7 shows examples of left-deep and bushy trees. Bushy trees allow for more parallelism in query execution, e.g., the join of A and B can be executed in parallel with the join of C and D. For left-deep trees the inner operand of a join is always a base relation, ensuring indexes can be used. Relational optimizers consider left-deep plans only [38].

The *classical cost model* estimates the cost of every operator of the plan and then sums up the individual costs. The cost of a plan is the total resource consumption. In a centralized system this will include CPU costs and disk I/O. In a distributed system also communication cost must be considered. The *response time model* which does not estimate the resource consumption but the response time. It takes intraquery parallelism into account and finds the plan with the lowest response time.

One of the challenges of query optimization when dealing with heterogeneity in a federated database system is that the capabilities of the sources may differ from each other [24]. Several approaches to this problem have been proposed like describing the capabilities as views or context-free grammars. DP accesses this problem by using *planning functions* provided by the wrappers to enumerate the plans. Another challenge is the cost estimation since it is not necessarily known how the wrapper executes the plan [24]. The *calibration approach* uses a generic cost model for all wrappers and adjusts the parameters of this model based on the results of test queries. An alternative would be *individual wrapper cost models*. Using this approach the developers of the wrapper also provide cost formulas to calculate the cost of the generated plans. The advantage of this approach is that the cost estimation can be as accurate as possible. However, the downside of this approach is that wrapper developers are left with the complicated task of creating cost formulas. The *learning curve approach* keeps statistics of the execution costs of queries and estimates the cost of a plan based on these statistics. This approach releases the wrapper developers from the burden of cost estimation, but it can be very inaccurate.

2.4.2 Performance Measures

There are several measures to compare the performance of query processing engines. Standard measures for query processing performance include the time to first answer, time to last answer, source selection time, query decomposition time, total query execution time, cardinality of the result set and answer completeness. In the following a measure for continuous efficiency will be presented.

The *diefficiency*, introduced in [1], measures the continuous efficiency of a query engine, i.e., quantifying the answer generation in terms of continuity. In order to calculate the diefficiency answer traces that record the point of time when the answer is generated are needed. There are two different methods for calculating the diefficiency, $dief@t$ and $dief@k$.

The diefficiency at time t measures the continuous efficiency of the query engine in the first t time units of query execution. Therefore, the area under the curve of the answer distribution until t is computed:

$$dief_{\rho,Q}@t := \int_0^t X_{\rho,Q}(x)\,dx \tag{3}$$

Where ρ is the engine that executes query Q with the answer distribution function $X_{\rho,Q}$. A higher value means a better efficiency in terms of continuously producing answers over time.

The diefficiency at k answers measures the continuous behavior from the first produced answer up to the k-th answer. This results in $dief@k$ being calculated as follows with t_k being the point of time when the k-th answer of query Q was produced:

$$dief_{\rho,Q}@k := \int_0^{t_k} X_{\rho,Q}(x)\,dx \tag{4}$$

A lower value is better in terms of producing answers continuously.

The concepts described in this chapter are important to understand the development of this thesis. The Semantic Web technologies play a key role in the query processing of the proposed approach. It is also crucial to know the difference between Data Warehouses and Semantic Data Lakes, i.e., the data in a SDL stays in its original format. In order to optimize the query execution plans for RDB queries a good understanding of physical database design is needed; the main focus is on indexing and schema normalization. In addition, knowledge about query optimization is important to be able to improve the query plans and define heuristics guiding the optimizer. Those concepts are also used in the work related to this thesis.

Chapter 3

Related Work

Topics related to this thesis have been extensively treated in the literature. This chapter presents an overview of what has been done already in federated query processing and query optimization for RDF data sources and in terms of heterogeneous data sources. The research related to this areas is driven by the Database Community as well as the Semantic Web Community.

3.1 Federated Query Processing over RDF Data

Several query engines have been proposed for federated query processing against RDF data sources. *FedX* [41] is one of those query engines. FedX aims at minimizing the number of requests to be send to the sources and uses exclusive groups for source selection, i.e., a group of triple patterns that can be exclusively evaluated by a single endpoint. This approach does not rely on any preprocessed metadata, but relies on the *variable counting approach* as a basic heuristic for join ordering, i.e., the more variables an exclusive group contains the higher the estimated cost will be. FedX generates left-deep tree plans.

SPLENDID [18] is a query engine that solely relies on statistics extracted from the *Vocabulary of Interlinked Datasets* (VoID) description of the data sources even though it was not initially designed for query optimization. The VoID descriptions are aggregated in a local index. General information like triple count and number of distinct subjects, predicates and objects are stored for each data set. Statistical information for every predicate and RDF type are organized in inverted indexes. The indexes are used for preselecting sources and `ASK` queries are used to prune sources that can not answer the triple pattern. SPLENDID uses exclusive groups like FedX, but it prefers bushy trees.

ANAPSID [2] is an adaptive query processing engine for SPARQL endpoints. For each endpoint ANAPSID stores a list of the predicates in the endpoint. This information is then used for query decomposition. ANAPSID decomposes the query in star-shaped sub-queries [45]. The novelty of this approach is the use of adaptive physical operators to produce results as soon as they arrive from the sources. The evaluation showed that the adaptive operators implemented perform better than the traditional blocking operators. ANAPSID also implements heuristics for source selection for star-shaped queries [30].

SemLAV [34] uses the *Local-as-View* approach to describe the global schema of the federation of endpoints. The query rewriting problem is NP-complete and, therefore, SemLAV avoids query rewriting by building an instance of the global schema on-the-fly with data from the relevant views. In contrast to traditional wrappers the SemLAV wrappers return RDF graphs that are composed of the triples that match the view definition. In traditional LaV approaches as many rewritings as possible are executed and SemLAV follows this idea by instanciating as many views as possible. The number of covered rewritings is dependent on the order in which the views are instanciated. SemLAV proposes a solution to the so called *maximal coverage problem*.

Fedra [33] tackles the problem of data availability from public endpoints. Therefore, fragmentation of the data sets and opportunistic replication of the fragments is proposed. During source selection a *divergence threshold* must be met by the relevant sources in order to be selected. Source selection can rely on fragment definitions in federations with quite static data or use `ASK` queries for dynamic data. Fedra avoids contacting public endpoints if possible and selects as few endpoints as possible. For deciding which endpoints to prune fragment containment is used. Fedra then executes the query plan over an existing query engine.

LILAC [32] is an approximate solution to the *query decomposition problem with fragment replication* (QDP-FR) and can be used with state-of-the-art federated query engines. In contrast to Fedra LILAC assumes all fragments are perfectly synchronized. Therefore, LILAC generates the largest Cartesian product free sub-queries possible to evaluate the triple patterns with just one relevant fragment. One unique consideration by LILAC is to merge non-selective sub-queries with joinable sub-queries that can be executed at the same endpoint.

MULDER [13] is a federated query engine and uses the adaptive operators of ANAPSID and star-shaped sub-queries. The sources are described in terms of RDF Molecule Templates which have been extracted from the sources in advance. The decomposition into star-shaped sub-queries as well as the source selection is guided by the RDF Molecule Templates. MULDER reduces query execution time and increases

answer completeness by using semantics in the source descriptions.

Fed-DSATUR [46] applies graph theory to the *federated SPARQL query decomposition problem* and maps it to the *vertex coloring problem*. The vertex coloring problem is to find a coloring of the graph with the minimal number of colors used such that no neighboring vertices have the same color. This problem is NP-complete and, therefore, Fed-DSATUR is an approximate solution implementing a greedy iterative algorithm. Vertices correspond with the triple patterns from the query while the colors represent the selected endpoint. This approach finds a good trade-off between execution time and answer completeness by trying to minimize the number of star-shaped sub-queries and not dropping to many relevant sources.

Optimizing SPARQL queries in a federated system is a challenge. Cost-based approaches are often not feasible because the database does not have the needed statistics for such an approach. The *heuristic-based SPARQL planner* (HSP) [43] uses several heuristics that consider triple pattern order and join positions to reduce the query execution time. The proposed heuristics can be used independently or combined.

3.2 Polystores

More recently, research has begun to also focus on query processing against heterogeneous data sources. Different approaches on how to store, integrate, and query the heterogeneous federation have been proposed. Next some of these solutions will be discussed in detail.

SeBiDA [28] is a proof-of-concept for a semantified Big Data architecture proposing three requirements for such architectures: **(i)** ingesting semantic and non-semantic data, **(ii)** preserving semantics and metadata, and **(iii)** enabling scalable and efficient query processing. Datasets are differentiated in semantic, annotated with semantics and non-semantic. Non-semantic data sets can optionally be lifted with semantics if mappings are provided. SeBiDA uses *Apache Spark* to reformat the data according to classes. The metadata will be stored in *MongoDB* and the reformatted data in *Apache Parquet* tables. Therefore, the data is integrated in a centralized or clustered manner and can be queried using SQL.

Constance [19] is a Data Lake system storing the data from different sources in their original format with focus on collecting and matching metadata. Metadata is not only collected during the ingestion phase but also in the maintenance layer of the Constance architecture by a component called *Structural Metadata Discovery*. Despite the name this component also extracts metadata from semi-structured data. The *Semantic Metadata Matching* component uses semantic annotations to link at-

tributes with different labels across the data sources. Constance provides keyword search as well as answering JSONiq queries.

Ontario [4, 12] is a Semantic Data Lake implementation. The sources are kept in their original format and mapped to existing semantic vocabularies using RML mappings if they are not in RDF. Therefore, the global schema looks like an RDF graph and can be queried with SPARQL. Like in MULDER queries are decomposed in star-shaped sub-queries based on RDF Molecule Templates. Furthermore, Ontario uses the adaptive operators introduced in ANAPSID. During query execution the wrappers translate the SPARQL query to the native query language of the data source. In contrast to earlier approaches, Ontario does not transform the data from the different data sources into a common format but is able to use the data sources in their original format.

PolyWeb [23] is a federated query engine that supports data sources in different data formats. State-of-the-art query engines support only one single format, e.g., RDF or relational databases. PolyWeb works with the native data of the sources by using their native query language. This reduces the costs of data conversion but requires query translation. The source selection approach relies on mapping definitions and an index of predicates with the associated data sources. For query optimization PolyWeb uses the same cost-base model as FedX and *predicate-based join groups* to reduce the number of local joins. All plans generated by this approach are left-deep trees with nested loop joins.

Several query processing engines have been proposed, but most of the query engines focus on a single data model, like RDF. Optimizing query execution plans for a single model has been studied extensively in the past, e.g. ANAPSID introduced adaptive operators for continuous answer generation and MULDER proposed RDF-MT based source descriptions to guide query processing. Heuristics for RDF query engines mainly focus on the selectivity of sub-queries and triple ordering. When dealing with heterogeneous sources, many systems integrate the data of the federation at one site in a common data format. For very big data integrating the data might not be feasible. Therefore, Ontario and PolyWeb keep the sources in their original format and query them using their native query language. The proposed approach is based on Ontario and, therefore, addresses query processing over Semantic Data Lakes. This thesis aims to improve the query performance of Ontario over heterogeneous data sources by providing source specific heuristics to guide the optimizer. These heuristics are tailored for the different capabilities of the varying data formats in the Data Lake. The novelty of the proposed approach is the use of source specific heuristics when dealing with heterogeneous data sources in their original format.

Chapter 4

Approach

Query optimization is often lead by heuristics to decrease the time complexity. The proposed approach in this thesis builds bushy tree plans following the algorithm of Ontario [12] with two different sets of heuristics: **(i)** general heuristics that do not consider the data format, and **(ii)** source specific heuristics that consider the data format. The second approach also includes the heuristics of the first one. Next, the research problem of this thesis is explained in more detail. Afterwards, the tree generation algorithm and the different sets of heuristics are described.

4.1 Problem Statement

The problem addressed in this thesis uses previously discussed concepts. Nevertheless, the most important concepts are formally defined in order to understand the problem statement.

Definition 8 (RDF Molecule Template [12]). *An RDF Molecule Template is a 5-tuple* $\sigma = \langle S, C, \gamma, IntraL, InterL \rangle$*, where:*

- *S - an interface to access data set G;*
- *C - an RDF class such that the triple pattern (?s rdf:type C) is true in G;*
- γ *- a set of pairs (p, T) such that p is a property with domain C and range T, and the triple patterns (?s p ?o), and (?s rdf:type C) are true in G;*
- $IntraL$ *- a set of pairs (p,*C_j*) such that p is an object property with domain C and range* C_j*, and the triple patterns (?s p ?o), (?o rdf:type* C_j*), and (?s rdf:type C) are true in G;*
- $InterL$ *- a set of triples (p,* C_k*, SW) such that p is an object property with domain C and range* C_k*, SW is a Web service API that provides access to an*

RDF data set K, the triple patterns (?s p ?o), and (?s rdf:type C) are true in G, and the triple pattern (?o rdf:type C_k) is true in K

Definition 9 (Semantic Data Lake [12]). *A Semantic Data Lake (SDL) is a triple $SDL = \langle \Psi, \mathbb{S}, M \rangle$, where Ψ a set of RDF Molecule Templates, $\mathbb{S}$ a set of sources in raw formats (stored either in a file system or DBMS) in the Data Lake, M a set of conjunctive rules that associate sources in $\mathbb{S}$ with RDF Molecule Templates in Ψ.*

Definition 10 (Instantiation of an RDF-MT [12]). *An Instantiation of an RDF-MT, $[\sigma]$, is defined as a set of RDF molecules, σ^*, that are the instances of a class from data source(s) as described in the template:*

$$[\sigma] = \{\sigma^* | \forall p \in \sigma^*, p \subseteq \gamma, where\ \gamma \subseteq \sigma\} \tag{5}$$

Definition 11 (Virtual Knowledge Graph [12]). *Given a Semantic Data Lake $SDL = \langle \Psi = \{\sigma_1, \ldots, \sigma_k\}, \mathbb{S} = \{S_1, \ldots, S_n\}, M \rangle$, a Virtual Knowledge Graph (KG^*) for the SDL is a virtual RDF graph that corresponds to the union of all the instantiations of RDF-MTs, σ^*, that are created by applying the rules in M to the data sources in $\mathbb{S}$:*

$$KG^* = \bigcup_{i=1}^{n} \bigcup_{j=1}^{k} [\sigma_j]_{S_i} \tag{6}$$

The SPARQL queries need to be rewritten into queries on the original data sources in order to query the Virtual Knowledge Graph. Like already discussed in Subsection 2.1.3 the SPARQL query language is based on matching graph patterns. A Basic Graph Pattern (BGP) is a set of triple patterns and (optional) filter clauses.

Definition 12 (Basic Graph Pattern [13]). *Let I be the set of all URIs, B the set of blank nodes, L the set of literals and ϵ the set of variables. A SPARQL Basic Graph Pattern (BGP) expression is defined recursively as follows:*

1. *A triple pattern $\tau \in (I \cup B \cup \epsilon) \times (I \cup \epsilon) \times (I \cup B \cup L \cup \epsilon)$ is a BGP;*
2. *The expression (P FILTER E) is a BGP, where P is a BGP and E is a SPARQL filter expression that evaluates to a Boolean value;*
3. *The expression (P1 AND P2) is a BGP, where P1 and P2 are BGPs.*

According to the definition a BGP contains at least one star-shaped sub-query (SSQ), a non-empty set of triples that share the same subject variable or constant.

Definition 13 (Star-Shaped Sub-Query [45]). *A star-shaped sub-query* `SSQ(S, ?X)` *on a variable (or constant)* `?X` *is defined as:*

- `SSQ(S, ?X)` *is a triple pattern (*`?X p o`*), and* `p` *and* `p` *are different from* `?X`*.*
- `SSQ(S, ?X)` *is the union of two stars,* `SSQ(S1, ?X)` *and* `SSQ(S2, ?X)`*, where triple patterns in* `S1` *and* `S2` *only share the variable (or constant)* `?X`*.*

Definition 14 (Query Rewriting [12]). *Let Q a SPARQL query, $\beta(Q)$ the set of Basic Graph Patterns (BGPs) in Q, and $SQL = \langle \Psi, \mathbb{S}, M \rangle$ a Semantic Data Lake. A rewriting Q' of Q over the sources in $\mathbb{S}$ corresponds to a SPARQL query composed of BGPs in $\beta(Q')$ that meet the following conditions [12]:*

- *$\beta(Q)$ has the same number of triple patterns as $\beta(Q')$, i.e., $\tau(Q) = \tau(Q')$*
- *there is a function $\mu : \beta(Q) \rightarrow \beta(Q')$ that maps BGPs in $\beta(Q)$ to its corresponding rewriting in the sources of the SDL.*

$$\mu\langle BGP_i \rangle = \{\langle BGP_{ij}, S \rangle | BGP_{ij} \subset BGP_i, S \text{ a non-empty set and } S \subset \mathbb{S}\} \quad (7)$$

The problem addressed by this thesis is the problem of query rewriting in a federation of heterogeneous data sources. The federation is modeled as a Semantic Data Lake (SDL). Given a SPARQL query Q, a Semantic Data Lake $SDL = \langle \Psi, \mathbb{S}, M \rangle$, a Virtual Knowledge Graph KG^* of the SDL, and a set of BGPs in Q, the problem can be defined as find a query Q' that satisfies the following conditions [12]:

- The evaluation of Q over heterogeneous data sources in SDL is complete, i.e., the evaluation of Q in KG^* is equivalent to the evaluation of Q' in SQL:

$$[[Q']]_{SDL} = [[Q]]_{KG^*} \quad (8)$$

- The cost of executing Q in SDL has a minimal execution cost, i.e., if $cost([[Q']]_{SDL})$ represents the execution time of Q' in SDL, then

$$[[Q]]_{SDL} = \underset{[[Q']]_{SDL}}{\operatorname{argmin}} \, cost([[Q']]_{SDL}) \quad (9)$$

4.2 Bushy Tree Generation

The different sets of heuristics are integrated into the bushy tree generation of Ontario which is presented in Algorithm 4.1. Given a list of star-shaped sub-queries Φ and the actual query Q this algorithm produces a bushy tree α. First, the triples of each SSQ are sorted (line 3). Afterwards, the list of SSQs is sorted (line 7). Both sortings are based on the selectivity of the triples or SSQs, respectively, i.e., the first triple or SSQ is more selective than the second one and so on. In order to create a tree from the sorted list of SSQs they have to be joined. Two SSQs are joinable if they share at least one variable. Starting with one SSQ (line 9) the algorithm creates a

list of all SSQs that are joinable (line 11). The list of joinable SSQs is sorted (line 12) according to the same rules as before. The first element of this list is joined with the starting SSQ (line 14). Afterwards, the joined SSQ is removed from the list of SSQs and the join is inserted instead. If no joinable SSQ was found, the starting SSQ is added to a list of yet unjoined SSQs. This procedure is repeated until there is only one element left in the list of SSQs. This element no longer represents a single SSQ but a tree composed of the generated joins. If there are unjoined SSQs they are joined with the tree (line 24). In the end, the list of SSQs has become an optimized bushy tree composed of all sub-queries.

Algorithm 4.1 Bushy Tree Generation

Input: Φ: List of star-shaped sub-queries; Q: SELECT query
Output: α: bushy tree

```
1:  α ← []
2:  for SSQ ∈ Φ do
3:      orderTriples(SSQ)
4:      α.push(SSQ)
5:  end for
6:  P ← Q.projs()                    ▷ Q.projs(): list of join and projection variables
7:  α ← OrderSSQs(α, P)
8:  while len(α) > 1 do
9:      SSQ_i ← α.pop()
10:     δ ← []
11:     β ← [SSQ_j for SSQ_j ∈ α if shareVars(SSQ_i, SSQ_j)]
12:     β ← OrderSSQs(β, P)
13:     for SSQ_j ∈ β do
14:         J ← join(SSQ_i, SSQ_j)
15:         α.remove(SSQ_j)
16:         α.push(J)
17:         break
18:     end for
19:     if |β| = 0 then
20:         δ.push(SSQ_i)
21:     end if
22: end while
23: if len(δ) > 0 then
24:     α ← join(α, δ)
25: end if
26: return α
```

4.3 General Heuristics

The *general heuristics approach* makes use of heuristics that do not consider the data format of the source. Therefore, these heuristics are considered to be general. The join ordering is based on the selectivity of the sub-query. Just like in Ontario, a nested loop join is used, if possible. The heuristics are part of the sorting methods of the query optimizer depicted in Algorithm 4.1. This approach considers the following heuristics in order to create the query execution plan.

Heuristic 1 (Percentage of Constants). *Triple patterns with the highest number of constants at any part of the triple pattern are more selective.*

This is quite intuitive, e.g., in a relational database a query with a higher number of bound attributes will more likely return only a few results, hence it is more selective than a query with less constants. Since this is true for any kind of data source this heuristic is considered to be general.

Heuristic 2 (Triple Ordering). *A set of triple patterns in a sub-query can be ordered as follows: spo > sp > s > po > o > p. That is, a triple pattern with constants in all parts precedes a triple pattern with constants in subject and predicate. Similarly, a triple pattern with constants at subject and predicate precedes a triple pattern with a constant at subject part only, and so on.*

The triple ordering heuristic was proposed in [43] for SPARQL endpoints. Even though this heuristic can be categorized as general since depending on the capabilities of the query optimizer of a data source it might help to put the most selective triple patterns at the beginning of a query. This is especially true for data sources that do not reorder the triples, but execute them in the exact same order as they were posted in the original query.

Heuristic 3 (Number of Projected Variables in Sub-Queries). *Given two SSQs as potential inner part of a join with the same number of constants and variables, the SSQ with fewer projections is chosen.*

The SSQ with fewer projected variables is executed first because it only contains a subset of the variables contained in the other SSQ. Therefore, choosing the SSQ with fewer projections leads to a smaller set of instantiations that will be joined with the outer join operand. As the result is smaller the intermediate result of the join might also be smaller in comparison to the join with the other SSQ.

4.4 Source Specific Heuristics

The source specific heuristics used in this approach are based on the findings made during the development of the paper [12] preceding this thesis. The following categories of SSQs can be identified: **(CI)** SSQs with neither constant object nor filter clause on object variables, **(CII)** SSQs with neither constant object nor filter clause on object variables and distributed over multiple relations in the Semantic Data Lake, **(CIII)** SSQs with at least one constant object or filter clause on object variables, and **(CIV)** SSQs with at least one constant object or filter clause on object variables and distributed over multiple relations in the Semantic Data Lake.

The tree generation algorithm depicted in Algorithm 4.1 is modified to also compute the category of each SSQ (Algorithm 4.2). First, the number of constant objects and presence of a filter condition on an object of the SSQ is tested (line 2). Second, the number of relations covered in the data source is checked (line 6). Afterwards, the category is set according to the definitions above (lines 9-16). The optimizer considers the following heuristics in addition to the general ones.

Algorithm 4.2 Sub-Query Category Computation

Input: SSQ: star-shaped sub-query; Q: SELECT query; DS: data source
Output: cat: the category of SSQ

```
 1: hasInstantiation ← False
 2: if SSQ.constantObjects() > 0 or objectFilter(SSQ, Q) then
 3:     hasInstantiation ← True
 4: end if
 5: multipleRelations ← False
 6: if numberRelations(SSQ, DS) > 1 then
 7:     multipleRelations ← True
 8: end if
 9: if not hasInstantiation and not multipleRelations then
10:     cat ← CI
11: else if not hasInstantiation and multipleRelations then
12:     cat ← CII
13: else if hasInstantiation and not multipleRelations then
14:     cat ← CIII
15: else
16:     cat ← CIV                          ▷ hasInstantiation and multipleRelations
17: end if
18: return cat
```

Heuristic 4 (Pushing down instantiations into a star-shaped sub-query). [12] *Given a star-shaped sub-query SSQ_i from category CI executed over an RDF engine, if this SSQ is part of a join, it is used as inner sub-query of a nested loop join. Push down filter expressions of Q if they contain variables from SSQ_i. This transforms sub-queries from CI into sub-queries from CIII.*

By using SSQ_i as the inner part of a nested loop join the join variable is bound to the results from the outer operand. Therefore, the intermediate result may be smaller than the complete result of SSQ_i with an unbound join variable. Pushing down filter expressions of Q might further reduce the size of intermediate results and can be evaluated in a reasonable time by RDF engines.

Heuristic 5 (Breaking up joins in star-shaped sub-queries). [12] *Given a star-shaped sub-query SSQ_i from category CII executed over an RDF engine, the SSQ is divided into as many sub-queries as joins are defined in the corresponding RDF-MT and the attributes used in SSQ_i. These sub-queries are connected by nested loop join operators executed at Ontario level. Thus this heuristic transforms sub-queries from CII into sub-queries from CIV.*

The SSQ is split in several sub-queries joined with a nested loop join in order to create instantiations. The filter expressions created from the bound join variable can be efficiently evaluated in SPARQL endpoints. Instantiations are likely to reduce the number of results returned by the sub-query. Therefore, intermediate results can be reduced.

Heuristic 6 (Pushing up instantiations into a star-shaped sub-queries). [12] *Given a star-shaped sub-query SSQ_i from category CIII or CIV executed over an RDB engine with an instantiation over an unindexed attribute, the filter is executed on Ontario level. If the SSQ is part of a join in Q a hash join or GJoin [2] is used. This rule transforms sub-queries from CIII and CIV into sub-queries from CI and CII, respectively.*

Filtering unindexed attributes at the level of the RDB engine is expensive. Therefore, it is considered to transfer a larger intermediate result and perform the filtering at Ontario level. Afterwards, the filtered result is joined using a symmetric join approach. Hence, the SSQ can be either outer or inner join operand.

Heuristic 7 (Combining joins into a star-shaped sub-query). [12] *Given two star-shaped sub-queries SSQ_i and SSQ_j from category CI executed over an RDB engine, if they can be evaluated at the same endpoint and can be joined on an indexed attribute, merge SSQ_i and SSQ_j into one star-shaped group $SSQ_{i,j}$. This transforms sub-queries from CI into sub-queries from CII.*

Joins on indexed attributes in RDB engines are considered to be fast as long as the number of joins is kept reasonable. Since both SSQs are described in terms of a single relation and a join might reduce the intermediate result size, the join of the two SSQs is pushed down to the RDB engine.

Heuristic 8 (Pushing down SPARQL joins). *Given two star-shaped sub-queries SSQ_i and SSQ_j executed over an RDF engine, if they share at least one variable and can be evaluated at the same endpoint, merge them into one star-shaped sub-query $SSQ_{i,j}$. This transforms sub-queries from CI and CIII into sub-queries from CII and CIV, respectively.*

This heuristic was already used by the original optimization of Ontario. Since SPARQL endpoints have indexes over the subjects, predicates, and objects as well as the combinations of them, pushing down the join relieves the mediator from performing the join. Depending on the selectivity pushing down the join might also reduce the intermediate result.

This thesis addresses the problem of query rewriting in a federation of heterogeneous sources. Bushy trees are generated in order to increase the parallelism during query execution. The proposed approach comprises of two different sets of heuristics to follow during query optimization. The first set contains general heuristics only, i.e., heuristics that do not consider the type of the data source. The second set of heuristics is source specific, i.e., heuristics that do consider the data source type for optimization. The set of source specific heuristics includes the general heuristics as well as heuristics tailored for RDF and RDB, respectively. The proposed approach is implemented on top of Ontario.

Chapter 5

Implementation

The proposed approach is implemented on top of Ontario, a federated query engine that is able to execute queries over heterogeneous data sources. Ontario is implemented in Python 3.6. The heuristics are integrated into the query optimization of the current version of Ontario. Additionally, the SQL wrappers are changed due to problems addressed later. This chapter is organized as follows: In the beginning, the implementation of Ontario query processing is discussed which includes query decomposition and query optimization. Finally, the changes made to the existing SQL wrappers are presented in detail.

5.1 Query Processing

Query processing includes *query decomposition*, *source selection*, *query optimization*, and *result retrieval*. The latter is not discussed here since it is not part of the subject of this thesis, query optimization. Anyhow, query decomposition and source selection are covered because they are important for the query optimization. The goal of this section is to give an overview of the implementation of Ontario's query processing. First, the implementation of query decomposition is discussed. Ontario performs decomposition and source selection at the same time. This is possible due to the use of RDF-MT based source descriptions. Afterwards, the query optimizer implemented in Ontario is presented. This is where the proposed heuristics are integrated. Finally, further remarks to the implementation are made.

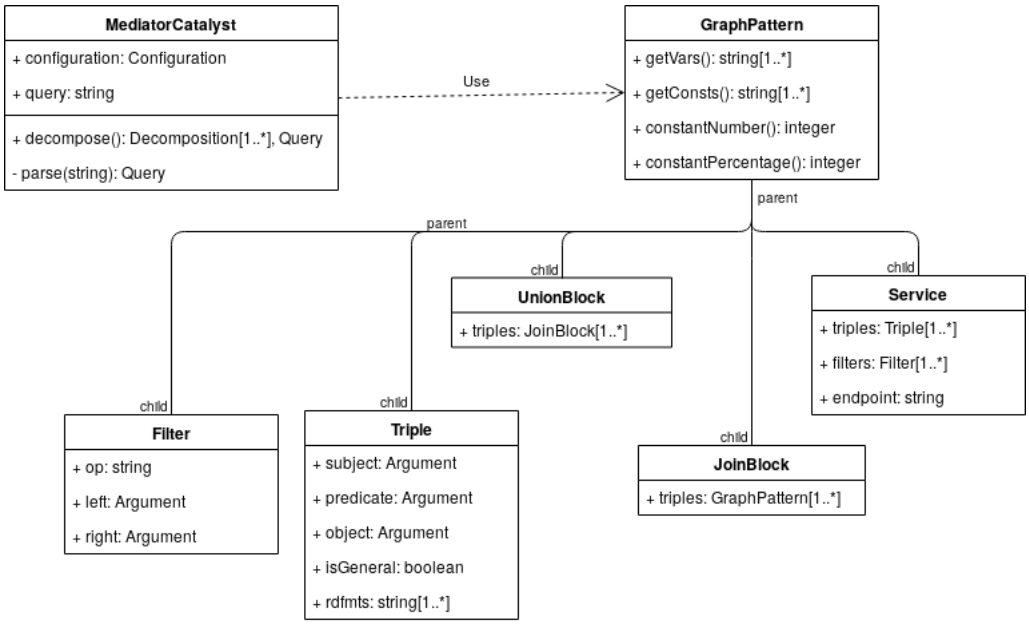


Figure 5.1: **Implementation Query Decomposition**

5.1.1 Query Decomposition

A SPARQL query is decomposed into star-shaped sub-queries that can be answered by the data sources. The decomposition is guided by RDF-MT based source descriptions which allow decomposition and source selection at the same time. The actual implementation is depicted in Figure 5.1. First, the different implementations of graph patterns are discussed in order to understand the decomposition procedure of the `MediatorCatalyst` which is studied afterwards.

Graph Patterns Ontario implements different kinds of graph patterns. The base class `GraphPattern` provides methods for basic interactions which have to be implemented by the sub-classes. The method `getVars()` returns a list of all variables occurring in the triple patterns of the graph pattern represented as string. Similarly, `getConsts()` returns a list of all constants of the triple patterns. The methods `constantNumber()` and `constantPercentage()` are used for estimating the selectivity of a graph pattern. They return the number of constants and the percentage of constants in a graph pattern, respectively. Each triple has three places, subject, predicate, and object. Therefore, the percentage of constants is the number of constants divided by three times the number of triples in the graph pattern.

The results of graph patterns within a `UnionBlock` are combined to form the union of the individual results. The graph patterns contained in a `UnionBlock` are always `JoinBlocks`. `JoinBlocks` can contain any type of graph patterns. The graph patterns of a `JoinBlock` need to be joined. A `Service` represents a star-shaped subquery that can be answered by the data source `endpoint`. `Services` are composed of lists of `Triples` and `Filters`. `Triples` represent actual RDF triples. The subject, predicate, and object are stored as `Argument`, a datatype comprised of the variable or constant as string, a boolean indicating whether or not the argument is a constant as well as a boolean to indicate if the argument is an URI or literal. Additionally, a triple is annotated with a boolean to show whether or not it is general, i.e., using a general predicate like `rdf:type` or `rdfs:label` as well as the RDF-MTs that are able to answer this triple. `Filters` are used to evaluate restrictions on certain variables. They are composed of the left and right operand as well as the operator used, e.g., equal or less.

Mediator Catalyst First, the `MediatorCatalyst` parses the query string. The query parser returns a `Query` object which contains a `UnionBlock` comprised of further `JoinBlocks`. Those graph patterns are based on the stars of the query, i.e., sub-queries with the same subject. The decomposition starts with the main `UnionBlock`. Every `JoinBlock` contained in the `UnionBlock` is decomposed. If one of the graph patterns is a `UnionBlock` or a `JoinBlock` it is decomposed according to its type. `Triples` or `Filters` are added to a triple list and filter list, respectively. In this case a BGP composed of at least one star-shaped group is decomposed. The following information is collected during the decomposition of a BGP, the predicates, the stars themselves, connections between the stars, the RDF-MTs able to answer the stars, and the connections between the RDF-MTs. All predicates occurring in the stars are stored in a list. A star itself is a more complex structure. It is comprised of the common subject of the triple patterns as identifier, a list of all the triples, a list of the RDF-MTs that are able to answer the triple patterns of the star, a Python dictionary for mapping the predicates to object variables, and data sources that contain the RDF-MTs. Therefore, source selection is done during the decomposition based on the RDF-MT based source descriptions. The connections to other stars indicate which stars can be joined. Additionally, the join position is stored, i.e., a join over the subject or an object. The connection between RDF-MTs shows that the range of at least one of the predicates of a star belonging to the RDF-MT is from another RDF-MT. Those connections can be used for joins. In the end, all the collected information about the stars is returned along with the `Query` object. The nested structure of `UnionBlocks` and `JoinBlocks` is preserved.

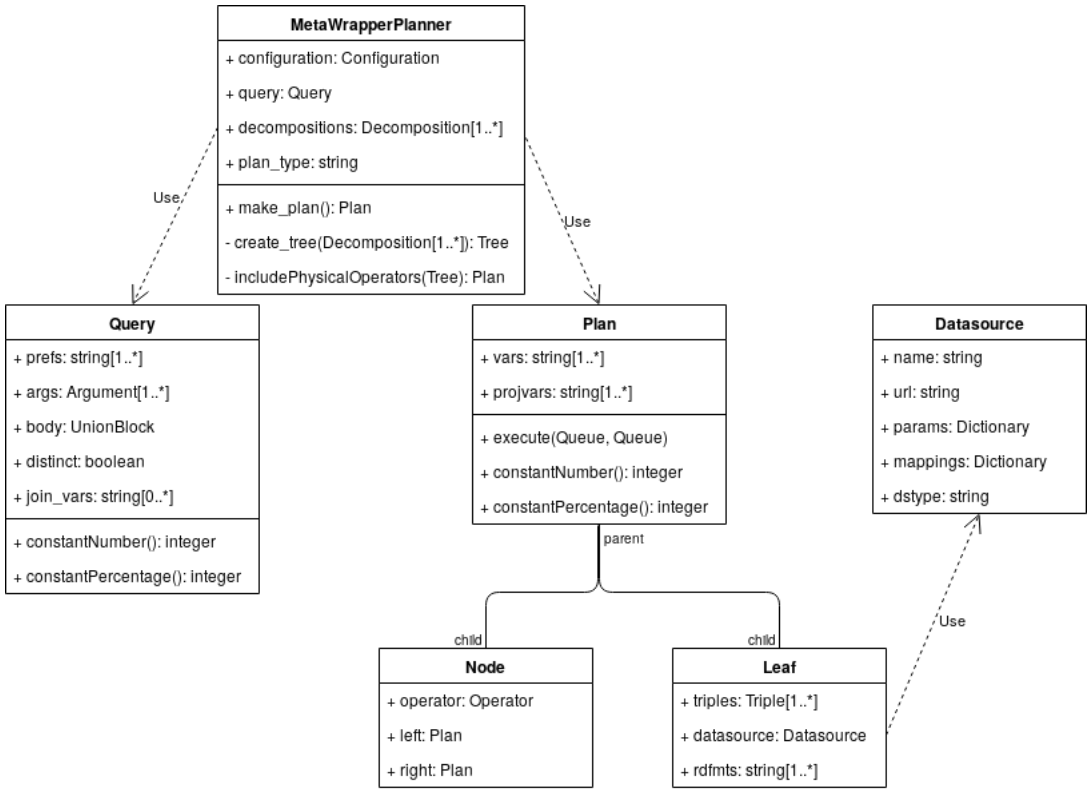


Figure 5.2: **Implementation Query Optimizer**

5.1.2 Query Optimizer

The query optimizer of Ontario is called `MetaWrapperPlanner`. It creates an optimized query execution plan from the decomposed query. Additionally, the representation of the query itself is used, e.g., the `Query` object is needed to add the duplicate elimination operator if necessary. Furthermore, it is used get the prefixes which have to be send to the sources. The optimization is guided by different heuristics. The plan generation is realized in two steps each one guided by heuristics: **(i)** create an optimized query tree from the decomposition, and **(ii)** include physical operators in that tree in order to produce an optimized query execution plan. The implementation is shown in Figure 5.2. First, the representation of a query is discussed. Second, the implementation of data sources is explained. Afterwards, the query plan is studied. The tree structure is not examined because it is an intermediate result and does not differ much from the structure of the query plan. Finally, the query optimizer is presented in detail.

Query As previously discussed, the representation of a query contains a `UnionBlock` comprised of `JoinBlocks`. More important at this stage of query processing are the other attributes of a `Query` object. The prefixes of a query are stored as a list of strings (`prefs`). The attribute `args` is a list of the projected variables of the query, i.e., the variables to return. `Distinct` is a boolean indicating whether or not the result of the query may contain duplicates. If it is set to `true`, duplicate elimination has to be performed. The join variables, i.e., the variables used to join the sub-queries, are stored as a list called `join_vars`. The methods `constantNumber()` and `constantPercentage()` work as described before and they are calling the same function of the `UnionBlock`.

Data Source Data sources are represented as objects of the class `Datasource`. Each data source has a name that serves as identifier. Hence, two different data sources in the same federation cannot have the same name. The address of the data source is stored in the `url` attribute. The attribute `params` is a Python dictionary holding connection parameters, e.g., user name and password for MySQL sources. Non-RDF sources are associated with mappings in order to add semantics to the data. The mappings are saved in a Python dictionary named `mappings`. This dictionary is created from parsing the RML mappings (cf. Subsection 2.1.4) of the RDF-MTs belonging to the data source. Additionally, the type of the data source is stored as string, e.g. *SPARQL_Endpoint* and *MySQL* for RDF and MySQL sources, respectively. The type of the source is needed for source specific heuristics.

Plan A `Plan` is an abstract class providing attributes and methods the nodes and leafs of the plan have in common. As for `Query`, the methods `constantNumber()` and `constantPercentage()` call the related method of the underlying graph pattern. The method `execute()` is used to process the operation represented by the node. The attribute `vars` stores a list of the variables occurring in the child nodes. If the node is a leaf, the list contains the variables of the triples belonging to the leaf. The variables projected are saved in a list called `projvars`. A plan is represented by its root. Hence, in most cases it is a `Node`. The nodes are an operation performed at Ontario level, e.g., join, projection, and duplicate elimination. Which operation is to be executed at the node is saved in the `operator` attribute. The attributes `left` and `right` hold the left and right operand, respectivley. For unary operations the right operand is simply left empty. `Leafs` represent the star-shaped sub-queries that can be executed at a data source. Therefore, they contain a list of triples that are evaluated at a specific source. The data source in question is also stored at the leaf. A list of RDF-MTs that are associated with the triples of the leaf is stored as well.

Meta Wrapper Planner The `MetaWrapperPlanner` is the query optimizer of Ontario. It creates an optimized query execution plan in two steps guided by heuristics: **(i)** create an optimized tree from the decomposition, and **(ii)** include physical operators into that tree in order to retrieve an optimized query plan. Therefore, the query execution plan is optimized during its generation. The `configuration` basically stores every information that might be needed for query processing, e.g., in the case of query optimization the indexed attributes of RDB sources. As discussed before, the query representation is needed for query planning, e.g., to include the duplicate elimination operator. The result of the query decomposition is stored in an attribute `decompositions`. The query optimizer is able to include different heuristics based on the specified `plan_type`, i.e., if the heuristics of the original implementation are used, the general heuristics or the source specific heuristics.

Ontario calls the method `make_plan()` to retrieve the optimized query execution plan. First, a tree is created from the decomposition by an internal call to `create_tree(Decomposition)`. This method is an implementation of the tree generation algorithm presented in Algorithm 4.1. The star-shaped sub-queries are taken from the decompositions. If necessary, the category of each SSQ is computed, i.e., if source specific heuristics are considered. Each SSQ is converted to an internal representation of a tree leaf. The leafs are sorted according to the heuristics based on selectivity. Which heuristics are considered depends on the given plan type. While there is more than one node, i.e., the structure is not yet a single tree, leafs that share at least one variable are joined, i.e., a node is created with the two leafs as left and right child, respectively. At this point Heuristic 7 can be checked. If all conditions are met, both leaves are merged into a single leaf. Hence, the join is pushed down to the source. The same approach is used for Heuristic 8 during the decomposition of a BGP. Once there is only one node remaining, i.e., the root of a proper tree, SSQs are joined with the root that could not be joined before. Afterwards, the method `includePhysicalOperators(Tree)` is called to transform the tree into a query execution plan. Since the nodes and leaves of the tree can be interpreted as graph patterns, a main `UnionBlock` is added that contains the tree and, when including the physical operators, follows a similar approach as the query decomposition. If there is more than one graph pattern in a `UnionBlock`, the union operator is added to connect them. `JoinBlocks` are connected with a join operator. The actual implementation of the join depends on the used heuristics and, therefore, it is also based on the selectivity of the operands. Other operators like filters are added according to the operator referenced in a node. Finally, operators like projection and duplicate elimination are added to the plan if necessary. Afterwards, the generation of an optimized query execution plan is finished and the plan is returned.

5.1.3 Remarks

One change to the original implementation of Ontario, i.e., the version of Ontario before the work on this thesis started, is that the projection operation is only carried out if necessary. The original implementation always included a projection. Since this operation adds overhead to the query execution it is only added if needed, i.e., if not all variables returned from the last operation are projected variables. In order to decide whether or not to use the heuristics defined for RDB sources the optimizer has to know the indexes. A hand-made JSON file containing all relevant indexes is used for this purpose. Finding indexes in RDB sources may be integrated into the RDF-MT extraction and, therefore, into the RDF-MT based source descriptions, but this is out of the scope of this thesis.

5.2 SQL Wrapper

The original implementation of Ontario includes SQL wrappers with a two phased result generation: **(i)** retrieve 10,000 results from the data source, **(ii)** semantify the retrieved results. Due to these two phases, retrieving the next results is blocked by the semantification process. Additionally, the query is modified using `LIMIT` and `OFFSET` to retrieve 10,000 results only. Hence, the query is executed several times depending on the total number of results. In order to allow for continuous result retrieval the semantification is done during query translation. Thus a query returning the title of all movies in the database is translated to `SELECT CONCAT("http://example.org/", title) AS title FROM movies` instead of the non-semantified query `SELECT title FROM movies`, which eliminates the second phase.

Executing the query several times introduces overhead in step one. Since the query has to be executed ten times if the result size is between 90,001 and 100,000 this will take longer than running the query once and retrieving the results afterwards. Ontario uses the `mysql-connector-python` library to access MySQL databases. The library offers different ways of retrieving the results, i.e., retrieving the results row by row or in blocks of fixed size. The latter one is not working from version 8.0.12 onwards. There is a bug that permits the result retrieval to finish because the very last result will be repeated infinitely. At the point of writing, the current version is 8.0.16; therefore, Ontario is implemented on version 8.0.11. To solve the aforementioned problem of executing the query several times the new wrapper implementation executes the query once and retrieves the results row by row. The new wrappers enable an easy change of this behavior because the result retrieval is implemented for all possibilities and is defined by a parameter of the function.

The changes to the wrappers are not limited to retrieving the results from the data sources. However, the translation from SPARQL to SQL queries is extended to enable translations of SPARQL queries composed of triple patterns from more than one RDF-MT. Pushing down joins of certain SSQs in SQL is suggested by Heuristic 7. Algorithm 5.1 presents the theoretical query translation. The current implementation is not able to deal with queries containing more than two RDF-MTs. The query translation operates on the RDF-MTs of the SSQ that are covered by the data source (line 1). In the simplest case the query contains triple patterns from one RDF-MT only. In that case the query can be translated with respect to the mappings of that RDF-MT (line 3). The translation function works on the query to translate and the RDF-MT to be respected. The mappings needed to be able to translate the query to SQL (cf. Subsection 2.1.4) are directly accessible from the translation function. Several cases have to be considered when more than one RDF-MT have to be respected. **(i)** If the RDF-MTs do not share predicates used in the query (line 6), the triples belonging to one RDF-MT (line 8) can be translated with respect to that RDF-MT (line 9). Afterwards, the translations are joined (line 11). This approach creates SQL queries with sub-queries in the `FROM` clause, e.g. `SELECT A.b, B.d FROM (SELECT a, b FROM r1) AS A JOIN (SELECT c, d FROM r2) AS B ON (A.a = B.c)`. **(ii)** If all predicates used in the query are shared between the RDF-MTs (line 12), the query has to be translated with respect to all RDF-MTs. For each RDF-MT the query is translated (line 14) and finally all translations are combined with the `UNION` operator (line 16). **(iii)** The most complex case includes some predicates that can be answered by more than one RDF-MT but also triples that can be exclusively answered by one RDF-MT (lines 17-28). Algorithm 5.1 shows an abstract representation of this case. Pairwise intersections are needed in order to generate a correct translation. This case is not yet implemented. All triples that can be exclusively answered by one RDF-MT are translated in the same manner as in the other cases (line 24). Triples containing a predicate that can be answered by more than one RDF-MT need to be stored separately (line 25). After the translations are finished, the sub-queries sharing triples are unified following the previously mentioned method. Finally, all sub-queries are joined.

The proposed heuristics are integrated into the query optimizer of Ontario. First, the query is parsed and decomposed. The source selection is guided by the RDF-MT based source descriptions and integrated into the query decomposition. Next, the query is optimized following the heuristics enabled by a flag indicating which set of heuristics should be used. The decomposed query is transformed into a bushy tree representing the order of execution. Afterwards, the physical operators are included into that tree. Depending on the operator and heuristics the order of the operands

may change. After all operators are included, the optimized query execution plan is returned. The SQL wrappers are updated in order to allow semantification during the query translation. The SPARQL to SQL translation is also extended to enable translations of queries composed of more than one RDF-MT. The proposed approach and its implementation is empirically evaluated in the next chapter.

Algorithm 5.1 Query Translation

Input: SSQ: star-shaped sub-query; DS: data source
Output: SQL query translated from SPARQL

```
1: rdfmts ← SSQ.rdfmts() ∩ DS.rdfmts()
2: if len(rdfmts) = 1 then
3:     return translate(SSQ, rdfmts[0])
4: else
5:     translations ← []
6:     if intersection_of_predicates(rdfmts) = ∅ then
7:         for rdfmt ∈ rdfmts do
8:             triples ← triples_of_rdfmt(SSQ, rdfmt)
9:             translations.append(translate(triples, rdfmt))
10:        end for
11:        return join(translations)
12:    else if all_rdfmts_same_predicates() then
13:        for rdfmt ∈ rdfmts do
14:            translations.append(translate(SSQ, rdfmt))
15:        end for
16:        return union(translations)
17:    else        ▷ abstract representation of this case; pairwise intersections are needed
18:        triples_intersection ← triples_of_intersection(SSQ, rdfmts)
19:        translations_union ← []
20:        for rdfmt ∈ rdfmts do
21:            triples ← triples_of_rdfmt(SSQ, rdfmt)
22:            triples_union ← triples ∩ triples_intersection
23:            triples ← triples \ triples_union
24:            translations.append(translate(triples, rdfmt))
25:            translations_union.append(translate(triples_union, rdfmt))
26:        end for
27:        translations.append(union(translations_union))
28:        return join(translations)
29:    end if
30: end if
```

Chapter 6

Experimental Evaluation

The experimental evaluation aims to measure the performance of query processing over heterogeneous data sources. The research questions addressed by this thesis are: **RQ1)** What is the overhead of considering heterogeneity during federated query processing? **RQ2)** Can RDF-MT based source descriptions be effectively and efficiently applied for source selection, query decomposition, and optimization for non-RDF data sources? **RQ3)** Are Ontario optimization techniques able to generate effective and efficient query plans for heterogeneous data sources? **RQ4)** Do the proposed heuristics improve query performance? The remainder of this chapter is structured as follows: First, the used benchmark is described. Second, the data preparation is presented. Afterwards, the setup of the experiment is depicted. Finally, the results are shown and analyzed.

6.1 LSLOD Benchmark

There are many data sets on the Web that can be queried. For example the *Linked Open Data* (LOD) cloud. The LOD cloud contains more than 1200 different data sets with more than 16000 links [29]. Some of the data sets are of general interest like *DBPedia* or *Wikidata*, but there are also domain specific data sets. Hence, the LOD cloud contains subclouds of certain domains. One of these subclouds is of the Life Science domain, referred to as *Life Science Linked Open Data* (LSLOD) cloud. There is an existing set of benchmarking queries for this subcloud [21]. These queries are supposed to simulate frequently used queries in the Life Science domain.

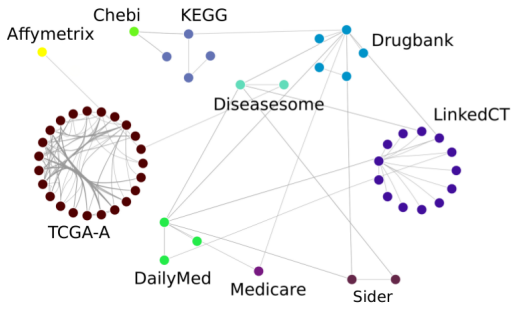


Figure 6.1: **LSLOD Molecule Template Connections** (adapted from [13])

6.1.1 LSLOD Data Sets

The LSLOD cloud contains ten data sets, namely Affymetrix, ChEBI, DailyMed, Diseasome, DrugBank, KEGG, LinkedCT, Medicare, SIDER, and TCGA-A. Affymetrix contains the probesets used in the Affymetrix microarrays. ChEBI is the *Chemical Entities of biological Interest* data set. Dailymed provides information on marketed drugs along with additional information like chemical structure and indication. Diseasome publishes data that allows to find the common genetic origin of many diseases. DrugBank contains information on drugs, their composition and their interactions with other drugs. The *Kyoto Encyclopedia of Genes and Genomes* (KEGG) provides further information about chemical compounds and reactions focusing on information relevant for genetics. LinkedCT publishes clinical trials. In this benchmark Medicare data includes drugs only. The drugs from Medicare are linked to drugs in Dailymed. SIDER contains information on marketed drugs and their adverse effects. TCGA-A is a subset of the *Cancer Genome Atlas* (TCGA) data and contains methylation of exons as well as a large number of links to Affymetrix.

Figure 6.1 shows the intra- and inter-data set connections. TCGA-A has a high intraconnectivity. Even though it was mentioned that TCGA-A has a large number of links to Affymetrix, these links are between two RDF-MTs only. Overall the data sets of the LSLOD cloud contain 55 RDF-MTs. Table 6.1 presents information about the different data sets, like number of triples, number of distinct subjects, predicates, and objects, number of RDF-MTs, and the actual RDF file size. Affymetrix is the largest data set in number of triples even though it covers only one RDF-MT. TCGA-A has the most RDF-MTs and is the largest data set in terms of RDF file size and also the second largest in number of triples. State-of-the-art query engines are compared using the LSLOD benchmark. In order to compare different tools and enable reproducibility this benchmark is used. The variety of data sets also allows the natural implementation of the benchmarks in varying data models.

Dataset	#tp	#s	#p	#o	#MTs	$RDF
Affymetrix	44.20M	1.42M	41	13.24M	1	6.4 GB
ChEBI	4.77M	50.48K	28	772.14M	1	539 MB
DailyMed	162.92K	9,975	27	67.77K	3	110 MB
Diseasome	72.42K	8,132	27	27.70K	2	14 MB
DrugBank	516.89K	19.57K	117	276.13K	5	100 MB
KEGG	1.09M	34.26K	21	939.26K	4	118 MB
LinkedCT	9.80M	981.72K	90	3.81M	13	1.7 GB
Medicare	44.49K	6,819	6	23.30K	1	6.8 MB
SIDER	99.15K	2,661	11	27.07K	2	18 MB
TCGA-A	35.33M	5.78M	383	8.29M	23	6.8 GB
Total	96.10M	8.32M	742	27.47M	55	16.0 GB

Table 6.1: **LSLOD Data Set Characteristics**, where #tp - number of triples, #s - number of subjects, #p - number of predicates, #o - number of objects, and $RDF - RDF file size.

6.1.2 Benchmarking Queries

The benchmark contains 20 queries over the data sets. The queries were created in collaboration with domain experts and are supposed to simulate frequently used queries in the Life Science domain [21]. The queries can be divided into two groups, 10 simple queries and 10 complex queries. Table 6.2 presents the key characteristics of the simple queries from the LSLOD benchmark. The query optimization is based on graph patterns. The complex queries do not add any additional challenges in terms of optimizing the query plan based on its sub-queries. Therefore, it is sufficient to evaluate the approach using the simple queries only. The queries are used to follow the test bed and are depicted in Appendix A.

6.2 Data Preparation

To ensure reproducibility a dump of the unified endpoint [22] from 2016 is used. The dump contains all triples of the data sources in the LSLOD cloud partitioned according to the RDF-MTs, i.e., one file per RDF-MT. There is one additional file belonging to the dump that describes all the RDF-MTs within the data of the dump. To identify the impact of variety on the performance of existing federated query engines data sets of different data formats are needed. In this experimental setup the used data formats are **(i)** SPARQL endpoint, and **(ii)** relational database. Hence,

	SQ1	SQ2	SQ3	SQ4	SQ5	SQ6	SQ7	SQ8	SQ9	SQ10
#TPs	4	7	6	5	5	3	4	3	8	8
#BGPs	2	1	1	1	2	1	1	1	1	1
#SSQs	2	3	4	2	3	2	3	2	2	2
`UNION`	✓									
`OPTIONAL`					✓					
`DISTINCT`				✓					✓	✓

Table 6.2: **LSLOD Query Characteristics**, where #TPs - number of triple patterns, #BGPs - number of Basic Graph Patterns, and #SSQs - number of star-shaped sub-queries.

the semantic data from the RDF dump needs to be loaded to a relational database. A common feature of relational database management systems is populating tables with data from CSV/TSV files. Therefore, the RDF data is converted to TSV.

6.2.1 Converting RDF to TSV

Algorithm 6.1 describes the process of the RDF to TSV conversion. First the source root URL of the RDF-MT is extracted from the LSLOD MT file (line 1). Second all predicates for the RDF-MT are extracted from this file (line 2). Then the prefix for all predicates is computed, put into a prefix list and cut off from the predicate itself (line 3). After this preprocessing, the RDF file is read and every triple is put into a Python dictionary (line 4). If there are predicates with multiple values for one subject, the Python dictionary entry is updated to use a list instead of a single value. After reading all triples, each predicate is checked for subjects with multiple values (line 5). Those predicates are stored in a list to be able to differentiate them from single value predicates. Afterwards, the Python dictionary is written to disk as a TSV file (line 6), but only single value predicates are considered here. Predicates with multiple values for at least one subject are stored in a separate file. Those multi-

Algorithm 6.1 RDF2TSV

```
1: source_root = get_source_root(rdf_file)
2: predicates = get_predicates_from_mapping(source_root)
3: prefixes, predicates = compute_prefixes(predicates)
4: data = read_rdffile(predicates)
5: multivalue_predicates = find_multivalue_predicates(data, predicates)
6: write_tsv_file(data, predicates, multivalue_predicates)
7: write_mapping_file(prefixes, predicates, multivalue_predicates)
```

value predicate files contain two columns, one for the subject and one for the object of the triple. This approach is chosen because it is not feasible to store the Cartesian product of all predicate values for each subject due to very high redundancy. In the end, RML mapping files for MySQL are generated (line 7).

6.2.2 Implementation of the Relational Schema

For importing data from TSV into MySQL, a table has to be created in advance. The table definition is created from the TSV header, i.e., all column names are read from the previously generated TSV file. Two cases have to be considered when creating the tables: **(i)** creation of the main table for the RDF-MT where only the first column (subject column) is the primary key, and **(ii)** creation of a multi-value predicate table where both columns build a combined primary key. Due to the data quality there are some tables with the same string only differing in upper and lower case at key positions. In order to not loose any data both entries should be preserved. This is achieved by setting the default coalition to `latin1_general_cs` which enables *case-sensitive* keys. After the table is created, the data is loaded into it using the `LOAD DATA INFILE` method of MySQL. One row is ignored since the header does not contain data but metadata. The relational schemas generated that way are not necessarily in 3NF. The tables needed for the queries of the benchmark are analyzed using the algorithm described in [8] to find functional dependencies. Indexes for candidate keys are added by hand to the SQL scripts generated previously. Some tables have to be split up in order to fulfill the requirements of 3NF. This splitting is done by hand in the SQL scripts and the RML mappings.

6.3 Experimental Setup

The RDF version of Ontario is compared to the state-of-the-art query engines FedX [41], ANAPSID, [2] and MULDER [13]. The results of the RDB and the RDF+RDB version of Ontario are also evaluated against Ontario-RDF. Each query is run ten times per engine and the best run is reported. Where the best run is the run with the shortest query execution time while obtaining maximal answer completeness, i.e., fast runs due to the unavailability of a source are not reported.

Benchmark: For the experimental evaluation the ten simple queries of the LSLOD benchmark are used. These queries can be evaluated by eight of the ten data sets, Table 6.3 shows the data sets needed for the queries as well as the used format when heterogeneity is considered. The data sets were distributed in a way that ensures that every query contacts at least one RDF and one RDB source.

Source	Format
ChEBI	RDF
Dailymed	RDF
Diseasome	RDF
DrugBank	RDB
KEGG	RDF
LinkedCT	RDB
SIDER	RDB

Table 6.3: **Data Source Formats**

Metrics: The following metrics are reported and analyzed in the reminder of this chapter: **(i)** *Execution Time:* The time elapsed between the submission of a query and the retrieval of all results produced by the engine. The reported time corresponds to absolute wall-clock system time as returned by the Python `time.time()` function. The timeout for each query is set to 300 seconds (five minutes). **(ii)** *Cardinality:* The number of results produced by the engine for a given query, i.e., the cardinality of the result set. **(iii)** *Completeness:* The percentage of the query results with respect to the answers produced by the unified SPARQL endpoint, i.e., an endpoint with the data of all ten data sets. The result cardinality of the unified endpoint is referred to as *ground truth*. **(iv)** *dief@t:* A measure of continuous efficiency of an engine in the first t time units of query execution (cf. Subsection 2.4.2).

Implementations: Ontario and the proposed optimizations are implemented in *Python 3.6*. For the SPARQL endpoints *Virtuoso 6.01.3127* is used. The relational databases are realized in *MySQL 5.7*. Every component of the experiment runs in a dedicated Docker container, i.e., one container per query processing engine, and one container per data set and format. In total the experiment uses 24 Docker containers, i.e., four engine containers, ten RDF containers, and ten RDB containers. The RDF containers are limited to 16 GB of memory and eight threads each. All Docker containers run on the same server and, therefore, network cost can be neglected. The experiments are executed on an Ubuntu 16.04.6 LTS 64 bit machine with two Intel(R) Xeon(R) Platinum 8160 2.10 GHz CPUs (total: 48 physical cores, 96 threads), and 755 GiB DDR4 RAM. Three version of Ontario are compared: **(i)** *Ontario* the baseline implementation of Ontario, i.e., without the proposed heuristics, **(ii)** *Ontario-GH* makes use of the general heuristics from Section 4.3, and **(iii)** *Ontario-SSH* using the source specific heuristics described in Section 4.4.

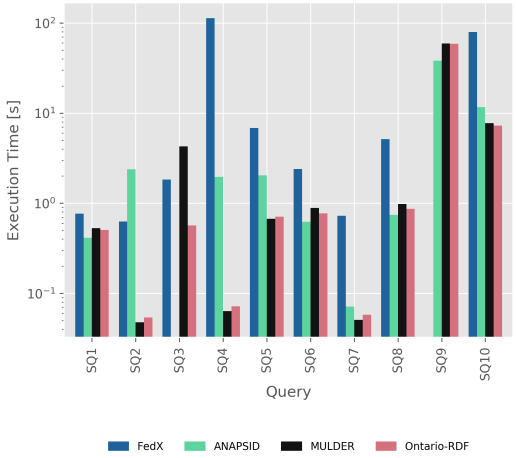


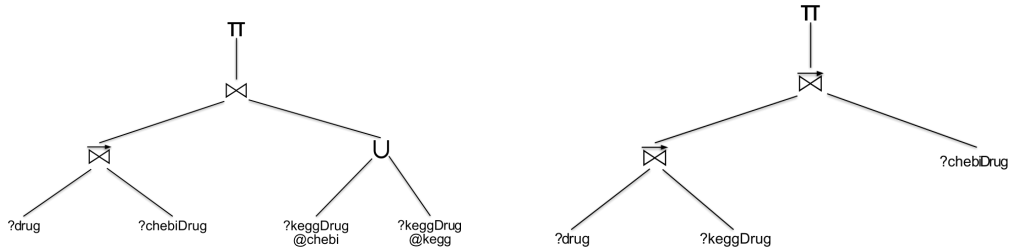
Figure 6.2: **Query Execution Time per RDF Engine**

6.4 Empirical Evaluation

In the following the results obtained in the experiments are analyzed regarding the metrics used and the research questions. First, Ontario is compared to the state-of-the-art RDF engines FedX [41], ANAPSID [2], and MULDER [13]. Afterwards, the different heuristics implemented in this thesis are evaluated against the baseline implementation of Ontario. Every implementation is examined over three different settings of the data sources: **(i)** *RDF* a federation of RDF graphs accessible via SPARQL endpoints, **(ii)** *RDB* all data sets represented as RDB tables stored in MySQL, and **(iii)** *SDL* a Semantic Data Lake composed of RDF graphs accessible via SPARQL endpoints and RDB tables stored in MySQL, distributed as shown in Table 6.3. Throughout the empirical evaluation Ontario is annotated with the data formats used, i.e., *Ontario-GH-RDF* denotes Ontario with general heuristics over RDF. Since Ontario is meant to be run over a Semantic Data Lake, executing the source specific heuristics over the SDL is called *Ontario-SSH*. The baseline implementation is referred to as *Ontario*.

6.4.1 Experiment I: RDF Engines

In this experiment, the state-of-the-art RDF engines FedX, ANAPSID, and MULDER are compared to the baseline implementation of Ontario, i.e., Ontario without the proposed heuristics. This experiment investigates **RQ1**.

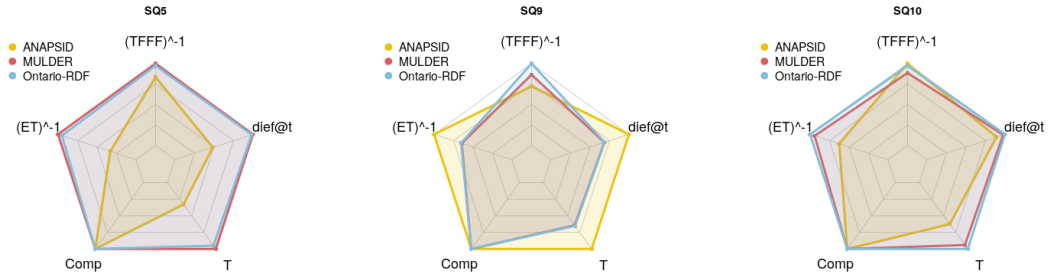


(a) MULDER Plan (b) Ontario Plan

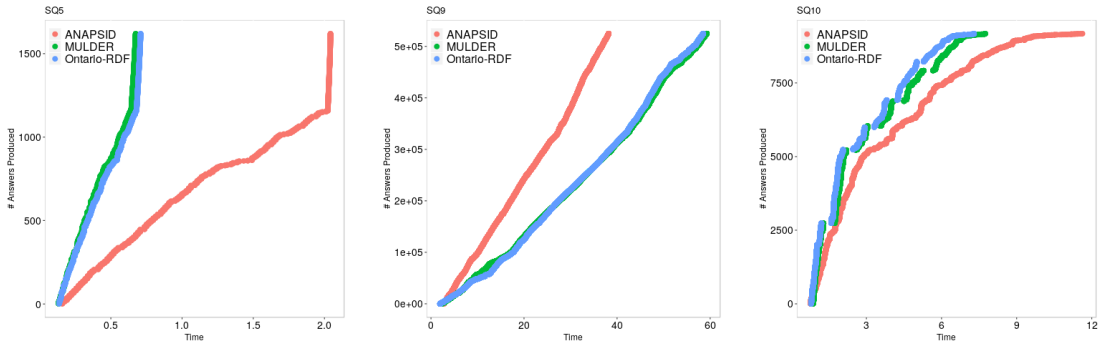
Figure 6.3: **Query Plans for SQ3**. An arrow over the join indicates a nested loop join. (a) MULDER plan with union over ChEBI and KEGG. (b) Ontario plan without union.

Execution Time Figure 6.2 shows the execution time for every query and baseline RDF engine. ANAPSID outperforms the other query engines in SQ1, SQ6, SQ8, and SQ9. Ontario generates the same plans as ANAPSID for those queries apart from the ordering of SSQs for the first three queries. The operators with changed order of left and right operand are `UNION` and `GJoin` [2]. Since the order of operands does not matter for these operators, this does not explain the difference in execution time. Because Ontario uses the operators of ANAPSID this behavior is unexpected. It is possible that there are changes made to the ANAPSID operators used by Ontario that reduce the efficiency, but this is subject of further research. MULDER does perform better than Ontario in some queries where both engines generate the same plan, i.e., SQ2, SQ4, SQ5, and SQ7. For the other queries with the same plans generated by MULDER and Ontario, the latter performs better. For SQ3 Ontario finds an even better plan than MULDER does. Ontario achieves this by pruning ChEBI as source for the triple pattern `?keggDrug bio2RDF:url ?keggUrl` which can be answered by ChEBI and KEGG. However, contacting ChEBI is unnecessary since the triple pattern `?drug drugbank:keggCompoundId ?keggDrug` binds the variable `?keggDrug` to an RDF-MT in KEGG. The different query plans are shown in Figure 6.3. Additionally, Ontario performs better in SQ9 and SQ10 because it omits the projection operator which is not needed for those queries. Thus, **RQ1** can be answered by the findings in this experiment. It can be suggested that considering heterogeneity does not have an impact on query performance.

Continuous Behavior Figure 6.4 reports the performance of state-of-the-art RDF query engines in producing answers continuously. FedX does not produce results continuously, but returns all results at the same time. Therefore, its continuous behavior



(a) Continuous Efficiency Measures for Queries SQ5, SQ9, and SQ10



(b) Answer Traces for Queries SQ5, SQ9, and SQ10

Figure 6.4: **Continuous Performance of RDF Engines**. (a) Continuous efficiency measures, where $TFFF^{-1}$ – inverse time for first result, ET^{-1} – inverse execution time, Comp – completeness, T – throughput, and *dief@t* continuous efficiency at time t. (b) Traces showing the incremental answer generation; a steeper angle correlates with a higher diefficiency

is not measured. The continuous efficiency at time t, i.e., *dief@t*, inverse of time for the first tuple ($TFFF^{-1}$), inverse of total query execution time (TE^{-1}), percentage of answers produced (Comp), and throughput (T) are presented in Figure 6.4a using radar plots. The interpretation of these measures is *'higher is better'* in each axis. For all queries, the completeness (Comp) is 100%. The throughput T varies because it correlates with the total execution time of the query. As clearly shown, the continuous efficiency of Ontario-RDF and MULDER is better for SQ5 and SQ10. Additionally, MULDER performs better on SQ5 and Ontario-RDF performs better on SQ10. As opposed to that, the continuous efficiency of ANAPSID is better for SQ9, even though MULDER and Ontario-RDF are able to produce the first answer faster than ANAPSID. Figure 6.4b presents the incremental answer generation of

Engine	SQ1	SQ2	SQ3	SQ4	SQ5	SQ6	SQ7	SQ8	SQ9	SQ10
FedX	5,146	**1**	393	**24**	1,620	8,120	27	8,201	–	**8,708**
ANAPSID	5,146	3	**0**	28	1,620	8,120	27	8,201	524,694	9,174
MULDER	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-RDF	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
GNDT	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174

Table 6.4: **Number of Results per Query over RDF**, where GNDT - ground truth and a dash representing timed out queries.

the RDF query engines. It can be seen that Ontario-RDF and MULDER produce the results faster than ANAPSID for SQ5. However, ANAPSID generates the results of SQ9 at a faster rate. For SQ10 Ontario-RDF retrieves all answers first, but the answer generation of ANAPSID is smoother, i.e., there are no gaps between the generation of two answers.

Answer Completeness The number of answers returned by the RDF engines is reported in Table 6.4. Except for the following exceptions all engines return all results. FedX times out during the execution of query SQ9. Since FedX is producing all results at once, no results are returned. Also the results for SQ2, SQ4, and SQ10 produced by FedX are not complete. FedX only returns one of the three answers to SQ2, 24 of 28 answers of SQ4, and 8,708 of 9,174 answers of SQ10. ANAPSID retrieves no answer for SQ3 even though it finds the same plan as MULDER. The number of answers collected from the endpoints and the result of the union operator are correct, but ANAPSID fails to use the result of the union as input of a nested loop join. Thus, this join returns no result.

6.4.2 Experiment II: Ontario with Different Heuristics

The second experiment aims at evaluating the proposed heuristics compared to the baseline implementation of Ontario. All versions of Ontario are analyzed over RDF, RDB, and RDF+RDB. This allows examining the source specific heuristics as well as the performance in a SDL. The experiment is guided by **RQ2**, **RQ3**, and **RQ4**.

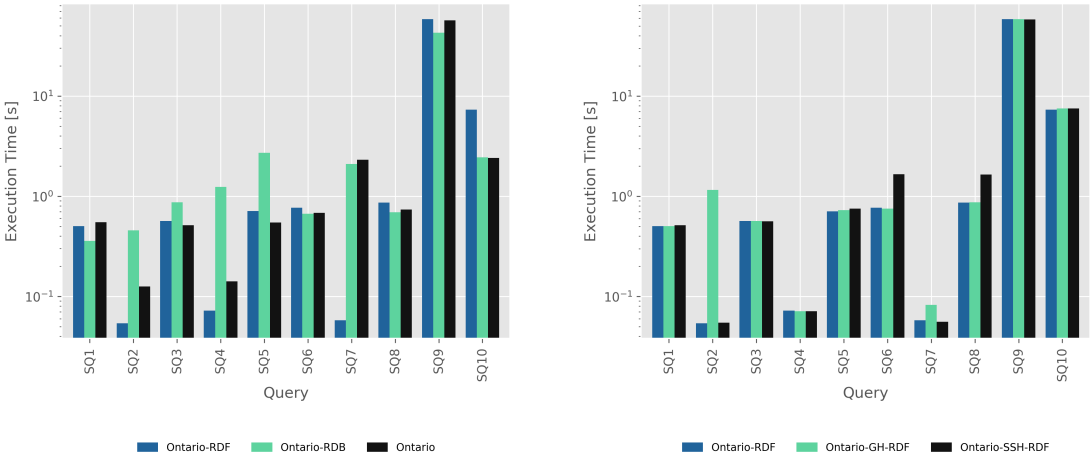
Execution Time Figure 6.5a reports the query execution of the baseline implementation over the different data formats. Ontario finds the answers for SQ1 faster when using RDB. Using RDF the results for SQ2 are created much faster while SQ5 benefits from being executed over the SDL. The results suggest that none of the data

formats clearly outperforms the others using the baseline implementation. However, executing SSQs over the data format with the highest performance for this SSQ increases the performance.

Figure 6.5b shows the performance of the different heuristics implemented in this thesis executed over a federation of RDF graphs. The difference in execution time of SQ2 proves that Heuristic 8 is able to improve query performance and thus provides an answer to **RQ4**. The results of SQ6 and SQ8 suggest that further conditions should be checked when Heuristic 4 can be applied. The only difference in the plan for those queries is that according to the source specific heuristic a nested loop join is used. The GJoin used by the other approaches performs better for queries SQ6 and SQ8. Both operands of the join have low selectivity. Therefore, applying Heuristic 4 as it is seems to be inefficient when dealing with two sub-queries with low selectivity which answers **RQ4** as well.

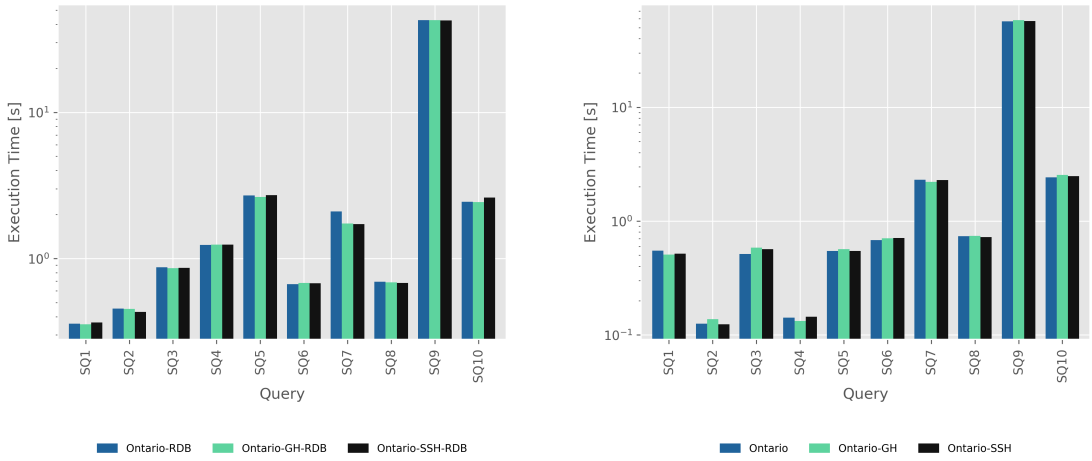
Figure 6.5c presents the execution time of the queries from the benchmark executed over RDB with different heuristics. There are no differences in the execution times of the queries because the LSLOD queries do not match the conditions of the RDB heuristics. Therefore, these heuristics are evaluated using additional queries designed for them. The additional queries are listed in Appendix B. They contain joins that can be pushed down to MySQL, a constant object or a filter expression on an object variable. The constant objects and filter expressions are over unindexed columns in the relational table. The results of the additional queries for the RDB heuristics are discussed in Subsection 6.4.3.

Figure 6.5d reports the query performance of the proposed heuristics compared to the baseline implementation when executed over the SDL. Despite the fact that all versions of Ontario produce the same plan for SQ1 and SQ4, the general heuristic approach performs better for those queries. This might be due to normal variation within query execution time of the sources. Similar to the RDF case the performance of SQ2 is increased by pushing down the join of two SSQs evaluated over RDF. However, the difference is smaller in the case of a SDL. This is due to a different join order and implementation. The baseline implementation of Ontario performs better in executing SQ3. This difference in performance is caused by a change in the order of a nested loop join. Due to the data source score, the baseline implementation changes the order of the SSQs. The score is based on experience and re-orders the list of SSQs in a way such that SSQs over more expansive data sources are evaluated as late as possible. But the implementation of this heuristic may ignore the order of two SSQs based on the selectivity if they have the same data source score. In this case the described behavior improves the query execution plan not because of any choices made by a heuristic but because of the implementation only.



(a) Baseline over RDF, RDB, and SDL

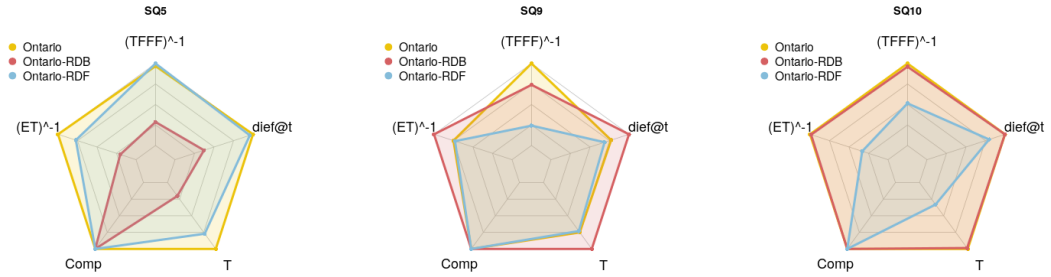
(b) Different heuristics over RDF



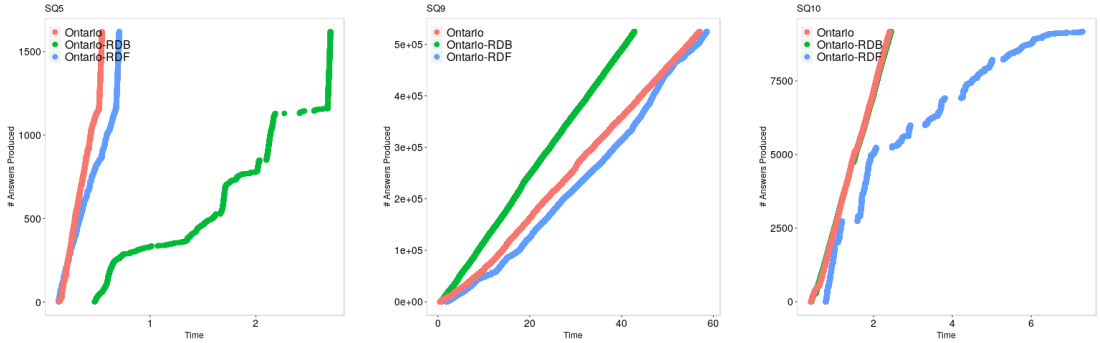
(c) Different heuristics over RDB

(d) Different heuristics over SDL

Figure 6.5: **Query Execution Time for Ontario** in seconds with different heuristics over varying data formats. A shorter bar means shorter execution time and, therefore, better query performance. (a) No data format clearly outperforms the others. (b) For some queries general heuristics over RDF are not sufficient. (c) The benchmark queries are not suitable for evaluating the RDB heuristics. (d) Source specific heuristics have no impact on query performance in the setup of the SDL.



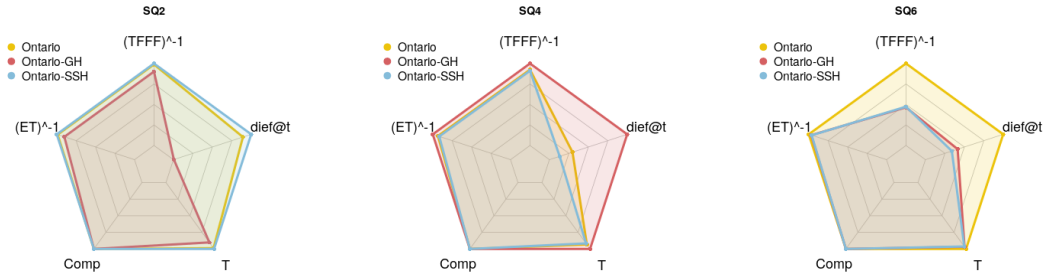
(a) Continuous Efficiency Measures for Queries SQ5, SQ9, and SQ10



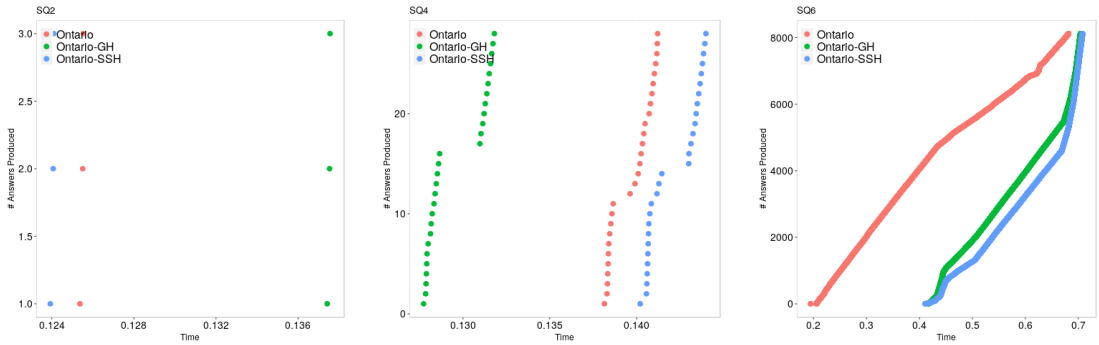
(b) Answer Traces for Queries SQ5, SQ9, and SQ10

Figure 6.6: **Continuous Performance of Baseline Ontario**. (a) Continuous efficiency measures, where $TFFF^{-1}$ – inverse time for first result, ET^{-1} – inverse execution time, Comp – completeness, T – throughput, and *dief@t* continuous efficiency at time t. (b) Traces showing the incremental answer generation; a steeper angle correlates with a higher diefficiency

Continuous Behavior Figure 6.6 shows the continuous performance of the baseline implementation of Ontario over the different data models. The continuous efficiency measures are presented in Figure 6.6a. As clearly shown, SQ5 benefits from being executed over the Semantic Data Lake. However, Ontario executed over the SDL retrieves the first answer the fastest, but overall Ontario-RDB performs better on query SQ9. For SQ10 Ontario performs similarly when executed over the SDL or RDB. The traces of the continuously generated results are visualized in Figure 6.6b and aligned with the previously discussed behavior. Hence, Ontario is able to find effective and efficient plans for heterogeneous data sources. In addition, RDF-MT based source descriptions can be applied efficiently for query processing over non-RDF sources. These findings allow for answering **RQ2** and **RQ3**.



(a) Continuous Efficiency Measures for Queries SQ2, SQ4, and SQ6



(b) Answer Traces for Queries SQ2, SQ4, and SQ6

Figure 6.7: **Continuous Performance of Ontario over SDL**. (a) Continuous efficiency measures, where $TFFF^{-1}$ – inverse time for first result, ET^{-1} – inverse execution time, Comp – completeness, T – throughput, and *dief@t* continuous efficiency at time t. (b) Traces showing the incremental answer generation; a steeper angle correlates with a higher diefficiency

Figure 6.7 reports the continuous performance of the different heuristics evaluated over the SDL. The measures of continuous efficiency presented in Figure 6.7a suggest that the new heuristics have an impact on the continuous behavior. While the difference in execution time is quite small, the diefficiency at time t varies a lot. This can best be observed in SQ6. The baseline implementation is able to find the first answer way faster than the other heuristics, but the execution time is almost similar. Those findings are aligned with the answer traces presented in Figure 6.7b. The baseline implementation produces the results for SQ6 faster in the beginning, but in the end all approaches are more or less equal. Ontario is able to find effective and efficient plans for query processing over the SDL which answers **RQ3**.

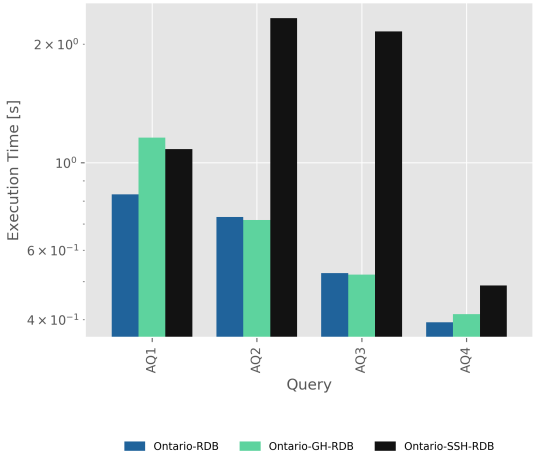


Figure 6.8: **Query Execution Time for Additional Queries** in seconds. Source specific heuristics are increasing the query execution time.

Answer Completeness Each version of Ontario produces the same number of results despite the data format. Table 6.4 shows that Ontario-RDF returns all answers. Ontario retrieves all answers for the additional queries as well. Hence, the optimization techniques used are able to generate effective query plans for heterogeneous data sources. This partially answers **RQ2** and **RQ3**.

6.4.3 Experiment III: Additional Queries

The simple queries from the LSLOD benchmark do not meet the conditions for the RDB specific heuristics proposed in Section 4.4. Hence, these heuristics are evaluated using additional queries tailored for them. The additional queries are listed in Appendix B. They contain joins that can be pushed down to MySQL, a constant object or a filter expression on an object variable. These filters are executed over unindexed columns. In the following only the execution time of these queries regarding the heuristics is discussed since the evaluation of the additional queries does not change the prior findings on continuous behavior and answer completeness.

Figure 6.8 presents the query execution time of the additional queries executed with different heuristics over RDB. Ontario-RDB outperforms the other approaches but for AQ2 where Ontario-GH-RDB performs slightly better. Ontario-SSH-RDB increases the query execution time for every query compared to the baseline. Even though Ontario-RDB and Ontario-GH-RDB produce the exact same plans for all four queries, the query execution time varies. For AQ2 - AQ4 the difference might

be within the normal range of query execution time at the data sources, but the difference in AQ1 is not. However, during the execution of AQ1 a string comparison of an unindexed column has to be evaluated. Ontario-SSH-RDB is pushing this filter up to Ontario level. On one hand, the execution time of AQ1 indicates that even though the filtered column is unindexed it might be faster to evaluate the expression in MySQL. On the other hand, Ontario-GH-RDB takes longer than Ontario-SSH-RDB. The results of AQ2 and AQ3 clearly suggest to not push down simple joins to MySQL, at least not in the current manner. Ontario translates both star-shaped sub-queries and joins the results of the execution of both queries. Following this approach no index can be used for the join and, therefore, the execution time increases. However, the execution time might be reduced when translating the query into one single SQL query containing the join. AQ4 modifies AQ3 into using a constant object. As the filter expression is evaluated first, Ontario-SSH-RDB is not pushing down the join in this case. This results in a smaller difference to the other approaches. The difference can be explained as in the discussion of AQ1.

The previously discussed results of executing the additional queries over RDB suggest that heuristics to improve query performance over RDB in a SDL have to be further researched and depend on the implementation of the wrapper. Heuristic 6 is decreasing query performance even though it is intended to perform better than evaluating the filter expression on an unindexed column in MySQL. Heuristic 7 aims to increase query performance by pushing down simple joins over indexes attributes to MySQL. This might reduce query execution time and allow other operations to be run in parallel, but the current implementation of query translation does not allow the use of the index over the join attribute, hence the query execution time is increasing. Therefore, the answer to **RQ4** is negative for the RDB heuristics.

The results of the experiments show that Ontario and the proposed heuristics improve query performance. Experiment I shows that considering heterogeneity during query processing does not add significant overhead. For some queries Ontario is even able to find better execution plans than state-of-the-art RDF query engines. Experiment II proves that RDF-MT based source descriptions can be used effectively and efficiently for non-RDF sources as well. As also shown in this experiment, Ontario generates effective and efficient plans for heterogeneous sources. The RDF heuristics are likely to improve the query performance. The heuristics for RDB sources are studied in experiment III since the queries from the benchmark do not match the conditions of the heuristics. The RDB heuristics are good in theory, but decrease query performance with the current implementation. Ontario and the proposed heuristics overcome some limitations of state-of-the-art query engines, but source specific heuristics can be improved.

Chapter 7

Conclusions and Future Work

This thesis presents basic heuristics to improve query performance over Semantic Data Lakes by finding efficient and effective query plans, considering not only the data descriptions of the data sources, but also their query processing capabilities. The approach is based on the query processing engine Ontario, an engine that is able to execute queries over Semantic Data Lakes composed of data sources in different data formats. The query processing of Ontario uses RDF-MT based source descriptions, i.e., an abstract description of the properties belonging to the entities in the unified schema of the data in the Data Lake. The proposed heuristics were tested over the LSLOD benchmark and compared to the baseline implementation of Ontario which was compared to state-of-the-art RDF query engines. This chapter concludes the work of this master's thesis by summarizing the answers of the research questions presented in chapter 6, discussing problems and limitations of the approach as well as its implementation. Finally, an overview of ideas for future research work is given.

7.1 Conclusions

As discussed in the empirical evaluation of RDF query engines (Subsection 6.4.1), it can be suggested that considering heterogeneity during federated query processing does not have an impact on query performance. This conclusion can be drawn since Ontario has similar query execution times as MULDER does. If both query engines generate the same plan, either Ontario or MULDER may perform slightly better, but neither of these two query engines is always performing better when generating the same query execution plan. The answer to **RQ1** is: *'Considering heterogeneity during federated query processing does not significantly impact on the query performance.'*

The analysis of the continuous behavior and answer completeness of Ontario in Subsection 6.4.2 answers the second research question which investigates the effectiveness and efficiency of RDF Molecule Template based source descriptions for non-RDF sources. Since some queries can be answered faster when executing them over RDB compared to RDF while sustaining complete results, RDF-MT based source descriptions can be effectively and efficiently applied for non-RDF sources as well. Hence, the answer to **RQ2** is: *'RDF-MT based source descriptions can be applied effectively and efficiently during query processing over non-RDF data sources.'*

The effectiveness and efficiency of the optimization techniques of Ontario are evaluated in Subsection 6.4.2. The findings of the experiment analysis show that Ontario produces effective and efficient query plans for heterogeneous data sources. The effectiveness is proven by examining the number of returned results which shows an answer completeness of 100% for all queries executed by any version of Ontario. Hence, the generated query plans are effective. The efficiency can be proven by studying the continuous behavior of Ontario. The query plans generated for queries over the Semantic Data Lake are not only compatible with query plans over RDF sources, but for some queries Ontario is able to increase query performance when executing the query over heterogeneous data sources compared to the execution of the same query over RDF sources only. This clearly shows the efficiency of the optimization techniques used. Therefore, the answer to **RQ3** is: *'The optimization techniques of Ontario are able to generate effective and efficient query execution plans for heterogeneous data sources.'*

As shown in Figure 6.5b, the general heuristics approach does not improve query performance but decreases it for queries SQ2 and SQ7. The performance of the RDF based heuristics is discussed in Subsection 6.4.2. Pushing down a SPARQL join improves query performance for SQ2 and, therefore, Heuristic 8 proves to be of help in optimizing queries for RDF data sets. However, the results of the experiment suggest that strictly following Heuristic 4 is likely to decrease query performance. This is the case for SQ6 and SQ8 where both operands of the join have low selectivity. As clearly shown in Subsection 6.4.3, the RDB based heuristics failed to improve the query performance. On the contrary, applying the RDB specific heuristics leads to increased query execution times. This suggests that the filter implementation of MySQL over unindexed attributes is faster than the implementation of filters in Ontario which also includes the increased intermediate result that has to be received. Hence, the answer to **RQ4** is: *'Most of the proposed heuristics do not improve query performance or on the contrary increase query execution time. Further analysis on source specific heuristics and how to implement them efficiently is needed.'*

7.2 Limitations

Not all of the proposed heuristics work as intended. Heuristic 4 is likely to increase query execution time if both operands of the join have low selectivity. Heuristic 6 decreases query performance, suggesting that retrieving a larger intermediate result and evaluating simple string comparison at Ontario level is significantly less efficient than evaluating the filter over an unindexed attribute in MySQL. Functions like `str` occurring in a SPARQL filter expression can not be pushed down to RDB sources using this approach. The functions are dropped and the source tries to find the answer without evaluating the function. Hence, filters containing functions should be executed at Ontario level or in RDF sources. Heuristic 7 is not improving query execution due to the implementation of the query translation. Since both sub-queries are translated on their own and used as `FROM` sub-queries in the join the index can not be used. Therefore, the query performance is decreasing when pushing down the join. Next, the query translation is discussed in more detail.

The current implementation of the query translation from SPARQL to SQL has limitations. Semantifying the projections considers single column templates only. Therefore, templates like `<http://example.org/{start}_{end}>` can not be translated correctly. Algorithm 5.1 is implemented for queries with at most two RDF-MTs. Hence, queries with three or more RDF-MTs can be only partially translated. Furthermore, only two of the three possible cases when dealing with queries containing several RDF-MTs are implemented at the time of writing. Queries containing a join and also a union over a subset of the attributes can not be translated using the current implementation. Therefore, the query translation works for queries containing a join or fully unified query results only. Another limitation of the current translation process is that the intersection of predicates answerable by the different RDF-MTs is checked and not the annotations of the triples.

7.3 Future Work

The results of this thesis suggest that federated query processing over Semantic Data Lakes might be subject of further research for the next years or even decades. Based on the findings concerning the proposed heuristics, more sophisticated heuristics for RDB and even RDF are needed in order to increase the query performance. Real world applications might not be limited to data from RDF and RDB and, therefore, heuristics for other data models could be defined and integrated along with appropriate wrappers. The query translation of Ontario does not cover all possible cases for semantification. Additionally, the translation from SPARQL to

SQL queries is not efficient for joining star-shaped sub-queries at the relational data source. Hence, the query translator can be improved which eventually leads to better query performance. In order to understand the generated query plans and improve them, the performance difference of Ontario and ANAPSID when both engines generate the same plan should be investigated. This investigation is out of the scope of this thesis since it takes time to analyze the implementations of both query engines. As Ontario uses the same operators as ANAPSID does, the query performance should be similar if the same plan is executed. The relevant indexes for the queries from the benchmark were defined in a JSON file by hand. In the future, finding indexes in RDB sources could be integrated into the extraction of RDF-MTs from the sources. Hence, the indexes would be a part of the RDF-MT based source description of RDB sources like the user information for the database connection. The results of the empirical evaluation suggest that there is an optimal data model for each sub-query. In a real world application several different data models might be used. Machine learning techniques might be able to predict the best data model for a sub-query. These predictions could be used for query planning. If a sub-query would benefit from being executed over RDF, the actual data could be lifted to RDF in order to improve the query execution time.

Ontario and the proposed heuristics effectively and efficiently deal with the main challenges in query processing over Data Lakes, i.e., source selection, query execution plan generation, and combining partial results from the sources in the Data Lake. As shown in the evaluation, Ontario does not add a significant overhead by considering heterogeneity during query processing. Therefore, it is not necessary to lift all data sets to RDF because they can be queried efficiently in their native format. Further research should focus on improving source specific heuristics for RDF and RDB as well as defining source specific heuristics for other frequently used data models.

Appendix A

LSLOD Simple Queries

The queries are taken from [21] and can also be found online at the project page accessible via http://srvgal78.deri.ie/BioFed/queries.html. Some of the queries where slightly changed in order to correct typos and other errors in the published queries.

Listing A.1: Prefixes

```
PREFIX bio2RDF: <http://www.bio2rdf.org/ns/bio2rdf#>
PREFIX dailymed: <http://www4.wiwiss.fu-berlin.de/dailymed/resource/dailymed/>
PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX drugcategory: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugcategory>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX linkedCT: <http://data.linkedct.org/resource/linkedct/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX purl: <http://purl.org/dc/elements/1.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX sider: <http://www4.wiwiss.fu-berlin/sider/resource/sider/>
```

Listing A.2: **SQ1:** Find all the drugs along with their indications

```
SELECT ?genericName ?indication WHERE {
    {
        ?dn drugbank:genericName ?genericName ;
        drugbank:indication ?indication .
    }
    UNION
    {
        ?dn dailymed:name ?genericName ;
        dailymed:indication ?indication .
    }
}
```

Listing A.3: **SQ2:** Find all the drug descriptions and chemical equations of reactions related to drugs from category Cathartics

```
SELECT ?drugDesc ?cpd ?equation WHERE {
    ?drug drugbank:drugCategory durgcategory:cathartics .
    ?drug drugbank:keggCoumpoundId ?cpd .
    ?drug drugbank:description ?drugDesc .
    ?enzyme kegg:xSubstrate ?cpd .
    ?enzyme rdf:type kegg:Enzyme .
    ?reaction kegg:xEnzyme ?enzyme .
    ?reaction kegg:equation ?equation .
}
```

Listing A.4: **SQ3:** Find all drugs, together with the URL of the corresponding web-pages as well as images if available

```
SELECT ?drug ?keggUrl ?chebiImage WHERE {
    ?drug rdf:type drugbank:drugs .
    ?drug drugbank:keggCompoundId ?keggDrug .
    ?keggDrug bio2RDF:url ?keggUrl .
    ?drug drugbank:genericName ?drugBankName .
    ?chebiDrug purl:title ?durgBankName .
    ?chebiDrug bio2RDF:image ?chebiImage .
}
```

Listing A.5: **SQ4:** Find KEGG drug names of all drugs in DrugBank belonging to category Micronutrient

```
SELECT ?drug ?title WHERE {
    ?drug drugbank:drugCategory drugcategory:micronutrient .
    ?durg drugbank:casRegistryNumber ?id .
    ?keggDrug rdf:type kegg:Drug .
    ?keggDrug bio2RDF:xRef ?id .
    ?keggDrug purl:title ?title .
}
```

Listing A.6: **SQ5:** Find all drugs and their mass that affect humans and mammals. For those having a description of their biotransformation, also return this description

```
SELECT ?drug ?transform ?mass WHERE {
    ?drug drugbank:affectedOrganism 'Humans and other mammals' .
    ?drug drugbank:casRegistryNumber ?cas .
    ?keggDrug bio2RDF:xRef ?cas .
    ?keggDrug bio2RDF:mass ?mass .
    OPTIONAL {
        ?drug durgbank:biotransformation ?transform
    }
}
```

Listing A.7: **SQ6:** Find diseases and corresponding drugs that target those diseases

```
SELECT ?drug ?disease ?name WHERE {
    ?drug drugbank:molecularWeightAverage ?weight .
    ?drug drugbank:possibleDiseaseTarget ?disease .
    ?disease diseasome:name ?name .
}
```

Listing A.8: **SQ7:** Find drugs and their side effects with labels for the drug name "Sodium Phosphate" in dailymed

```
SELECT ?drug ?sideeffect ?label WHERE {
    ?drug dailymed:name 'Sodium Phosphates' .
    ?drug owl:sameAs ?drugAlt .
    ?drugAlt sider:sideEffect ?sideeffect .
    ?sideeffect rdfs:label ?label .
}
```

Listing A.9: **SQ8:** Find diseases and corresponding drugs that target those diseases along with their labels

```
SELECT ?drug ?disease ?label WHERE {
    ?disease diseasome:name ?diseasename .
    ?disease drugbank:possibleDiseaseTarget ?drug .
    ?drug rdfs:label ?label .
}
```

Listing A.10: **SQ9:** Find intervention names and ids for the drugs in dailymed with drug dose, description, inactive ingredients as well as possible disease targets

```
SELECT DISTINCT * WHERE {
    ?intervention a linkedCT:intervention ;
    linkedCT:intervention_name ?intervention_name ;
    linkedCT:intervention_id ?intervention_id ;
    rdfs:seeAlso ?dailymedDrug .
    ?dailymedDrug dailymed:dosage ?dosage ;
    dailymed:description ?description ;
    dailymed:inactiveIngredient ?inactiveIngredient ;
    dailymed:possibleDiseaseTarget ?possibleDiseaseTarget .
}
```

Listing A.11: **SQ10:** Find intervention names and types for the drugs in DrugBank with drug chemical structure, drug state, its protein binding and smiles string canonical

```
SELECT DISTINCT * WHERE {
    ?intervention a linkedCT:intervention ;
    linkedCT:intervention_name ?intervention_name ;
    linkedCT:intervention_type ?intervention_type ;
    rdfs:seeAlso ?drugbankDrug .
    ?drugbankDrug drugbank:structure ?structure ;
    drugbank:state ?state ;
    drugbank:proteinBinding ?proteinBinding ;
    drugbank:smilesStringCanonical ?smilesStringCanonical .
}
```

Appendix B

Additional Queries

The additional queries were designed for the evaluation of the RDB heuristics presented in Section 4.4. Since those queries are run over the LSLOD data sets, they use the same prefixes as presented in Listing A.1.

Listing B.1: **AQ1:** Find all drugs in DrugBank and their title in KEGG with the chemical formula C8H10N4O2

```
SELECT ?drug ?title ?cf WHERE {
    ?drug drugbank:casRegistryNumber ?id .
    ?keggDrug rdf:type kegg:Drug .
    ?keggDrug bio2RDF:xRef ?id .
    ?keggDrug purl:title ?title .
    ?drug drugbank:chemicalFormula ?cf .
    FILTER(?cf = "C8H10N4O2")
}
```

Listing B.2: **AQ2:** Find all DrugBank drugs along with their target's label and chromosome location

```
SELECT DISTINCT ?drug ?label ?loc WHERE {
    ?drug drugbank:target ?target .
    ?target rdfs:label ?label .
    ?target drugbank:chromosomeLocation ?loc .
}
```

Listing B.3: **AQ3:** Find all Dailymed drugs along with their active ingredient and route of administration

```
SELECT DISTINCT ?drugLabel ?ingredientLabel ?route WHERE {
    ?drug dailymed:fullName ?drugLabel .
    ?drug dailymed:routeOfAdministration ?route .
    ?drug dailymed:activeIngredient ?ingredient .
    ?ingredient rdfs:label ?ingredientLabel .
}
```

Listing B.4: **AQ4:** Find all Dailymed drugs along with their active ingredient for which the route of administration is oral

```
PREFIX route:
    <http://www4.wiwiss.fu-berlin.de/dailymed/resource/routeOfAdministration/>
SELECT DISTINCT ?drugLabel ?ingredientLabel WHERE {
    ?drug rdfs:label ?drugLabel .
    ?drug dailymed:routeOfAdministration route:Oral .
    ?drug dailymed:activeIngredient ?ingredient .
    ?ingredient a dailymed:ingredients .
    ?ingredient rdfs:label ?ingredientLabel .
}
```

Appendix C

Experiment Results

Some experiment results could not be visualized within the result analysis. Table C.1 presents the query execution time of the different query engines on the simple queries from the LSLOD benchmark. The execution times are rounded to the fourth decimal place. Section 6.4 reports this results using bar charts. Table C.2 presents the number of results obtained by each engine for every query. This table was replaced by a smaller version (Table 6.4) due to the complete answers of all versions of Ontario despite the data format. Table C.3 and Table C.4 show the number of retrieved results and query execution time of the additional queries, respectively.

Query Engine	SQ1	SQ2	SQ3	SQ4	SQ5	SQ6	SQ7	SQ8	SQ9	SQ10
FedX	0.7674	0.6279	1.8250	113.3612	6.8227	2.3914	0.7237	5.1216	–	79.2462
ANAPSID	0.4119	2.3799	–	1.9615	2.0421	**0.6221**	0.0715	0.7437	**38.1960**	11.6569
MULDER	0.5309	**0.0479**	4.2815	**0.0635**	0.6743	0.8850	**0.0506**	0.9799	59.2715	7.7527
Ontario-RDF	0.5033	0.0541	0.5674	0.0721	0.7093	0.7691	0.0577	0.8654	58.5209	7.2970
Ontario-RDB	0.3586	0.4553	0.8694	1.2392	2.7043	0.6668	2.0979	0.6937	42.7880	2.4489
Ontario	0.5503	0.1256	**0.5149**	0.1416	0.5471	0.6816	2.3100	0.7355	56.9758	**2.4156**
Ontario-GH-RDF	0.5033	1.1560	0.5679	0.0710	0.7253	0.7547	0.0821	0.8684	58.7231	7.5339
Ontario-GH-RDB	**0.3553**	0.4515	0.8577	1.2460	2.6312	0.6804	1.7324	0.6878	42.6883	2.4371
Ontario-GH	0.5062	0.1376	0.5837	0.1321	0.5649	0.7042	2.2043	0.7404	58.3286	2.5436
Ontario-SSH-RDF	0.5147	0.0548	0.5640	0.0710	0.7520	1.6583	0.0560	1.6519	58.1081	7.5212
Ontario-SSH-RDB	0.3652	0.4305	0.8646	1.2467	2.7087	0.6763	1.7185	**0.6811**	42.5063	2.6146
Ontario-SSH	0.5173	0.1241	0.5667	0.1442	**0.5467**	0.7079	2.2930	0.7210	57.1575	2.4855

Table C.1: **Query Execution Time** in seconds, a dash representing time out or unanswered.

Query Engine	SQ1	SQ2	SQ3	SQ4	SQ5	SQ6	SQ7	SQ8	SQ9	SQ10
FedX	5,146	**1**	393	**24**	1,620	8,120	27	8,201	–	**8,708**
ANAPSID	5,146	3	**0**	28	1,620	8,120	27	8,201	524,694	9,174
MULDER	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-RDF	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-RDB	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-GH-RDF	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-GH-RDB	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-GH	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-SSH-RDF	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-SSH-RDB	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
Ontario-SSH	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174
ground truth	5,146	3	393	28	1,620	8,120	27	8,201	524,694	9,174

Table C.2: **Query Answer Cardinality**, a dash represents timed out queries.

Query Engine	AQ1	AQ2	AQ3	AQ4
Ontario-RDB	2	5,304	2,340	1,187
Ontario-GH-RDB	2	5,304	2,340	1,187
Ontario-SSH-RDB	2	5,304	2,340	1,187
ground truth	2	5,304	2,340	1,187

Table C.3: **Cardinality of Additional Queries**

Query Engine	AQ1	AQ2	AQ3	AQ4
Ontario-RDB	0.8316	0.7290	0.5250	0.3942
Ontario-GH-RDB	1.1573	0.7164	0.5208	0.4133
Ontario-SSH-RDB	1.0833	2.3282	2.1544	0.4887

Table C.4: **Query Execution Time of Additional Queries** in seconds

Bibliography

[1] Maribel Acosta, Maria-Esther Vidal, and York Sure-Vetter. "Diefficiency Metrics: Measuring the Continuous Efficiency of Query Processing Approaches". In: *The Semantic Web – ISWC 2017. ISWC 2017. Lecture Notes in Computer Science*. Ed. by C. d'Amato et. al. Vol. 10588. Springer, Oct. 2017, pp. 3–19.

[2] Maribel Acosta et al. "ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints". In: *International Semantic Web Conference* (2011).

[3] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. "Foundations of RDF databases". In: *Reasoning Web. Semantic Technologies for Information Systems. Reasoning Web 2009. Lecture Notes in Computer Science*. Ed. by S. Tessaris et al. Vol. 5689. Springer, June 2008.

[4] Sören Auer et al. "The BigDataEurope Platform – Supporting the Variety Dimension of Big Data". In: *Web Engineering. ICWE 2017. Lecture Notes in Computer Science*. Ed. by J. Cabot, R. De Virgilio, and R. Torlone. Vol. 10360. Springer, June 2017.

[5] François Belleau et al. "Bio2RDF: Towards a Mashup to Build Bioinformatics Knowledge Systems". In: *Journal of Biomedical Informatics* 41.5 (Oct. 2008), pp. 706–716. ISSN: 1532-0464. DOI: 10.1016/j.jbi.2008.03.004.

[6] Philip A. Bernstein. "Synthesizing Third Normal Form Relations from Functional Dependencies". In: *ACM Transactions on Database Systems* 1.4 (Dec. 1976), pp. 277–298.

[7] Dan Brickley and R. V. Guha. *RDF Schema 1.1*. W3C Recommendation. Feb. 2014. URL: https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[8] Matt Buranosky et al. "FDTool: a Python application to mine for functional dependencies and candidate keys in tabular data". In: *F1000Research* 7.1667 (2018).

[9] Gavin Carothers and Andy Seaborn. *RDF 1.1 N-Triples. A line-based syntax for an RDF graph*. W3C Recommendation. Feb. 2014. URL: https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[10] C. J. Date. *SQL and Relational Theory: How to Write Accurate SQL Code*. O'Reilly Media, 2015.

[11] Anastasia Dimou et al. "RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data". In: *Proceedings of the 7th Workshop on Linked Data on the Web*. Apr. 2014.

[12] Kemele M. Endris et al. "Ontario: Federated Query Processing against a Semantic Data Lake". In: *Database and Expert Systems Applications. Lecture Notes in Computer Science*. Springer, Cham, 2019.

[13] Kemele M. Endris et al. "Querying Interlinked Data by Bridging RDF Molecule Templates". In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXIX. Lecture Notes in Computer Science*. Ed. by Abdelkader Hameurlain et al. Vol. 11310. Springer, Berlin, Heidelberg, Nov. 2018, pp. 1–42.

[14] Jonas C. Ericsson. "Mediaton systems. An approach to retrieve data homogeneously from multiple heterogeneous data sources". Bachelor's Thesis. University of Gothenburg, 2009.

[15] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML Syntax*. W3C Recommendation. Feb. 2014. URL: `https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/`.

[16] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. New Jersey, Upper Saddle River: Prentice Hall, 2001.

[17] Behzad Golshan et al. "Data Integration: After the Teenage Years". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '17. Chicago, Illinois, USA: ACM, 2017, pp. 101–106.

[18] Olaf Görlitz and Steffen Staab. "SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions". In: *Proceedings of the Second International Conference on Consuming Linked Data - Volume 782*. COLD'11. Bonn, Germany: CEUR-WS.org, 2011, pp. 13–24.

[19] Rihan Hai, Sandra Geisler, and Christoph Quix. "Constance: An Intelligent Data Lake System". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 2097–2100.

[20] Alon Y. Halevy. "Answering Queries Using Views: A Survey". In: *The VLDB Journal* 10.4 (Dec. 2001), pp. 270–294.

[21] Ali Hasnain et al. "BioFed: Federated Query Processing over Life Sciences Linked Open Data". In: *Journal of Biomedical Semantics* 8 (Mar. 2017).

[22] Ali Hasnain et al. *BioFed Query*. SPARQL Endpoint. URL: `http://vmurq09.deri.ie:8007`.

[23] Yasar Khan et al. "One Size Does Not Fit All: Querying Web Polystores". In: *IEEE Access* 7 (2019), pp. 9598–9617.

[24] Donald Kossmann. "The State of the Art in Distributed Query Processing". In: *ACM Computing Surveys* 32.4 (Dec. 2000), pp. 422–469.

[25] Donald Kossmann and Konrad Stocker. "Iterative dynamic programming: a new class of query optimization algorithms". In: *ACM Transactions on Database Systems* 25 (2000), pp. 43–82.

[26] Ora Lassila and Ralph R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C Recommendation. Feb. 1999. URL: `https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[27] Maurizio Lenzerini. "Data Integration: A Theoretical Perspective". In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. PODS '02. ACM, 2002, pp. 233–246.

[28] Mohamed Nadjib Mami et al. "Towards Semantification of Big Data Technology". In: *Big Data Analytics and Knowledge Discovery*. Ed. by Sanjay Madria and Takahiro Hara. Cham: Springer International Publishing, 2016, pp. 376–390.

[29] John P. McCrae et al. *The Linked Open Data Cloud*. LOD Website. Mar. 2019. URL: `https://lod-cloud.net/`.

[30] Gabriela Montoya, Maria-Esther Vidal, and Maribel Acosta. "A Heuristic-Based Approach for Planning Federated SPARQL Queries". In: *Proceedings of the International Workshop on Consuming Linked Data (COLD)* (2012).

[31] Gabriela Montoya, Maria-Esther Vidal, and Maribel Acosta. "Optimal SPARQL 1.1 Queries for Federations of Endpoints".

[32] Gabriela Montoya et al. "Decomposing Federated Queries in Presence of Replicated Fragments". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 42.C (Jan. 2017), pp. 1–18.

[33] Gabriela Montoya et al. "Fedra: Query Processing for SPARQL Federations with Divergence". In: *CoRR* abs/1407.2899 (2014).

[34] Gabriela Montoya et al. "SemLAV: Local-As-View Mediation for SPARQL Queries". In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XIII. Lecture Notes in Computer Science*. Ed. by Abdelkader Hameurlain, Josef Küng, and Roland Wagner. Vol. 8420. Springer, Berlin, Heidelberg, Mar. 2014, pp. 33–58.

[35] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database systems*. 3rd ed. New York Dordrecht Heidelberg London: Springer, Feb. 2011.

[36] Eric Prud'hommeaux and Gavin Carothers. *RDF 1.1 Turtle. Terse RDF Triple Language*. W3C Recommendation. Feb. 2014. URL: `https://www.w3.org/TR/2014/REC-turtle-20140225/`.

[37] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. Jan. 2008. URL: `https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`.

[38] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. New York, NY, USA: McGraw-Hill, Inc., 2003.

[39] Matthias Samwald et al. "Linked open drug data for pharmaceutical research and development". In: *Journal of Cheminformatics* 3.1 (2011). ISSN: 1758-2946. DOI: `10.1186/1758-2946-3-19`.

[40] Franćois Scharffe et al. "Enabling linked data publication with the Datalift platform". In: *AAAI 2012, 26th Conference on Artificial Intelligence, W10:Semantic Cities*. Toronto, Canada, July 2012.

[41] Andreas Schwarte et al. "FedX: Optimization Techniques for Federated Query Processing on Linked Data". In: *ISWC 2011, Part I. LNCS*. Ed. by L. Aroyo et al. Vol. 7031. Heidelberg: Springer, 2011, pp. 601–616.

[42] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. "Heuristic and Randomized Optimization for the Join Ordering Problem". In: *The VLDB Journal* 6.3 (Aug. 1997), pp. 191–208.

[43] Petros Tsialiamanis et al. “Heuristics-based query optimisation for SPARQL”. In: *Proceedings of the 15th International Conference on Extending Database Technology* (2012), pp. 324–335.

[44] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* New York, NY, USA: Computer Science Press, Inc., 1988.

[45] Maria-Esther Vidal et al. “Efficiently Joining Group Patterns in SPARQL Queries”. In: *ESWC* (2010).

[46] Maria-Esther Vidal et al. “On the Selection of SPARQL Endpoints to Efficiently Execute Federated SPARQL Queries”. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXV*. Ed. by Abdelkader Hameurlain, Josef Küng, and Roland Wagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 109–149.

[47] Gio Wiederhold. “Mediators, concepts and practice”. In: *Studies Information Reuse and Integration In Academia And Industry*. Ed. by Tansel Özyer et al. Springer Verlag Wien, 2012, pp. 1–27.

[48] Gio Wiederhold. “Mediators in the architecture of future information systems”. In: *IEEE Computer* 25.3 (Mar. 1992), pp. 38–49.