RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN
INSTITUT FÜR INFORMATIK III

Master Thesis

Efficient Wrapper for Executing BGPs over Non-Semantic Data Using Spark

Author: Miguel Patricio MÁRMOL PANAMÁ
E-mail: marmol@cs.uni-bonn.de
Matriculation Number: 2739586

First Evaluator: Prof. Dr. Sören AUER
Second Evaluator: Prof. (Univ. Simón Bolívar) Dr. María Esther VIDAL
Supervisor: M.Sc. Mohamed Nadjib MAMI

A thesis submitted in fulfilment of the requirements
for the degree of Master of Science

in the

Enterprise Information Systems
Computer Science Department

May 2017

EIS
ENTERPRISE
INFORMATION SYSTEMS

universitätbonn

Declaration of Authorship

I herewith declare that all the work described within this Master thesis is the original work of the author. Any published (or unpublished) ideas or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Miguel Patricio Mármol Panamá

Signed: _____

Date: _____

"El sueño se hace a mano y sin permiso..."

Silvio Rodríguez Domínguez.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Abstract

During the last years the adoption of RDF has significantly increased, giving as a result that nowadays many semantic datasets are available. Nevertheless, there exists a vast amount of data which is not in RDF format. In order to publish it as RDF, it has to go through a process of providing semantics by mapping it to customized or existing ontologies and physically transforming it into RDF graphs. This process brings to face two challenges: Data replication and query optimization. With data constantly changing and growing, it is necessary to repeat the process to keep the RDF version of the data updated, which in the end is just impractical. Besides that, we need to consider the variety of data, i.e, the different formats the data can be expressed in. In this thesis, we focus on the problem of semantically querying non-semantic data, specifically large-scale structured data.

Our approach combines the usage of RML mapping to semantically enrich non-semantic data and big data data processing engines to deal with large amounts of data. We propose a *Data Wrapper*, which is an independent reusable data extraction component able to query multiple data sources, with only a minimal change. It enables to query the variety of data sources, with only one entry point: SPARQL BGP. Evaluation results demonstrate that our Data Wrappers are able to query data with increasing sizes while preserving quite reasonable performances.

Acknowledgements

I would like to express my deepest gratitude to all the people that colaborated with the development of this thesis, specially, to my advisor, Prof.(Univ. Simón Bolívar) Dr. María Esther Vidal, for her support, patience, and wise guidance throughout this process.

My most sincere gratitude goes towards M. Sc. Mohamed Nadjib Mami, without his motivation and support, accomplishing this goal would have been almost impossible.

I would like to extend my gratitude towards Prof. Dr. Sören Auer for giving me the facilities to successfully finishing this work.

I am grateful to the University of Bonn for giving me the possibility of fulfilling my dream of studying abroad and have a stunning international experience.

I would also like to thank all my friends for supporting me in the toughest moments of this process; to my family for believing in me, and for all the love and support.

Finally, I would like to thank Secretaría de Educación Superior, Ciencia, Tecnología e Innovación (SENESCYT) and Instituto de Fomento y Talento Humano, institutions from the Ecuadorian Goverment for giving me the opportunity of studying abroad by granting me a full scholarship.

Chapter 1

Introduction

This section briefly provides information about the research work conducted throughout this thesis. The first part describes the problem and the motivation behind it. The second part lists the contributions of it. The last part presents the structure of this document.

1.1 Problem Description and Motivation

The main purpose of Semantic Web is to interconnect data that is machine-processable and human-readable. The common language for representing information about resources in the Semantic Web is RDF (Resource Description Framework). It is particularly intended for enabling information exchange between applications without loss of meaning. RDF represents metadata about web resources in terms of simple properties and values [3].

In the last years, without any doubt, we have experienced a paradigm shift in the World Wide Web (WWW) as a result of the significant adoption of RDF. We have witnessed a huge increase in the amount of semantic data that is available on the Web in many different fields of human activity. Several knowledge bases with billions of RDF triples from Wikipedia, open government sites in the UK and in the US and others, news and entertainment sources, have been created and published online.

The real added-value we can get from the available semantic data relies on our capability of accessing it. The problem is that there exists a huge amount of data which is not in RDF format. It can be found in a structured form, such as in tabular format (e.g., CSV, DataBases), or semi-structured form, such as XML and JSON. Before being published, it has to go through a process of providing semantics by mapping it to customized

or existing ontologies and physically transforming them into RDF graphs. Once we have obtained the data in RDF format, we face one of the main challenges of RDF data management, which is query optimization. Due to the nature of RDF data model featuring interlinked triples (of the form *subject-predicate-object*) instead of records or entities, querying RDF data involves performing a significantly larger number of joins, which in the end significantly affects the query execution time.

In a 2001 research report, Doug Laney, defined the challenges of data growth in three aspects or dimensions: increasing volume (amount of data), speed of data in and out, and the range of data types and sources [4]. These three dimensions have since been used to describe Big Data, commonly known as Big Data's three Vs: *volume*, *velocity*, and *variety*.

Requiring that this heterogeneous and voluminous data to be transformed into RDF, or to any other format, is just impractical. Generating a new data would not only cause an extra space overhead, but also compromise data freshness. Let alone that using the chosen unique model (RDF in our case) and its corresponding data management system (triple store in our case) might not be the ideal *size* that *fits* all the varied use cases. Today, in the Big Data context, it is even the opposite that is trending. Meaning that the data is required to stay in its *original* format, and data transformation is shifted to the individual applications, i.e., only when needed and on only the needed portion of the data. We are referring here to the concept of Data Lake [1]. Data Lake is the pool of data that is collected and kept in its original format. Data is accessed following the "data on read" principle, meaning that the data is stored without any adherence to a prior schema, in contrast to what we traditionally had in Data Warehouse.

In order to access data in the Data Lake, one method is to build a top layer that provides a uniform view over the nonuniform data. The Semantic Web has a long track of success building data integration systems. Basically, the heterogeneous data is *mapped* to a unique ontology, using a set of mapping languages (available today for most common data types), to finally generate RDF data. We borrow such mapping techniques but refrain from generating RDF data; we rather deal with the data *virtually on the fly* as if it was in RDF. A Data Lake accessed using this way is called a *Semantic Data Lake*.

We are proposing what we call *Data Wrappers*. A Data Wrapper is an independent data extraction unit that wraps a data source providing two interfaces: *input* and *output*. In input, a Data Wrapper accepts a SPARQL query, more precisely its BGP part. As output it returns data fulfilling the input BGP. This way, a user can have a uniform access to the Data Lake.

[1] First appearing here: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/

To realize the full Data Lake, a higher level querying system (outside of our scope) can be built on top, using our Data Wrappers to access the heterogeneous data uniformly by submitting a set of triple patterns. Such a system must take care of picking the relevant Data Wrapper for a given query, and also of reconciling (e.g., union or join) the results returned from the individual Data Wrappers.

1.2 Contributions

The main contribution of this thesis is then to implement the concept of Data Wrappers with the following characteristics:

- Employing the RML mapping language [5] to give non-semantic data a semantic description that can be used for querying. The Data Wrapper is able to analyze an RML mapping provided along with the data, and to generate a query plan that is executed to extract data given a query (BGP).
- The Data Wrapper is able to process large scale data on commodity machines, thanks to the set of big data technologies: Apache Spark Spark[2] and Apache Hadoop[3], used for its implementation.
- An important aspect of our Data Wrappers is their re-usability with a minimal effort. For example, the same Data Wrapper can be used to query CSV files, Parquet tables as well as databases (SQL and NoSQL). Only specifying the data type and location as part of the RML mappings.

1.3 Thesis Structure

This thesis is structured around six chapters as follows:

- *Chapter 1* starts with the introduction and the motivation that lead to this work.
- *Chapter 2* describes the theoretical background related to the focused area of this thesis.
- *Chapter 3* presents the state of the art works that have been done in this field.
- *Chapter 4* details the development of the proposed solution.

[2] Apache Spark
[3] Apache Hadoop

- *Chapter 5* discusses the data sets and the queries used to test the performance of our approach.
- *Chapter 6* provides conclusions, limitations, and future work of this thesis.

Chapter 2

Background

This chapter presents the necessary background laying the foundations behind the work. The section begins with introducing some key concepts, like RDF and SPARQL. The next section presents the mapping languages R2RML[1] and RML [5]. The third section describes Apache Spark[2, 6–8] as well as its building blocks used for the implementation of our solution. Finally, we demonstrate one prominent use scenario of our work, namely Data Lake.

2.1 The Semantic Web

The idea of adding semantics to the world wide web was proposed in the *Scientifc American* article by *Tim Barners-Lee*[1], et. al, where they stated the vision of what they called the *Semantic Web*. The main point is this fragment [9]: "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation"

The Semantic Web is based on the following four Linked Data principles [10]:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide usefu.l information, using the standards (RDF, SPARQL).
- Include links to other URIs, so that they can discover more things.

[1] https://www.w3.org/People/Berners-Lee/

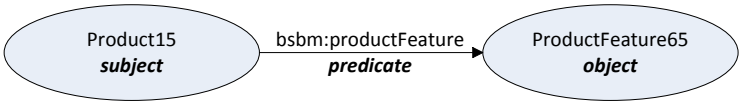

FIGURE 2.1: RDF triple basic structure.

2.1.1 RDF: Resource Description Framework

The task of the Semantic Web is represent knowledge providing a model and syntax to exchange machine-processable and human-readable information. The Resource Description Framework (RDF) [3] is a model that represents data through triples. An RDF triple is the building block of the RDF data model. An RDF Graph is a set of triples. An RDF triple is formed by *subject, predicate, and object.* The *predicate*, also known as *property*, represents the relation between the *subject* and the *object.*

A resource is anything from the real world that we want to represent in RDF, for instance: a person, an institution, the relation that the person has with this institution. In the context of RDF, a resource is uniquely identified by a URI (Universal Resource Identifier) [3].

Figure 2.1 shows an RDF graph composed by only one triple. The triple consists of a subject (*Product15*), and a object(*Product65*), connected through a predicate (*productFeature*). This helps to represent the fact that Product15 posses the feature ProductFeature65.

In our work we are interested only in ground triples, therefore, blank nodes have been ignored [11]. From this point on, and without loss of generality we will be using the term triples to refer to ground triples. In this context, an RDF Graph is a set of ground triples.

There are several serialization formats for RDF:

- **Turtle:** allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes [12]. Human-friendly format.
- **N-Triples:** is a line-based, easy to parse, plain-text format. N-Triples is a subset of Turtle, not compact as it [13].
- **N-Quads:** line-based, plain text format [14].
- **JSON-LD:** JSON-based format [15].

- **RDF/XML:** XML-based syntax that was the first standard format for serializing RDF [16].

In this thesis, we will use Turtle and N-Triples serialization formats.

2.1.2 SPARQL

SPARQL [17] (acronym for SPARQL Protocol and RDF Query Language) is the official W3C Recommendation language for querying RDF graphs. SPARQL is based on the concept of matching graph patterns. The simplest graph patterns are triple patterns, which are similar to an RDF triple but they could have variables in the subject, predicate, or object positions. A **Basic Graph Pattern (BGP)** is a conjunction of triple patterns. A BGP matches a subgraph of the RDF graph data when variables of the graph pattern can be substituted with the RDF terms in the graph and the result is and RDF graph equivalent to the subgraph.

A SPARQL query consists of a pattern matching, which includes several features of pattern matching of graphs, like optional, union of patterns, nesting, filtering values of possible matchings, and the possibility of choosing the data source to be matched by a pattern. The solution modifiers, which once the output of the pattern has been computed, allow to modify these values by applying classical operators like projection, distinct, order, limit, and offset. Finally, the output of a SPARQL query can be of four different forms: yes/no (ASK), selections of values of the variables (SELECT), construction of new triples from these values (CONSTRUCT) and description of resources (DESCRIBE).

The syntax of a SPARQL query is SQL-like, it follows the select-from-where paradigm. The **select** clause specifies the variables that will appear in the query results, each variable has the character "?" as prefix. In the **where** clause the graph patterns of the query are set. Finally, a **filter** expression explicitly specifies a condition on query variables.

Listing 2.1 presents a SPARQL query that selects the label, the comment and the product of a set of products that feature productFeature1.

```
1 PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT
4 *
5 WHERE
```

```
6  {
7     ?product rdfs:label ?label.
8     ?product rdfs:comment ?comment.
9     ?product bsbm:productFeature <http://example/productfeature1>.
10 }
```

Listing 2.1: SPARQL query.

2.1.2.1 Query Shapes

The shape of a SPARQL BGP can have serious impacts on query performance [18]. The most common BGP patterns are: star, linear, and snowflake (Figure 2.2). The diameter of a SPARQL BGP is defined as the longest path, i.e., longest connected sequence of triple patterns.

Star-shaped patterns have a diameter of one and occur frequently in SPARQL queries. As the join variable is in subject position, they are characterized by *subject-subject* joins between triple patterns.

Linear or *path-shaped* are also very common queries. These patterns are made of *subject-object* (or *object-subject*) joins, i.e., the join variable is on subject position in one triple pattern and on object position in another triple pattern.

Snowflake-shaped patterns are combinations of several star-shapes, usually connected by short paths. More complex structures can be obtained combining the aforementioned shapes.

In this thesis, we are only considering basic graph patterns; additionally, we exclusively work with SELECT form queries and basic query shapes.

2.2 Mapping Languages

2.2.1 R2RML: RDB to RDF Mapping Language

R2RML [2], a W3C Recommendation, is a language for expressing customized mappings from relational databases to RDF datasets.

An R2RML mapping is associated with logical tables. A Logical Table **(rr:logicalTable)** can be one of the following:

[2] https://www.w3.org/TR/r2rml/

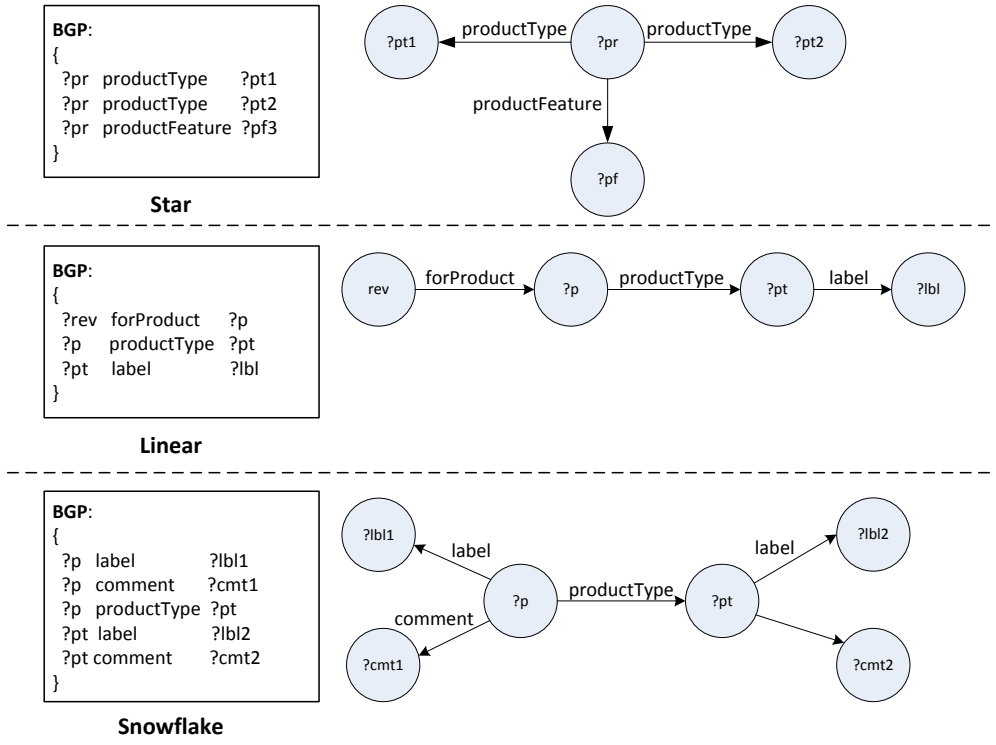

FIGURE 2.2: SPARQL BGP query shapes.

- A table **(rr:tableName).**
- A view, or valid SQL query **(rr:sqlQuery).**

Each logical table is mapped to RDF using a Triples Map that associates each row in the logical table with a number of RDF triples. The Triples Map is composed of three main parts:

- The Logical Table, aforementioned.
- A Subject Map **(rr:subjectMap)** that generates the subject of all RDF triples that will be generated from a row in the logical table.
- Zero or more Predicate-Object Maps **(rr:predicateObjectMap)**.

A Predicate-Object Map consists of Predicate Maps **(rr:predicate)**, which define the rule that generates the predicate of the triple and Object Maps **(rr:objectMap)** or Referencing Object Maps **(rr:RefObjectMap)**, which define the rule that generates the object of the triple.

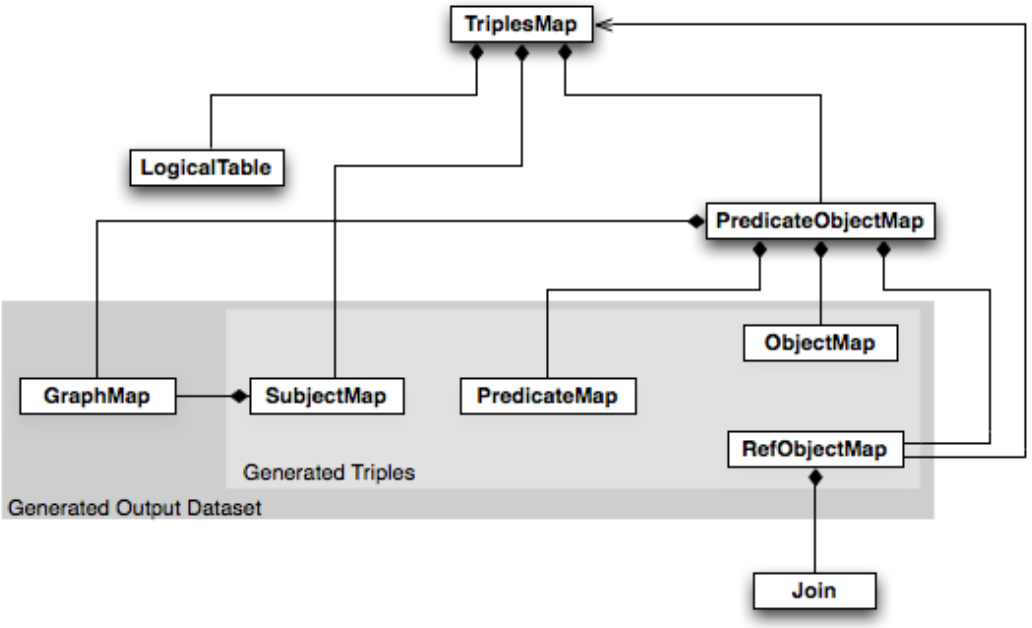

FIGURE 2.3: R2RML overview [1].

The Subject Map, the Predicate Map, and the Object Map are Term Maps **(rr:TermMap)** that generate and RDF Term (an IRI, a blank node, or a literal). A Term Map can be a constant **(rr:constant)**, or a reference to a column **(rr:column)** from the Logical Table.

R2RML supports cross-references between Triples Maps using a Referencing Object Map, which allows for using the subject of another Triples Map as the object generated by a Predicate-Object Map. If both triples come from different logical tables a join between them is necessary. The Join Condition **(rr:joinCondition)** performs the join in the same way as a join executed in SQL. The Join Condition has a reference to a column name **(rr:child)** that exists in the Logical Table of the Triples Map that contains the Referencing Object Map and a reference to a column name that exists in the Logical Table of the Parent Triples **(rr:parent)** Map of the Referencing Object Map.

Figure 2.3 shows an overview of the R2RML mapping components.

2.2.2 RML Mapping Language

The RDF Mapping Language[5], presented by Dimou et. al, is a generic mapping language defined to express customized mapping rules from heterogeneous data structures and serializations to the RDF data model. It is an extension of R2RML[3], the W3C Recommendation for mapping data in relational databases into RDF.

[3] R2RML: RDB to RDF Mapping Language

RML extends the applicability of R2RML not only for data in relational databases, but broadening the scope of R2RML to include other data formats like CSV, JSON and XML.

In order to achieve this purpose the following components of R2RML have been extended by RML:

- Logical Source extends Logical Table from R2RML to determine the input source that contains the data to be mapped. Instead of specifying the Table Name, it specifies the Source **(rml:source)**.
- R2RML uses the property **(rr:column)** to define a column-valued Term Map. RML introduces **rml:reference**, a more generic property which value must be a valid reference to the data of the input dataset.

Additionally to the aspects extended from R2RML, RML has its own following components:

- R2RML only accepts a database as input, RML introduces Reference Formulation **(rml:referenceFormulation)** to support a broader type of inputs. Currently, **ql:CSV** for CSV files, **ql:XPath** for XML files and **ql:JSONPath** for JSON files Reference Formulation are predefined.
- In R2RML is clear that a per row iteration occurs, RML presents a iterator **(rml:iterator)** that needs to be defined according to the structure of the input data.

Figure 2.4 depicts the components of RML. In blue the components from R2RML that have not been modified, in green the components from R2RML that have been extended and finally in red the specifics components.

For further details about RML, we refer to [5], where the authors provide a deeper description.

2.3 Apache Spark

Apache Spark, presented by Zaharia et. al [6, 7], is a fast general-purpose in-memory cluster computing engine for large-scale data processing with APIs in Scala, Java and

[4] RML Generic Mapping Language

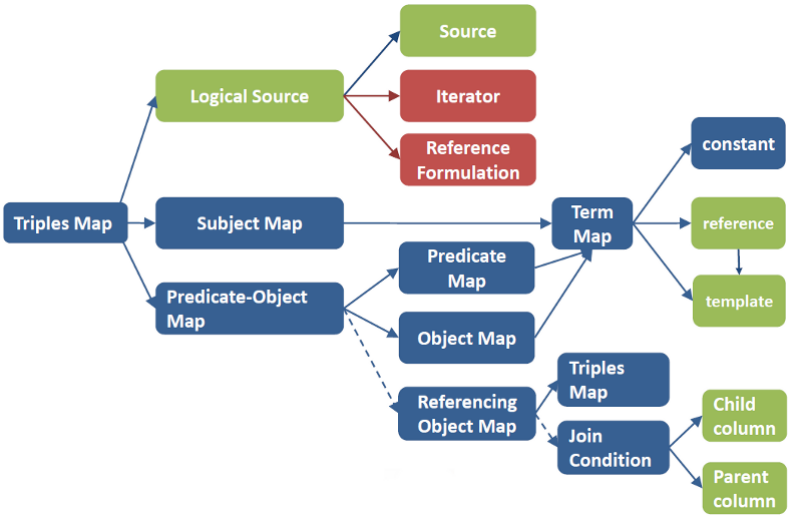

FIGURE 2.4: RML components. Figure taken from RML site[4].

Python and libraries for streaming, graph processing and machine. It can run on Hadoop and process data from any Hadoop data source.

The main data structure is called *Resilient Distributed Dataset* (RDD) [2] which is a fault-tolerant distributed collection of elements that can be operated in parallel. Each RDD is split into multiple partitions, that may be computed on different nodes of the cluster. RDDs can contain any type of objects.

RDDs offer two types of operations: transformations and actions. *Transformations* construct a new RDD from a previous one, e.g. filtering data that matches a word. *Actions*, on the other hand, compute a result based on an RDD, e.g., returning the first element of an RDD. Table 2.1 enumerates the main transformations and actions provided by Spark.

The Spark project contains multiple closely integrated components in a single unified data processing framework. It comes with a rich stack of high-level tools for batch processing, complex analytics, interactive exploration, graph and real-time stream processing as well a relational interface called Spark SQL [8]. This latter gives the possibility to work with structured data. It allows querying data via SQL and it supports many sources of data, including CSV, Parquet, and JSON.

In this thesis, we have used the transformations: Map, union and join, as well as actions like count, collect from the RDD operations. We have also used the Spark SQL interface.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction: \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V,V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f: \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p: \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (T, T) \Rightarrow T)$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K,V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : String)$	:	Outputs RDD to a storage system, e.g., HDFS

TABLE 2.1: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T. Table taken from [2].

2.4 Data Lake

The term data lake was coined by *James Dixon* [19] when he wrote in his blog: "If you think of a data mart as a store of bottled water – cleansed and packaged and structured for easy consumption – the **Data Lake** is a large body of water in a more natural state. The contents of the data lake stream in from a source to fill the lake, and various users of the lake can come to examine, dive in, or take samples" .

A Data Lake is a storage repository that contains a great amount of raw data that comes from multiple data sources in its native format until it is needed. It uses a flat architecture to store data. When a business question emerges, the data lake can be queried for relevant data.

The Data Lake supports the following capabilities[5]:

- Capture and store raw data at scale with a low cost.
- Store many types of data in the same repository.
- Perform transformations on data.
- Define the structure of the data at the time it is used, referred to as schema on read.
- Perform new types of data processing.
- Implement single subject analytics based on very specific use cases.

[5] Putting the Data Lake to Work. A Guide to Best Practices

Chapter 3

Related Work

This chapter presents the most relevant approaches related to the aim of this thesis. The section begins with presenting several approaches that benefit from big data processing engines to handle semantic-data and semantically enriched non-semantic data. Finally, we discuss RML mapping language [5].

3.1 Querying Big Data

There has been a large body of research tackling semantic data processing in large scale [20], [21], [22], [23], [24], [25], [26]. However, only a few approaches considered data integration problem of heterogeneous data sources using Semantic Web practices, the way we approach it. We list them and describe how they are related to our approach.

3.1.1 SeBiDA

SeBiDA (Semantified of Big Data) [27] is the work presented by Mami et. al. The authors introduce an approach to manage hybrid Big Data. SeBiDA applies semantic enrichment to non-semantic data as we do, but the main difference lays on that data is physically converted into RDF data in a mandatory pre-processing phase. Storage scheme of the loaded data is tabular using Apache Parquet[1]. To query those tables, they convert SPARQL queries into SQL. We go the opposite direction by not storing at all any data physically, we instead query it *on the fly*, virtually.

[1] Apache Parquet

3.1.2 GEMMS

GEMMS: *A Generic and Extensible Metadata Management System for Data Lakes* is the work presented in [28], where the authors describe a system that extracts metadata from the sources and manages the structural and semantic information in an extensible metamodel. The main difference with our approach is that they detect the file type using Apache Tika[2]. We explicitly indicate the type of source using RML mapping language. Another important difference is metadata managament. Our Data Wrappers can produce different results with the same data by just changing the RML mapping.

3.1.3 Personal Data Lakes

In Personal Data Lake with Data Gravity Pull [29], the authors present a unified storage facility for storing, analyzing and querying personal data. In this approach they put emphasis on metadata management, whereas our work relies on the RML mapping provided by the user.

3.2 RML Processing

RML [5] is highly extensible to support new source formats allowing different input sources in a uniform way. RML relies on expressions in a target expression language relevant to the source format to refer to the values of the sources. This target expression language is bounded to its format and act as a point of reference to the values in the source. Expressions are located in Term maps and rr:iterator to tell how to access the values; therefore, they should be valid according to the formulation specified in rr:referenceFormulation.

An RML Processor can be implemented using two alternative models: mapping-driven, data-driven, or a combination of both.

1. **Mapping-driven:** In this model, the processing is driven by the mapping module. The processor processes each triples maps in consecutive order. Based on the Reference Formulation, each Triples Map is delegated to a language specific sub-extractor. For each Triples Map, the corresponding sub-extractor iterates over the source according to the Triples Map's Iterator. For each iteration the mapping module requests an extract of data from the extraction module. The defined Subject Map and Predicate-Object Maps are applied and the corresponding triples are

[2] Apache Tika

generated. The execution of dependent Triples Maps, because of joins, is triggered by the Parent Triples Map and a nested mapping process occurs.

2. **Data-driven:** In this model, the processing is driven by the data sources. The processor extracts beforehand the iteration patterns from the triples maps. Each dataset is integrated by its language-specific sub-extractor. Based on the Reference Formulation and the iterator, each Triples Map is delegated to a specific sub-mapper. For each iteration, a data extract is passed to the processor, which in turn, delegates the extract of data to the corresponding sub-mapper. The defined Subject Map and Predicate-Object Maps are applied and the corresponding triples are generated. The execution of dependent Triples Maps, because of joins, is triggered by the Parent Triples Map and a nested mapping-driven process occurs.

   The authors implemented a RML Processor using the mapping-driven model.

   RML Processor is capable of semantifying several non-semantic data sources at once, but they do not offer any alternative to access the data by means of querying it. On the other hand, our approach also semantifies several non-semantic data sources at once (See Section 4.1.1) and besides that we offer access to the data using a SPARQL query (BGP).

Chapter 4

Proposed Solution

In order to tackle the problems presented in the introductory part of this work (Chapter 1), a prototype[1] has been implemented, named Data Wrapper. In this chapter, we provide details about its construction. The first section discusses the requirements that a Data Wrapper must comply to. Then, we give an overview over the architecture of a Data Wrapper. Finally, we describe the functionality of each component of the architecture.

4.1 Requirements

This section illustrates the necessary requirements that the implementation needs to achieve. They have been classified as functional and not functional.

4.1.1 Functional requirements

This classification is related to the requirements that our Data Wrapper must fulfill to be able to execute BGPs over semantically described, non-semantic structured data.

1. Use RML mapping language to associate a semantic description with a non-semantic data with.
2. Support semantic querying *on the fly* over semantically enriched (non-semantic) structured data.

[1] Data Wrapper

3. Provide two types of Data Wrappers: *Dynamic*, uses RML mappings to query (multiple) non-semantic data; *Static*, uses the RML mappings to obtain the semantified (RDF) version of the *whole* non-semantic data.

4. Allow users to to easily change execution settings (data type, data location, RML mappings location, Data Wrapper type) without changing the application itself, through the use of a human-readable configuration file.

4.1.2 Non-functional requirements

The non-functional requirements explain how the Data Wrapper should behave.

1. **Processing engine** The Data Wrapper should be built using an efficient large-scale processing engine.

2. **Versatility** The Data Wrapper should be able to easily be adapted to support multiple data sources, e.g., CSV, Parquet and databases.

3. **Performance** The Data Wrapper should fulfill its promises (executes a BGP efficiently) withing reasonable times. We deem reasonable in our context, i.e., query processing, to be around few minutes at most.

4. **Scalability** The Data Wrapper should preserve reasonable performances even with increasing sizes of data.

5. **Distribution and Fault-tolerance** The Data Wrapper should be able to process (query) data in large-scale distributed environments, with tolerance to data loss and processing failure, due to, e.g., machine failure.

4.2 Architecture

Figure 4.1 shows an overview of the architecture of the prototype. For a proper comprehension, it has been separated in several components as follows:

1. Bootstrapper.

2. SPARQL BGP Parser.

3. RML Parser.

4. SPARQL BGP Rewriter.

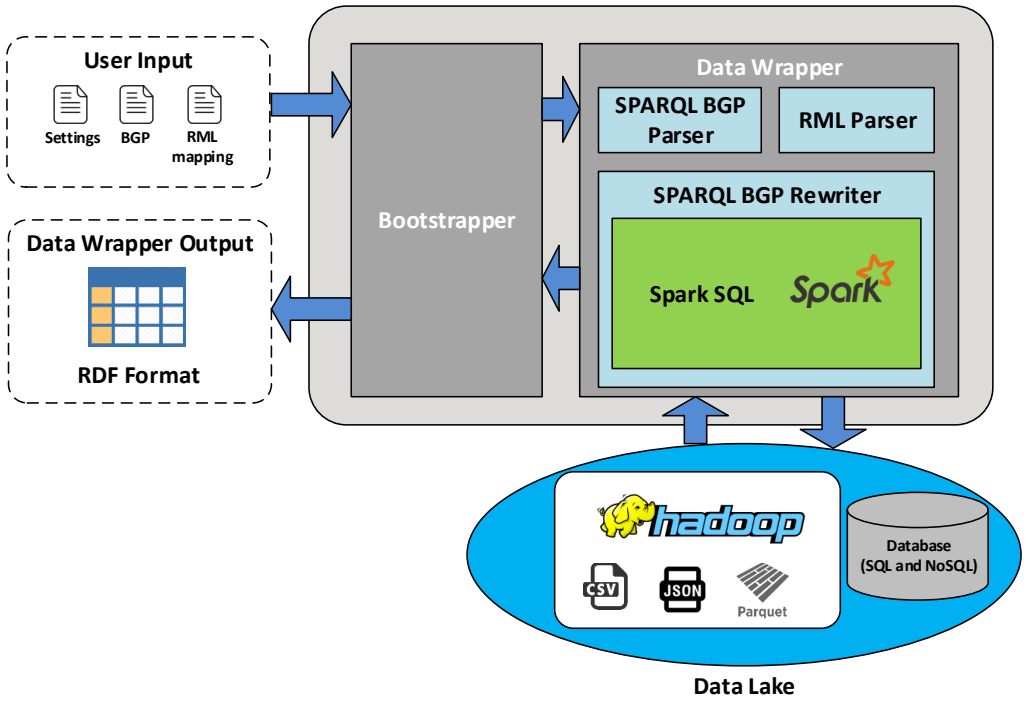

FIGURE 4.1: Components of the the prototype architecture.

4.3 Components

In this section, we discuss the different components implemented as part of this thesis. The prototype has been implemented using Apache Spark in Java and Scala, it also uses libraries from RML mapping language and Apache Jena[2]. The role that each component plays in our approach is discussed next.

4.3.1 Bootstrapper

This component reads from the user the needed arguments to bootstrap the application. This includes (path to) the *Settings file*, which in turn keeps information about the Data Wrapper. JSON is selected to compose the settings file as it is both machine- and human- readable format. The JSON setting file is parsed using a special JSON Java library *JSON.simple*[3]. At the end, depending on whether the type is Static or Dynamic, the Bootstrapper instantiates the corresponding Data Wrapper (class).

4.3.1.1 Input files

The user will provide three files:

[2] Apache Jena
[3] JSON.simple java library

```
{
 "rml-file" : "sources/rml/csv/bsbm/bsbm.ttl",
 "spark-server" : "spark://12.34.56.78:3077",
 "BGP-file" : "sources/sparql/csv/bsbm/bsbm_1.sparql",
 "wrapper-type": "dynamic"
 "jdbc-properties" : "sources/jdbc/database.properties"
}
```

FIGURE 4.2: Settings file.

1. **Settings** A JSON file that has the structure shown in diagram 4.2. It has six keys:
   - *rml-file*: Location of the RML mapping file.
   - *spark-server*: The master URL [4] of the currently running Spark cluster.
   - *BGP-file*: Text file that contains the BGP to execute, in case of Dynamic Data Wrapper.
   - *wrapper-type*: The value of this key is either 'static' or 'dynamic'. If it is 'static', then the BGP setting is ignored as it is not needed (to materialize the whole data in RDF format).
   - *jdbc-properties*: In case the data source is a database, this setting contains the details needed to connect to that database.
2. **BGP** A file that contains the SPARQL BGP to execute over the data source.
3. **RML Mapping** A file that contains the RML mappings of the data source.

4.3.2 SPARQL BGP Parser

We have used Apache Jena to serialize the SPARQL BGP passed to the Data Wrapper. Figure 4.3 shows the class diagram used to represent the graph obtained from the SPARQL BGP. All of our classes implement the interface *Serializable*[5].

This component is invoked when the method *evaluate()* from class *QueryGraph* is called. The parsing of the SPARQL BGP is made using the Apache Jena library (class *ApacheJenaQuery* in Figure 4.3), by calling the method *parseBGP()*. Algorithm 1 takes as input this Apache Jena Query ***query***. Then, an object, named ***graph***, from class *Graph* is instantiated. Our initial prototype only supports SELECT queries, thus an enumerator with value *SELECT* is passed to the constructor (line 1). The values for *distinct*,

[4] Spark Master URLs Format
[5] Interface Serializable

limit and *offset* are taken from ***query*** to ***graph*** (line 2), respectively. The value for *projectedVars* is the list of variables that are part of the select statement (line 3).

Algorithm 1: SPARQL BGP Parser

Input : Apache Jena Query *query*

Output: Graph *graph*

```
1  graph ← new Graph(type.SELECT);
2  graph.distinct ← query.distinct; graph.limit ← query.limit; graph.offset ← query.offset;
3  graph.projectedVars ← query.projectVars;
4  id ← 1;
5  forall element tp in query.triplePatterns do
6      triple ← ConverTriplePattern(tp, id);
7      Add triple to the list of triple patterns; id ← id + 1;
8  end
9  IdentifiyGenericVars();
10 return graph;
11 Function ConverTriplePattern (triplePattern, id)
12     s, p, o ← null;
13     sC, pC, oC ← false;
14     if triplePattern.subject.isVariable then
15         s ← new Variable(triplePattern.subject.name);
16     else
17         s ← new Value(triplePattern.subject.name); sC ← true;
18     end
19     if triplePattern.predicate.isVariable then
20         p ← new Variable(triplePattern.predicate.name);
21     else
22         p ← new Value(triplePattern.predicate.name); pC ← true;
23     end
24     if triplePattern.object.isVariable then
25         o ← new Variable(triplePattern.object.name);
26     else
27         o ← new Value(triplePattern.object.name); oC ← true;
28     end
29     tp ← new TripplePattern(id, s, sC, p, pC, o, oC);
30     return tp;
```

Method *convertTriplePatterns()* from class *QueryGraph* is called to create an object, named ***tp***, from class *TriplePattern* for each triple pattern that composes the SPARQL BGP (line 6). If the subject of the triple pattern is a variable, an object from class

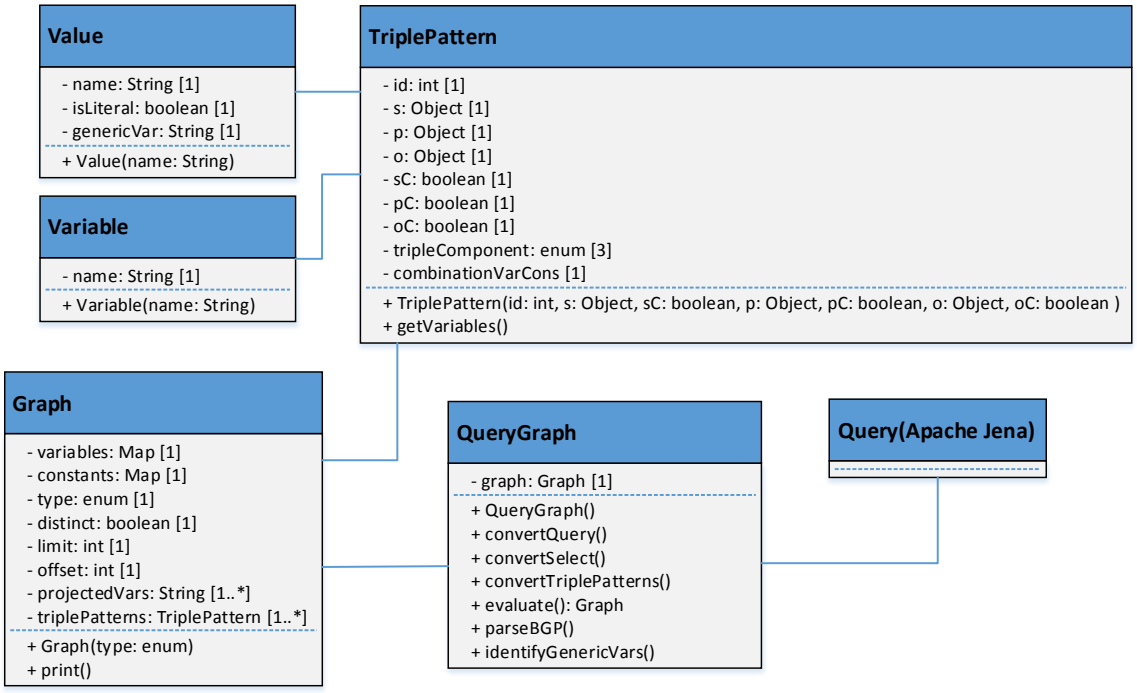

FIGURE 4.3: Class diagram for representing the graph obtained from the SPARQL BGP.

Variable is assigned to attribute *s*; otherwise, an object from class *Value*. The same criteria applies for attributes *p* and *o*. Attributes *sC*, *pC*, and *oC* get a true value if the corresponding position is a constant (lines 12 - 28). Attribute *combinationVarCons* represents the pattern of the triple pattern in terms of the position of the constants and variables, e.g., CCV means subject and predicate are constants, meanwhile object is variable. In total, we have eight possible combinations (Table 4.1).

Once all the attributes have been set, ***tp*** is added in the list of triple patterns (attribute *triplePatterns*) of ***graph*** (line 7). The maps data structures (*variables*, *constants*) are filled in with the corresponding variables and constants from all the set of triple patterns. Since we can have the same constant or variable in more than one triple pattern, they are only added once to their respective map data structure. Method *identifyGenericVars()* from class *QueryGraph* is called (line 9) to identify whether or not exist any URIs on subject and object positions. If there are, attribute *genericVar* from class *Value* receives a unique identifier for each distinct URI. Finally, ***graph*** is returned to the *Loader* (section 4.3.1) for further processing (line 10).

4.3.3 RML parser

We have implemented our own classes for RML, because *map* transformation from SPARK SQL (Table 2.1) requires objects from classes that implement *Serializable* interface and none of the classes from the RML library did this. Figure 4.4 shows the class diagram for our RML representation.

The interface *IRMLParser* has an attribute *tripleMaps*, which represents the list of triple maps inside the RML mapping file (Section 4.3.1). Class *StructuredData* implements interface *IRMLParser* for non-semantic structured data. This interface could be implemented for supporting semi-structured data in future work (Section 6.3).

Algorithm 2: RML mapping Parser

Input : RMLMapping *mapping*

Output: Collection of triple maps *tripleMapsList*

```
1  Let tripleMapsList be the collection of triple patterns;
2  forall element tpMap in mapping.triplesMaps do
3      tm ← new TripleMapGW();
4      logicalSource ← new LogicalSource();
5      ds ← new DataSource();
6      ds.sourceLocation ← mapping.triplesMaps.logicalSource.Source.template;
7      logicalSource.ds ← ds;
8      tm.logicalSource ← logicalSource;
9      subjectMap ← new SubjectMapGW();
10     subjectMap.template ← tpMap.subjectMap.template;
11     subjectMap.classURI ← tpMap.subjectMap.classIRIs;
12     tm.subjectMap ← subjectMap;
13     forall element predicateObjectMap in tpMap.predicateObjectMaps do
14         Let predicatesList be a list of objects from class PredicateObjectMapGW;
15         if predicateObjectMap.hasReferencingObjectMaps then
16             predicatesList ← ProcessReferencingObjectMaps(predicateObjectMap);
17         else
18             predicatesList ← ProcessNonReferencingObjectMaps(predicateObjectMap);
19         end
20         Add predicatesList to the attribute tm.predicateList;
21     end
22     Add tm to tripleMapsList;
23 end
24 return tripleMapsList;
```

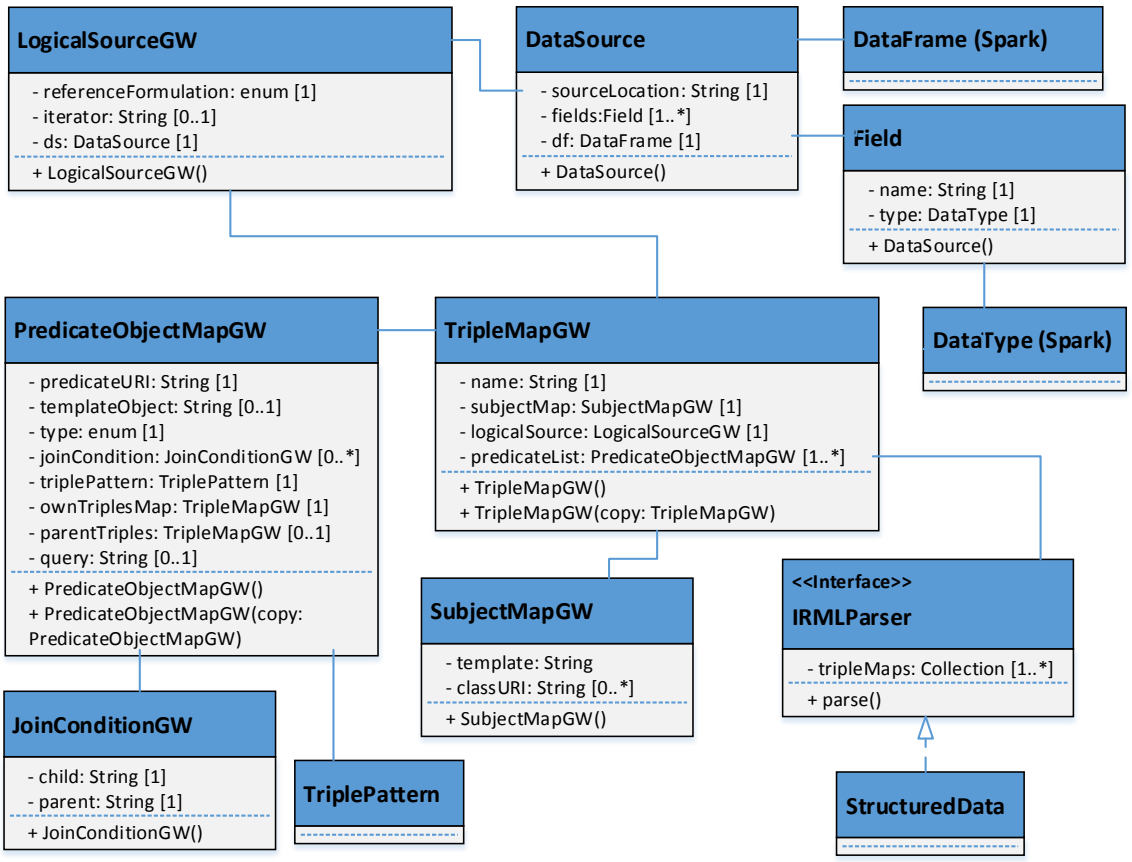

FIGURE 4.4: Class diagram for representing RML mapping language.

Method *parse()* is in charge of using the RML mapping language library to serialize the RML mapping file. Algorithm 2 takes this latter as input (***mapping***) and returns the list of triple maps, ***tripleMapsList***. It traverses this latter (lines 2, 20) and for each element ***tpMap*** creates an object, named ***tm***, from class *TripleMapGW* (line 3).

An object, named ***logicalSource***, from class *LogicalSource* is instantiated. Attribute *referenceFormulation* is set with value *CSV_ CLASS* for CSV files; or *JDBC_ CLASS* for databases; or *JSONPATH_ CLASS* for JSON files; or *PARQUET_ CLASS* for parquet files. Attribute *iterator* only exists for XML and JSON formats. An object, named ***ds***, from class *DataSource* is instantiated, it contains the path to the data source (attribute *sourceLocation*), the list of fields extracted from the data source according to its iterator, and a SPARK DataFrame (class *DataFrame (Spark)*). Finally, ***logicalSource*** is set in ***tm*** (lines 3-8).

An object, named ***subjectMap***, from class *SubjectMapGW* is instantiated, attributes *template* and *classURI* are set. Finally, ***subjectMap*** is set to ***tm*** (lines 9 - 12).

The next step is loop the predicate object map set (lines 13,21). If element ***predicateObjectMap*** has a reference object map, function ProcessReferencingObjectMaps() is

called, otherwise function *ProcessNonReferencingObjectMaps* (lines 15 - 19). Next, the list containing objects from class *PredicateObjectMapGW* is added to attribute *predicateList* of ***tm*** and finally this latter is added to *tripleMapList* (lines 20 - 22). Finally, it is returned to the *Bootstrapper* component (section 4.3.1) for further processing (line 24).

Algorithm 3 takes as input an RML *PredicateObjectMap* object (***pom***) that has a *ReferencingObjectMap*. It starts iterating this set (lines 2, 21). Then, for each element ***rom***, we traverse the set of predicates Map (lines 3, 20). For each *PredicateMap* ***predicateMap***, an object named ***predicate***, from class *PredicateObjectMapGW*, is instantiated. Attributes *predicateURI*, *parentTriple* and *ownTriplesMap* are set (lines 4 - 7).

If the size of the set join conditions of ***rom*** is bigger than 1, we assign to enumerator *type* the value *REFERENCE_MAP_JOIN* (line 9) and traverse the set of join conditions. Otherwise the value *REFERENCE_MAP_SELF* is assigned to the enumerator (line 17). When traversing the set of join conditions, an object named ***joinCondition***, from class *JoinConditionGW* is created for each element inside the set. Atributtes *children* and *parent* are set and the object is added to the list of join conditions of the object ***predicate*** (lines 10 - 14). Finally, it is added to collection ***predicateList*** and is returned to Algorithm 2 (lines 19 - 22).

Algorithm 4 takes as input an RML *PredicateObjectMap* that does not have a *ReferencingObjectMap*. We traverse the predicate map set and for each element we create an object named ***predicate***, from class *PredicateObjectMapGW* (lines 2 - 3). Attributes *predicateURI*, *ownTriplesMap* are set (lines 4 - 5). We traverse the set of object maps. If the object map is a *ReferenceMap*, the enumerator *type* receives the value *REFERENCE* and we assign the value of the reference map to the attribute *templateObject* (lines 8 - 9). Otherwise, if the o bject map has a value in its *template*, then, we assign this to attribute *templateObject*. Finally, ***predicate*** is added to collection ***predicateList*** and is returned to Algorithm 2 (lines 13 - 16).

4.3.4 SPARQL BGP to SQL Rewriter

In this section, we discuss the approach followed to build the component that rewrites the SPARQL BGP into SPARK SQL and executes it afterwards. Our approach considers SPARQL join queries that are of the form:

$$\begin{aligned} &\text{SELECT} \quad ?v_1, ?v_2, ?v_3, ..., ?v_n \\ &\text{WHERE} \quad \{tp_1 \ . \ tp_2 \ . \ ... \ . \ tp_m\} \end{aligned}$$

Algorithm 3: Process Referencing Object Maps

Input : RML PredicateObjectMap *pom*
Output: Collection of triple patterns *triplePatternsList*

```
1  Let predicateList be the collection of PredicateObjectMapGW;
2  forall element rom in pom.referencingObjectMaps do
3      forall element predicateMap in rom.predicateObjectMap.predicateMaps do
4          predicate ← new PredicateObjectMapGW();
5          predicate.ownTriplesMap ← predicateMap.ownTriplesMaps;
6          predicate.predicateURI ← predicateMap.constantValue;
7          predicate.parentTriple ← rom.parentTriplesMap;
8          if rom.getJoinConditions.size > 0 then
9              predicate.type is an enumerator which value is REFERENCE_MAP_JOIN;
10             forall element jc in rom.joinConditions do
11                 joinCondition ← new JoinConditionGW();
12                 joinCondition.child ← jc.child ;
13                 joinCondition.parent ← jc.parent ;
14                 Add joinCondition to the list of join conditions of predicate;
15             end
16         else
17             predicate.type is an enumerator which value is REFERENCE_MAP_SELF;
18         end
19         Add predicate to predicateList;
20     end
21 end
22 return predicateList;
```

Algorithm 4: Process Non-Referencing Object Maps

Input : RML PredicateObjectMap *pom*
Output: Collection of triple patterns *triplePatternsList*

```
1  Let predicateList be the collection of PredicateObjectMapGW;
2  forall element pm in pom.predicateMaps do
3      predicate ← new PredicateObjectMapGW();
4      predicate.predicateURI ← pm.constantValue;
5      predicate.ownTriplesMap ← pm.ownTriplesMaps;
6      forall element objectMap in pm.predicateObjectMap.objectMaps do
7          if objectMap.referenceMap is not null then
8              predicate.type is an enumerator which value is REFERENCE;
9              predicate.templateObject ← objectMap.referenceMap.reference;
10         else if objectMap.stringTemplate is not null then
11             predicate.type is an enumerator which value is TEMPLATE;
12             predicate.templateObject ← objectMap.stringTemplate;
13         Add predicate to predicateList;
14     end
15 end
16 return predicateList;
```

where tp_i is a *triple pattern* (Section 2.1.2), v_k denotes a variable and "." denotes the join operation between triple patterns. In a SPARQL join query, a variable that appears in multiple triple patterns implies a join between those triple patterns. To evaluate a SPARQL query, the query engine must find the variable bindings that satisfy the triple patterns, and return the bindings for the variables in the SELECT clause. In many RDF stores, triples are stored in a so-called *triple table*, which is a large table with the attributes *subject*, *predicate* and *object*, each line saving one triple. Thus, queries over RDF data involve a large number of inner-joins.

Example 1. Listing 4.1 shows query Q that consists of four triple patterns *tp1*, *tp2*, *tp3*, and *tp4*. *?v1* and *?v2* are the projected variables. *p1*, *p2*, *p3* and *p4* are constant predicates. *?o1* and *?o2* are variables that appear only once in *tp1* and *tp3* respectively, therefore they are not join variables. *?v1* appears in *tp1* and *tp2*, thus is a join variable. *?v2* appears in *tp2*, *tp3,* and *tp4*, so is also a join variable. In this example we have a total of 3 joins.

```
1 select ?v1 ?v2
2 where
3 {
4   ?v1 p1 ?o1    (tp1)
5   ?v1 p2 ?v2    (tp2)
6   ?v2 p3 ?o2    (tp3)
7   ?v2 p4 c3     (tp4)
8 }
```

Listing 4.1: Query Q.

Since in this thesis the focus is put on non-semantic structured data, where data is stored as records in tables instead of triples, query of Example 1 (see Listing 4.1) is executed differently. In the the RML mappings, there are Triple Maps that contain both properties *p1* and *p2*. This means that *tp1* and *tp2* can be found in the same table. In the final SQL query, instead of performing a join, we will only perform a projection (SELECT). This reduces considerably the query execution time each time a JOIN can be replaced with a projection.

In a triple pattern, positions *subject*, *predicate* and *object* can have constants or variables. Depending on the combination of them, we find eight patterns listed in Table 4.1. According to the work presented by Arias et al. [30], where the authors conducted an empirical study of real-world SPARQL queries, the most used pattern is *CCV* (i.e. given a subject and a predicate, obtain its values). *CVV* is also very common (i.e. given a

Pattern
VVV
VVC
VCV
VCC
CVV
CVC
CCV
CCC

TABLE 4.1: Triple patterns (V = Variable, C = Constant)

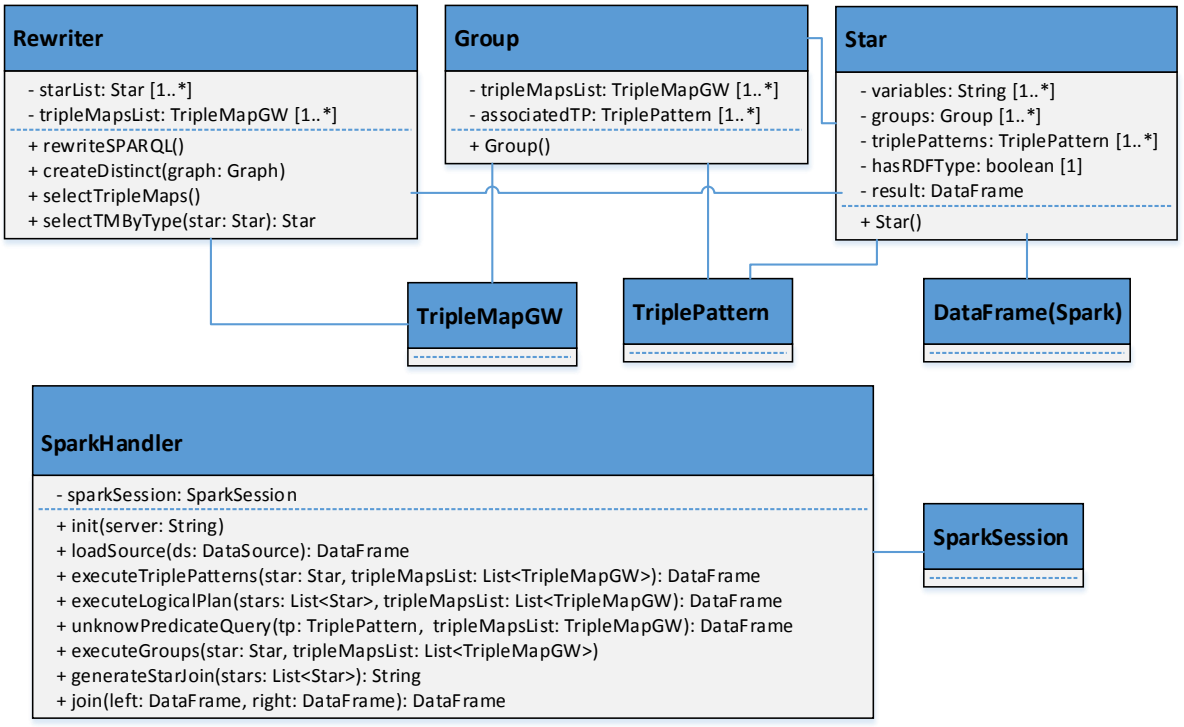

FIGURE 4.5: Class diagram of the SPARQL BGP Rewriter component.

subject, find all its predicates and values). The third most used pattern is *VCC* (i.e. given a predicate and its value, find all the subjects). It is very unlikely to find queries with unbound predicates, i.e., predicate position is a variable.

Figure 4.5 shows the class diagram used to build this component.

4.3.4.1 Logical Plan Construction

The process starts when method *rewriteSPARQL()* from class *Rewriter* is called, it receives a list of objects from class *TripleMapGW*, obtained using the RML parser component (Section 4.3.3).

In Section 2.1.2.1 we stated that *star-shaped* queries are the most common SPARQL queries. Considering this fact, we group the triple patterns by subject and use the method *createDistinct()* from class *Rewriter*. This method uses the object from class *Graph*, obtained using the SPARQL BGP Parser (Section 4.3.2). For each different subject, an object, named ***star***, from class *Star* is instantiated and all the related patterns are assigned to the attribute *triplePatterns*. The next step is to try to identify if there is any triple pattern within the group, that has predicate *rdf:type* and *object* URI, if this is the case, we look for the URI in the list of ***tripleMapsList***, if it is found, attribute *hasRDFType* of ***star*** is changed to *true*. Finally, ***star*** is added to ***stars***.

Afterwards, method *selectTripleMaps()* from class *Rewriter* is called, Algorithm 5 shows this latter. It takes a list of objects from class *Star*, ***stars*** and a list of objects from class *TripleMapGW* ***tripleMapsList***, as input and returns list ***stars*** updated with the groups for each star. The algorithm starts traversing ***stars***, each element is named ***star***. If attribute *hasRDFType* of ***star*** is true, i.e., class is known, then attribute *groups* is assigned to the result of calling method *selectTMByType()* from class *Rewriter* (lines 4-5).

In case the class is unknown, we traverse the triple pattern list of ***star***, each element of this list is named ***tp***. If the triple pattern's predicate is *rdf:type* and its object is a variable, or the predicate is a variable, we create an object from class *Group* and add it to the list of groups of ***star***; then we continue with the next element of the list (lines 8-12).

In case the predicate is constant, i.e., it is an URI, we search it in the *predicateList* of all the elements from ***tripleMapsList***. Each element of ***tripleMapsList*** is named ***tm*** and each element of *predicateList* from **tm** is named ***pom***. If attribute *predicateURI* is the same as ***tp***'s predicate then an object, named ***pomFound***, from class *PredicateObjectMapGW* is created, a copy of ***tm*** is created and assigned to object ***tpFound***, ***pomFound*** is added to ***tpFound***. If the map ***tmMap*** does not contain ***tpFound***, it is added to the map, otherwise ***pomFound*** is added (lines 14-28).

We use an array of *TripleMapGW* to order the map ***tmMap*** in descending order, considering as ordering criteria, the size of its *predicateList* (lines 32-33). If the group list (***groupList***) is empty, the triple map that has the majority or all predicates for the respective star, i.e., the one with biggest *predicateList* size, is added to the list of triple patterns from the object, named ***group***, from class *Group*; and this latter is added to ***groupList*** (lines 35-38). In case ***groupList*** is not empty, function *RemoveTripleMaps()* is called, the output of this is assigned to ***groupList***. Next step, is assign ***groupList*** to the *groups* of ***star***. The algorithm returns the updated list ***stars***.

Algorithm 5: Select Triple Maps

Input : List of objects from class Star *stars* and a list of objects from class TripleMapGW *tripleMapsList*

Output: Updated *stars*

```
1  Let tmMap be a Map of TripleMapGW;
2  Let tmArray be an Array of TripleMapGW;
3  forall element star in stars do
4      if star.hasRDFType then
5          star.groups = SelectTMByType(star);
6      else
7          forall element tp in star.triplePatterns do
8              if tp (predicate rdf:type and object is variable) or predicate is variable then
9                  group ← new Group;
10                 Add tp to group.associatedTP;
11                 Add group to star.groups;
12                 continue;
13             end
14             forall element tm in tripleMapsList do
15                 forall element pom in tm.predicateList do
16                     if pom.predicateURI == tp's predicate then
17                         pomFound ← new PredicateObjectMapGW(pom);
18                         tpFound ← new TripleMapGW(tm);
19                         Add pomFound to tpFound.predicateList;
20                         if tpFound ∉ tmMap then
21                             Add tpFound to tmMap
22                         else
23                             tpFound ← tmMap.get(tpFound);
24                             Add pomFound to tpFound
25                         end
26                         break;
27                     end
28                 end
29             end
30         end
31         tmArray ← tmMap;
32         Sort tmArray in descending order considering the size of its predicateList;
33         Let groupList be a list of Group;
34         forall element tm in tmArray do
35             if groupList is empty then
36                 group ← new Group;
37                 Add tm to group.triplePatterns;
38                 Add group to groupList;
39             else
40                 group ← RemoveTripleMaps(groupList, tm);
41                 Add group to groupList if it is not null;
42             end
43         end
44         Assign groupList to star.groups;
45     end
46 end
47 return stars;
```

Algorithm 6: Select Triple Map By Type

Input : Object from class Subject *star*

Output: List of objects from class Group *groupList*

```
1  Let tm, newTm be objects from class TripleMapGW;
2  Let tpDefinesClass an object from class TriplePattern;
3  Let groupProperties an object from class Group;
4  forall element tripleMap in tripleMapsList do
5      if tripleMap.subjectMap contains tpDefinesClass.o then
6          tm ← tripleMap;
7          break;
8      end
9  end
10 Remove tpDefinesClass from subject.triplePatterns;
11 forall element tp in subject.triplePatterns do
12     if tp.p is variable then
13         group ← new Group();
14         Add tp to group.associatedTP;
15         Add group to groupList;
16         Remove tp from subject.triplePatterns;
17     end
18     forall element pom in tm.predicateList do
19         if tp.p == pom.predicateURI then
20             newTm ← new TripleMapGW();
21             pom.triplePattern ← tp;
22             Add pom to newTm.predicateList;
23             Remove tp from subject.triplePatterns;
24             Add tp to groupProperties.associatedTP;
25             break;
26         end
27     end
28 end
29 Add newTm to groupProperties.tripleMapList;
30 Add groupProperties to groupList;
31 if star.triplePatterns.size > 0 then
32     forall element tp in subject.triplePatterns do
33         forall element tripleMap in tripleMapsList do
34             forall element pom in tm.predicateList do
35                 if tp.p == pom.predicateURI then
36                     group ← new Group();
37                     newTm ← new TripleMapGW();
38                     pom.triplePattern ← tp;
39                     Add pom to newTm.predicateList;
40                     Add tp to group.associatedTP;
41                     Add newTm to group.tripleMapsList;
42                     Add group to groupList;
43                 end
44             end
45         end
46     end
47 end
48 return groupList;
```

Line 5 from Algorithm 5 calls the method *selectTMByType()* from class *Rewriter*. Algorithm 6 shows this method. It takes as input an object, ***star***, from class *Star* and returns a list of objects from class *Group*, ***groupList***. The first step is to find the object from class *TripleMapGW*, ***tm***, which subject map contains the URI class that defines ***star*** (lines 4-9).

The next step is to remove the triple pattern that defines ***star*** from ***star.triplePatterns***, then, traverse this list. If the triple pattern's predicate is a variable, we create an object, ***group***, from class *Group*, add the current triple pattern to the *associatedTP* list of ***group***, then add this latter to ***groupList*** and remove the current triple pattern from ***star.triplePatterns*** (lines 12-17) .

If the current triple pattern's predicate is an URI, we loop the list of predicate object maps of ***tm***. In case of finding it, we create a new *TripleMapGW*, ***newTm***, assign the current triple pattern ***tp*** to the triplePattern attribute of the current predicate object map, ***pom***; then remove ***tp*** from ***star.triplePatterns***; add ***tp*** to ***groupProperties*** (lines 19-26). This latter is declared in line 3. Next, add ***newTm*** to ***groupProperties.tripleMapList***, the latter is added to ***groupList*** (lines 29-30).

Finally, if ***star.triplePatterns*** still has elements, the remaining triple patterns are searched in ***tripleMapList***; the same data structure is build as in the previous step (lines 31-47).

Algorithm 7 shows function *RemoveTripleMaps()* called in line 40 of Algorithm 5. It takes as input a list of objects from class Group, ***groupList*** and an object from class *TripleMapGW* ***newTm***; it returns an object from class Group , ***newGroup***; or null. In line 1 variable ***newGroup***, from class *Group*, is created and ***newTm*** is added to the list of triple maps of ***newGroup***; variable ***i*** is initalized to 0 (lines 2 -3). Next step, is traverse the list ***groupList***, for further reference in the algorithm each element of this list is called ***group*** (lines 4, 22). Variable ***tm***, which is an object from class *TripleMapGW*, gets the value of the first element is the list of TripleMapGW of ***group*** (line 5). If *predicateList* of ***tm*** does not have any element in common with the corresponding *predicateList* of ***newTm*** and variable ***i*** is the penultimate element of ***groupList*** then ***newGroup*** is returned (lines 6-8). Otherwise, if *predicateList* of ***tm*** does not have any element in common with the corresponding *predicateList* of ***newTm***, then the loop continues with the next element (lines 9-11). Otherwise, if *predicateList* of ***tm*** contains same elements as *predicateList* of ***newTm*** and the ***tm's*** name is the same as ***newTm's*** name, then ***newGroup*** is returned (lines 12-14). Otherwise, if *predicateList* of ***tm*** contains same elements as *predicateList* of ***newTm*** and the ***tm's*** name is not the same as ***newTm's*** name, then ***newTm*** is added to the *tripleMapsList* of ***group*** (lines 15-17). Otherwise, if *predicateList* of ***newTm*** is a subset of predicateList of ***tm*** then null is returned (lines

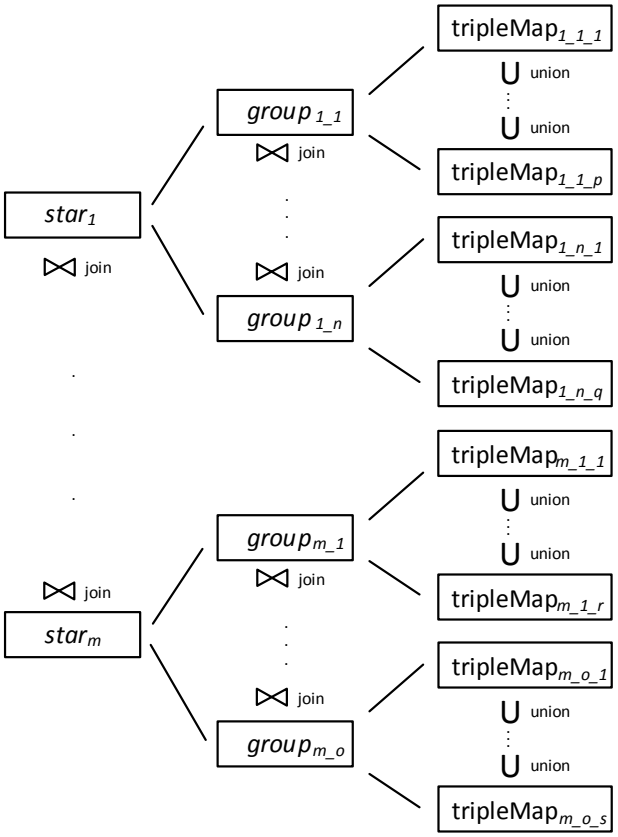

FIGURE 4.6: Logical Plan.

18-20). If none of the conditions are fulfilled, then variable $\boldsymbol{i}$ is increased by one and the loop continues (line 21). If the loop has finished the algorithm returns null (line 23).

Figure 4.6 shows the logical plan that will be executed by Spark SQL. A *triple map* can have one or more *predicate object maps*, and this latter contains the *triple patterns* to be executed. A *group* can have one or more *triple maps*, if it has more than one, a union is performed with the result of each *triple map*. A *star* can have one or more *groups*, a join is performed with the result of each *group*. Finally, a join is performed with the result of each *star* and a projection operation is done over the result of this.

4.3.4.2 Spark SQL Queries

The SQL queries generated starting from SPARQL queries are executed on Spark using its Spark SQL API. The basic data structure of Spark SQL is the so-called DataFrames, which are collections of data organized in columns and distributed over a cluster's machines. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimization's under the hood (mainly for distributed parallel query execution). DataFrames can be constructed from a wide array of sources such as: (semi-)structured data files (e.g., CSV, JSON, Parquet), tables in Hive, and

Algorithm 7: Remove Triple Maps

Input : List of objects from class Group *groupList*, object *newTm* from class TripleMapGW

Output: Object from class Group *group*

```
1  newGroup ← new Group();
2  Add newTm to newGroup.tripleMapsList;
3  i = 0;
4  forall element group in groupList do
5      tm ← get first element of group.tripleMapsList;
6      if tm.predicateList ∩ newTm.predicateList = ∅ and i = groupList.size - 1 then
7          return newGroup;
8      end
9      else if tm.predicateList ∩ newTm.predicateList = ∅ then
10         continue;
11     end
12     else if tm.predicateList = newTm.predicateList and tm.name = newTm.name then
13         return newGroup;
14     end
15     else if newTm.predicateList = tm.predicateList and tm.name ≠ newTm.name then
16         Add newTm to group.tripleMapsList;
17     end
18     else if newTm.predicateList ⊂ tm.predicateList then
19         return null;
20     end
21     i += 1;
22 end
23 return null;
```

many databases [6]. We exploit this very characteristic to build our *Multi-Data* Wrapper. All the supported data types and databases just mentioned, are loaded into a Spark DataFrame. DataFrames are data-type-agnostic, meaning that once created they are processed the same way, regardless of from where they were constructed from. In other words, DataFrames abstracts away the differences of the input data. We only need to indicate upon the loading what is the type of data to be loaded, so Spark uses the corresponding API. The type of the data is set using RML ReferenceFormulation (Section 2.2.2). RML by default supports a limited number of reference formulations (type) like CSV and database. We extended this RML feature to support Parquet files and the same could be done to cover data sources Spark supports, like CQL of Cassandra databases.

Once the DataFrame has been built, it is queried using the generated SQL queries. Scala object *SparkHandler* is in charge of generating the queries and executing them. The first step is to call the *init()* method, which initializes the *SparkSession* with the server URL specified in its argument. Next, the sources mapped in the RML file are loaded using

[6] Spark SQL, DataFrames and Datasets Guide

the method *loadSource()*. This method reads the attribute *logicalSource* that contains the enumerator that tells the algorithm the type of source of the triple map.

Algorithm 8: Execute Logical Plan

Input : List of objects from class Star *starList*, List of objects from class TripleMapGW *tripleMapsList*

Output: DataFrame with the result of the whole execution *result*

```
1  Let sql be a string;
2  Let sparkSession be an initialized SparkSession;
3  sql ← GenerateStarJoin(starList);
4  if starList.size = 1 then
5  |  star ← get the first element of starList;
6  |  return ExecuteGroups(star.groups, star, tripleMapsList);
7  else
8  |  i = 1;
9  |  forall element star in starList do
10 |  |  star.result ← ExecuteGroups(star.groups, star, tripleMapsList);
11 |  |  Create temporary table for star.result with name "final_" + i;
12 |  |  i += 1;
13 |  end
14 |  return sparkSession.sql(sql);
15 end
```

Algorithm 8 shows method *executeLogicalPlan()* from class *SparkHandler*, which is called to execute the logical plan. It receives as arguments the ***starList***, previosuly built (Section 4.3.4.1); and a list of objects from class *TripleMapGW*. It returns a *Spark DataFrame* that contains the execution of the whole process.

In line 3, variable ***sql*** receives the result of calling the method *generateStarJoin()*. Then, if ***starList*** has only one element, this latter is taken from the list and assigned to variable ***star***, the result of calling method *executeGroups()* is returned (lines 4 - 6). Otherwise, variable ***i*** is initialized to 1, then, ***starList*** is traverse, each element of the list is assigned to variable ***star*** (lines 9, 13). The attribute *result* of ***star*** is assigned with the result of calling method *executeGroups()*, next, a temporary table is created with the *DataFrame* of the ***star***, it is named with the prefix "final_" plus the value of the counter ***i***; this latter is increased by 1 (lines 10-12). Finally, the query statement ***sql*** is executed with the ***sparkSession*** and returned by the algorithm (line 14).

Method *generateStarJoin()* from class *SparkHandler* is in charge of generating the query that joins the variables that exist in the triple patterns list of each object from class *Star*. For each star, attribute *variables* is filled in with all the variables from the grouped triple patterns that belongs to it. For URIs on subject or object positions, the attribute *genericVar* is used, this is useful in case the same URI is present in other stars. The algorithm uses a *Map<k,v>*, named ***mapVar***, *k* being a String representing the key,

and v being a binary array of int, representing the respective value. The name of each variable of each star is put it into the Map as key ***k*** when it is not already there and its value ***v*** is initialized with the size of ***starList***. Each position on the array represent one star of the whole query. If the variable exists on the star, the respective position receives the value of 1, otherwise 0. Next, we iterate ***mapVar***. if the sum of the respective array is 1, it means that the variable appears only in one star, otherwise the variable appears in two o more stars, i.e., a join exists between the stars that have this variable. To build the final query is necessary to state that each star represents a table which name is *final_ i*, where i is the position of the star in ***starList***. The select clause is built using the table's name of each star plus the name of the variable, from clause is built using the table's name of each star. and the where clause is built using the joins aforementioned.

Method *executeGroups()* from class *SparkHandler* is shown in Algorithm 9. It takes as input an object from class Star, ***star***, and the list of all *triple maps* from the RML file, ***tripleMapsList***. It returns Spark DataFrame ***result***. In this algorithm variables ***group1***, ***left***, ***right***, ***groupLeft***, ***groupRight*** are used (lines 1-2).

If the list of groups from ***star*** has only one element and the triple maps list size of this latter is zero, then the result of calling the method *unknowPredicateQuery()*is returned (lines 4-6).

If the list of groups from ***star*** has only one element and the triple maps list size of this latter is one, then the result of calling the method *executeTriplePatterns()* is returned (lines 7-9). Otherwise, the list of groups from size is traversed (lines 11, 33). Variable ***i*** is initialized to zero (line 10). In case the value of ***i+1*** is equals to the size of the group list from ***star***, then the algorithm breaks the while clause (lines 12-14).

In the case that the value of ***i*** is greater than zero, ***result*** is assigned to ***left*** (lines 15-17). Otherwise, ***groupLeft*** gets the value of the element in position ***i*** of the list of groups from ***star*** (line 18); if the size of the triple maps's list from ***groupLeft*** is zero, then ***left*** gets the result of calling method *unknowPredicateQuery()*, otherwise the result of calling method *executeTriplePatterns()* (lines 19-23).

Variable ***groupRight*** gets the value of the element in position ***i+1*** of the list of groups from ***star*** (line 25); if the size of the triple maps's list from ***groupRight*** is zero, then ***right*** gets the result of calling method *unknowPredicateQuery()*, otherwise the result of calling method *executeTriplePatterns()* (lines 26-30).

Variable ***result*** gets the outcome of calling method *join()*, variable i increases its value by one. At the end of the algorithm ***result*** is returned (line 35).

Algorithm 9: Execute Groups

Input : Object from class Star *star*, List of objects from class TripleMapGW *tripleMapsList*

Output: Spark DataFrame *result*

```
1  Let group1 be an object from class Group;
2  Let left, right be Spark DataFrames Let groupLeft, groupRight be objects from class
     Group;
3  group1 ← Get the first element of star.groups;
4  if star.groups.size = 1 and group1.triplesMapList.size = 0 then
5  |  return UnknownPredicateQuery(group1, tripleMapsList);
6  end
7  if star.groups.size = 1 and group1.triplesMapList.size > 0 then
8  |  return ExecuteTriplePatterns(group1.triplesMapList, star);
9  else
10 |  i ← 0;
11 |  while i < star.groups.size do
12 |  |  if (i + 1) = star.groups.size then
13 |  |  |  break;
14 |  |  end
15 |  |  if i > 0 then
16 |  |  |  left ← result;
17 |  |  else
18 |  |  |  groupLeft ← star.groups [i ];
19 |  |  |  if groupLeft.triplesMapList.size = 0 then
20 |  |  |  |  left ← UnknownPredicateQuery(groupLeft, tripleMapsList);
21 |  |  |  else
22 |  |  |  |  left ← ExecuteTriplePatterns(groupLeft.triplesMapList, star);
23 |  |  |  end
24 |  |  end
25 |  |  groupRight ← star.groups [i +1];
26 |  |  if groupRight.triplesMapList.size = 0 then
27 |  |  |  right ← UnknownPredicateQuery(groupRight, tripleMapsList);
28 |  |  else
29 |  |  |  right ← ExecuteTriplePatterns(groupRight.triplesMapList, star);
30 |  |  end
31 |  |  result ← Join(left, right);
32 |  |  i + = 1;
33 |  end
34 end
35 return result;
```

Method *unknownPredicateQuery()* from class *SparkHandler* is called when the triple pattern's predicate is unknown, i.e., it is a variable. It receives as input an object, ***group***, from class *Group* and the list of triple maps from the RML file, ***tripleMapsList***. A Spark DataFrame ***result*** is returned. The ***tripleMapsList*** is traversed and a new Spark DataFrame is created for each SubjectMapGW ***subjectMap*** and each element of the list *predicateList*. The contents of this DataFrame are added to ***result*** using Spark

transformation *UNION*. The DataFrame has 3 columns named using the respective name of the variable when the position is a variable or genericVar when subject or object are URI. Using the Spark transformation *map* we can define how to construct the subject, predicate, and object of the triple.

From Table 4.1 is known that when the predicate is a variable, there are four cases: *VVV*, *VVC*, *CVV*, and *CVC*. These cases are used to build the query accordingly. For the *SubjectMapGW* the query to execute is formed according of the form of the triple pattern as follows:
VVV: Select clause is formed by the fields from the attribute *template*. Where clause is formed by the fields of the subject's template which values must be not *null*. The set *classURI* is traversed, a triple is created for each element on it.
VVC: It is necessary to check whether or not the URI in the object exists in the set *classURI*. If it exists, then select clause is the same as before. In the where clause all the subject template's fields must be not *null*.
CVV: Select clause is formed by the fields from the attribute *template*. For this case we extract the values for the subject's fields and put them in the where clause.
CVC:It is necessary to check whether or not the URI in the object exists in the set *classURI*. If it exists, then select clause is the same as before. Where clause is the same as *CVV*.

For each PredicateObjectMapGW, we have the same four cases, the difference is that the predicate is taken from attribute *predicateURI*. Select include subject fields and object fields. Where clause include subject conditions and the object condition is built considering the following criteria:

1. **Reference:** Attribute *templateObject* contains the name of the field. If the object is a constant, the condition that goes in the where clause is *field* = *value*, considering on the data type of the field. Otherwise, the condition is *field is not null*.

2. **Template:** Attribute *templateObject* contains the template to build the object. if the object is a constant, we extract the values from the template, form the condition $field_1 = value_1$ .. *and* $field_i = value_i$ and add it to the where clause. Otherwise, the conditions is $field_1$ *is not null ... and* $field_i$ *is not null*.

3. **Join Condition:** When the size of attribute *joinCondition* is greater than 0, it means there is a join condition. We add this condition to form the join

$$child\_table.child\_field_1 = parent\_table.parent\_field_1$$
$$and... \ and$$
$$child\_table.child\_field_i = parent\_table.parent\_field_i$$

And add it to the where clause. The object is formed using the template of attribute *parentTriple* and following the same criteria as for subject.

4. **No Join Condition:** When the size of attribute *joinCondition* is 0, it means there is no join condition, the object is formed using the template of attribute *parentTriple* and following the same criteria as for subject.

Method *executeTriplePatterns()* from class *SparkHandler* is in charge of obtaining the queries when the predicate is known, i.e., it is a constant. It takes as input a list of objects from class *TripleMapGW* ***tripleMapsList***, and the star they belong to. The method returns a Spark DataFrame ***result***. If the size of ***tripleMapsList*** is 1, it means that the predicate or set of predicates are found only in that triple map. In case the size is bigger than 1, it means that the predicate or set of predicates are found in all the triple maps that are part of the list. In the second case a Spark transformation *UNION* is required between the results of all the triple maps. This method does the same as *unknownPredicateQuery*, with the difference that select and where clauses include the set of Predicate Object Maps.

Method *join()* from class *SparkHandler* is in charge of performing the join between two Spark DataFrames, ***left*** and ***right***. If a variable exists in both dataframes, then a join condition is made using them.

Finally, after method *executeLogicalPlan()* (Algorithm 8) returns the DataFrame, method *Projection()* from class *SparkHandler* is called. This latter is in charge of selecting the fields that are part of the select clause of the BGP.

Chapter 5

Experiments

In this chapter, we report on the performances of our Multi-Data Wrappers. We evaluated two metrics:

- *Query Execution Time*: we measure the time needed to extract the relevant data from the wrapped data source.
- *Scalability*: we track query execution time over increasing size scales of data. As part of the scalability evaluation, we compare our *Static* Data Wrapper against the data extraction component of RML, RML Processor, for the time needed to exhaustively generate RDF data from the whole wrapped data source.

Even though our Data Wrappers are able to extract data from several sources, we evaluate their performances only over CSV data. This is because CSV is a raw format that does not benefit from any compressing or indexing other data sources do, like Parquet files, Hive tables, or HBase databases. Meaning that we are evaluating the worst-case scenario; any other advanced data sources would deliver better performances.

All experiments where conducted in a small-sized cluster of three machines each having:

- DELL PowerEdge R81
- 2x AMD Opteron 6376 (16 Core) CPU
- 256 GB RAM
- 3 TB SATA RAID-5 disk.

	Scale Factor					
	284826		**1200000**		**4800000**	
Source	**Size(GB)**	**#Records**	**Size(GB)**	**#Records**	**Size(GB)**	**#Records**
ProductFeature	0.0183	47884	0.0361	94259	0.0751	195779
ProductType	0.000797	2011	0.0015	3949	0.0032	8191
Producer	0.0023	5618	0.0099	23737	0.0397	94744
Product	0.4067	284826	1.72	1200000	6.94	4800000
ProductTypeProduct	0.0139	1424130	0.0618	6000000	0.2648	24000000
ProductFeatureProduct	0.0636	5533832	0.2786	23142573	1.2	92707965
Vendor	0.0012	2896	0.005	12017	0.02	47982
Offer	1.02	5696520	4.39	24000000	17.92	96000000
Person	0.0113	146093	0.0482	614976	0.1963	2460292
Review	3.97	2848260	16.74	12000000	67.37	47999800
TOTAL	5.508097	15992070	23.2911	67091511	94.0291	268314753

TABLE 5.1: CSV files generated using Berlin Benchmark

As mentioned throughout the thesis, we used Apache Spark version 2.1.0 [1] to implement our Data Wrappers, and the Distributed File System of Apache Hadoop 2.7.2[2] (HDFS) to store the different CSV.

5.1 Description of the Dataset

The synthetic dataset used to test our approach was generated using the Berlin SPARQL Benchmark (BSBM)[3] [31]. BSBM provides a data generator that generates data around an e-commerce use-case as follows: "*products offered* by *vendors* and purchased by *consumers* who have given their *reviews* about the purchased products". The generator can produce data in RDF and relational models. We have generated data in the latter model (as SQL dumps) [4] and exported it under CSV files. For this experiment we have used three scale factors (number of products): 284826, 1200000 and 4800000. If we semantify (convert into RDF) the first dataset, it produces 100m distinct triples. If we semantify the second dataset, it produces around 420m distinct triples. If we semantify the third dataset, it produces 1.7 billion distinct triples. Table 5.1 shows the characteristics of each of the CSV files generated starting from this benchmark.

5.1.1 RML Mapping

As described earlier (Chapter 2.2.2), our Data Wrappers require as input an RML Mapping (described in Chapter 3) to be associated with the data. In order to build the corresponding RML Mapping of the generated BSBM test data, it was necessary to look

[1] http://spark.apache.org/
[2] http://hadoop.apache.org/
[3] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/index.html
[4] Using the command `generate -fc -pc #scalefactor -s sql`

at the data Entity Relationship Diagram (ERD) [32]. We created the ERD (see Figure 5.1) considering BSBM data specifications as follows:

1. Each entity is represented by a Triple Mapping. The subject URI is defined according to the Data Specification.
2. All the attributes that are not used to establish a relationship between two entities are represented by either *rml:reference* or *rr:template*.
3. The relationship between two entities is represented by *rr:parentTriplesMap* and its join condition by *rr:joinCondition*.
4. According to the Data Specification there are self references within the properties. They have been represented by *rr:parentTriplesMap* without a join condition.

Listing 5.1 shows a snippet of the Offer Triple Map. The Subject Map uses *rr:template* to define its URI of type *bsbm:Offer*. One of the Predicate Object Maps of this mapping is the relationship between Offer and Product, represented by *rr:parentTriplesMap*. Its join condition on product (from Offer) and on *"nr"* (from Product) has been represented by *rr:joinCondition*. Finally, the attribute *price* has been defined using *rml:reference*.

```
<#OfferMapping>
...
 rr:subjectMap[
  rr:template "http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromVendor{vendor}/Offer{nr}";
  rr:class bsbm:Offer
 ];
 rr:predicateObjectMap[
  rr:predicate bsbm:product;
  rr:objectMap[
   rr:parentTriplesMap <#ProductMapping>;
   rr:joinCondition[
    rr:child "product";
    rr:parent "nr"
   ]
  ]
 ];
 rr:predicateObjectMap[
  rr:predicate bsbm:price;
  rr:objectMap[
   rml:reference "price";
   rr:datatype bsbm:USD
  ]
 ];
...
```

Listing 5.1: Snippet of the RML mapping for Offer

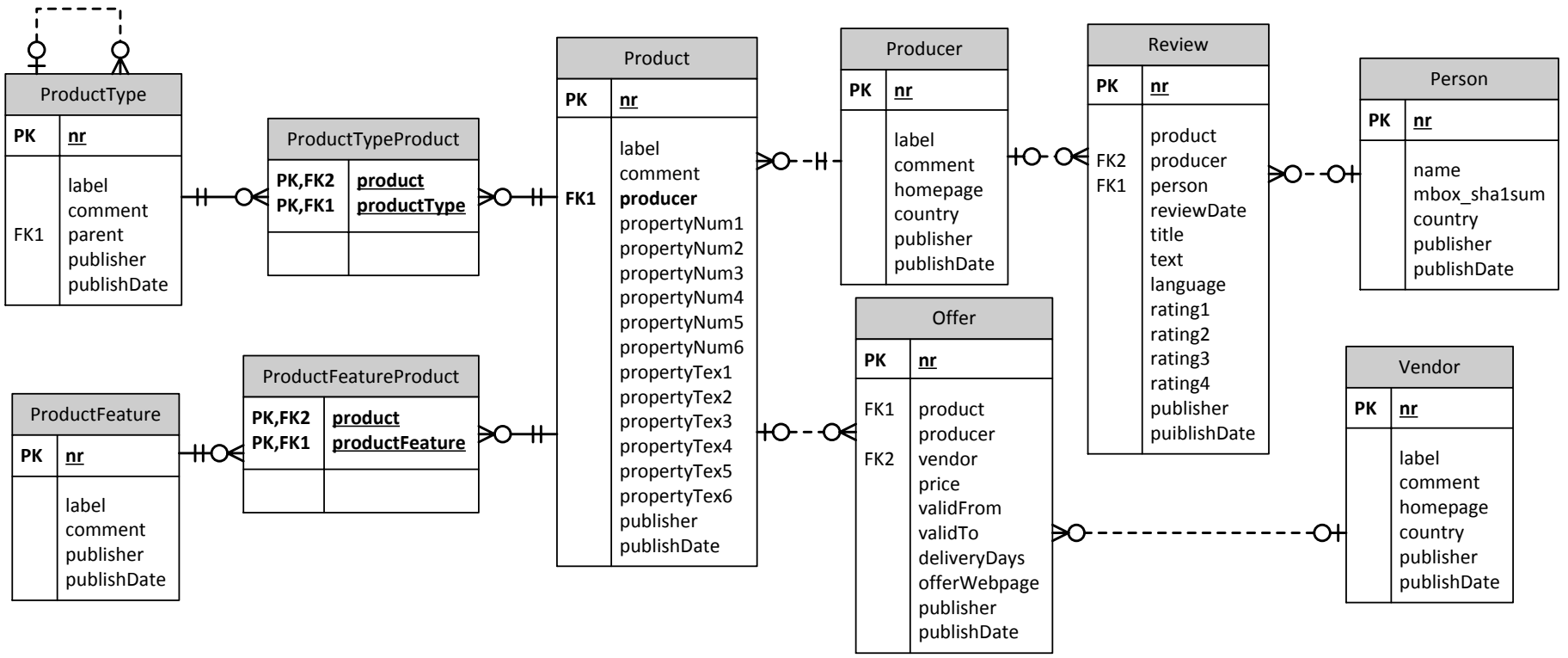

FIGURE 5.1: Berlin Benchmark Entity Relationship Diagram

BGP	**q1**	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**	**q8**	**q9**
#Triple Patterns	7	13	2	12	6	1	6	6	2
#Projected Variables	5	10	1	10	5	2	6	6	3
#Variables	5	12	1	10	6	2	6	6	3
#Shared vars	1	2	1	2	2	0	2	2	1
#TPs with 0 const	0	0	0	0	0	0	0	0	0
#TPs with 1 const	4	2	0	8	5	1	5	5	2
#TPs with 2 const	3	11	2	4	1	0	1	1	0
#TPs with 3 const	0	0	0	0	0	0	0	0	0

TABLE 5.2: BGP characteristics for Berlin Benchmark dataset

5.2 Description of Query Workload

Since our current work does not support OPTIONAL, FILTER, nor UNION, 3 of the 12 BSBM queries were discarded. Table 5.2 gives a description of the BGPs contained in each of the 9 queries. The BGPs are comprised of different numbers of triple patterns, variables and constants.

BGP **q1**, shown in Listing 5.2, contains 7 triple patterns, producing 1 star-shaped pattern on variable *product*. Triple patterns tp1, tp2, tp6, tp7 are grouped together and executed by triple map *ProductMapping* since it contains their respective predicates. Triple map *ProductTypeProductMapping* executes triple tp3 since it contains its respective predicate. Triple map *PoductFeatureProductMapping* executes triple tp4 and tp5, but separately, since it contains its respective predicate.

```
1  select
2   ?product ?label ?comment ?value1 ?value2
3  where {
4   ?product rdfs:label ?label .                                    (tp1)
5   ?product rdfs:comment ?comment .                                (tp2)
6   ?product a bsbm-inst:ProductType1 .                             (tp3)
7   ?product bsbm:productFeature bsbm-inst:ProductFeature595 .      (tp4)
8   ?product bsbm:productFeature bsbm-inst:ProductFeature601 .      (tp5)
9   ?product bsbm:productPropertyNumeric1 ?value1 .                 (tp6)
10  ?product bsbm:productPropertyNumeric2 ?value2 .                 (tp7)
11 }
```

LISTING 5.2: BGP q1

BGP **q2**, shown on Listing 5.3), contains 13 triple patterns, producing 3 star-shaped patterns on URI *bsbm-inst:Product1* and variables *p* and *f*. Triple patterns tp1, tp2, tp3, tp5, tp9, tp10, tp11, tp12, tp13 are grouped together and executed by triple map *ProductMapping* since it contains their respective predicates. Triple pattern tp7's predicate is *rdf:type*, when we have this predicate, the subject map is also evaluated, since the algorithm determines that subject *bsbm-inst:Product1* is found in triple maps *ProductMapping* and *ProductTypePorductMapping*. The triple pattern is executed in the subject map of *ProductMapping*. On the other hand *ProductTypeProductMapping*'s subject map does not contain a class in the set of classes, but it is found in the predicate object map, thus, it is executed using it as well. Triple pattern tp6 is executed by triple map *ProductFeatureProductMapping* since it contains its respective predicate.
There are several triple maps that contain predicate *rdfs:label*, nevertheless the algorithm knows that variable *p*, from triple pattern tp4, is referred only to triple map ProducerMapping, therefore it is used for the execution of this triple pattern.
The same behavior applies to variable *f*, from triple pattern tp8. The algorithm only uses triple map *ProductFeatureMapping*.

```
1  select
2   ?label ?comment ?producer ?productFeature ?propertyTextual1 ?propertyTextual2
3   ?propertyTextual3 ?propertyNumeric1 ?propertyNumeric2 ?productType
4  where {
5   bsbm-inst:Product1 rdfs:label ?label .                                    (tp1)
6   bsbm-inst:Product1 rdfs:comment ?comment .                                (tp2)
7   bsbm-inst:Product1 bsbm:producer ?p .                                     (tp3))
8   ?p rdfs:label ?producer .                                                 (tp4)
9   bsbm-inst:Product1 dc:publisher ?p     .                                  (tp5)
10  bsbm-inst:Product1 bsbm:productFeature ?f .                               (tp6)
11  bsbm-inst:Product1 a ?productType .                                       (tp7)
12  ?f rdfs:label ?productFeature .                                           (tp8)
13  bsbm-inst:Product1 bsbm:productPropertyTextual1 ?propertyTextual1 .       (tp9)
14  bsbm-inst:Product1 bsbm:productPropertyTextual2 ?propertyTextual2 .       (tp10)
15  bsbm-inst:Product1 bsbm:productPropertyTextual3 ?propertyTextual3 .       (tp11)
16  bsbm-inst:Product1 bsbm:productPropertyNumeric1 ?propertyNumeric1 .       (tp12)
17  bsbm-inst:Product1 bsbm:productPropertyNumeric2 ?propertyNumeric2 .       (tp13)
18 }
```

Listing 5.3: BGP q2

BGP **q3**, shown in Listing 5.4, contains 2 triple patterns that form 1 star-shaped query on variable *product*. In this query the class of variable *product* is explicitly specified in triple pattern tp2, therefore the algorithm knows that it is referring to triple map ProductMapping and only uses this latter for the execution of triple pattern tp1.

```
1 select
2  ?product
3 where {
4  ?product rdfs:label "antinomianism" .          (tp1)
5  ?product rdf:type bsbm:Product .               (tp2)
6 }
```

Listing 5.4: BGP q3

BGP **q4**, shown in Listing 5.5, contains 12 triple patterns forming 4 star-shaped patterns on URI *bsbm-inst:Product10* and variables *offer*, *vendor*, *review* and *reviewer*. Triple pattern tp1 is executed using triple map *ProductMapping* since the algorithm determines that tp1's subject belongs to *ProductMapping* Triple patterns tp2, tp3, tp4, tp7 and tp8 are executed using triple map *OfferMapping*, since it contains their respective predicates. Triple patterns tp5 and tp6 are executed using triple map *VendorMapping*, since it contains their respective predicates. Triple patterns tp9, tp10 and tp12 are executed using triple map *ReviewMapping*, since it contains their respective predicates. Triple pattern tp11 is executed using triple map *ReviewerMapping*, since it contains its respective predicate. Triple pattern tp1 is executed using triple map *ProductMapping*, since it contains its respective predicate and the algorithm identifies that the subject belongs to it.

```
1  select
2   ?productLabel ?offer ?price ?vendor ?vendorTitle
3   ?review ?revTitle ?reviewer ?revName ?validDate
4  where {
5   bsbm-inst:Product10 rdfs:label ?productLabel .                          (tp1)
6   ?offer bsbm:product bsbm-inst:Product10 .                               (tp2)
7   ?offer bsbm:price ?price .                                              (tp3)
8   ?offer bsbm:vendor ?vendor .                                            (tp4)
9   ?vendor rdfs:label ?vendorTitle .                                       (tp5)
10  ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#GB> . (tp6)
11  ?offer dc:publisher ?vendor .                                           (tp7)
12  ?offer bsbm:validTo ?validDate .                                        (tp8)
13  ?review bsbm:reviewFor bsbm-inst:Product10 .                            (tp9)
14  ?review rev:reviewer ?reviewer .                                        (tp10)
```

```
15  ?reviewer foaf:name ?revName .                                  (tp11)
16  ?review dc:title ?revTitle .                                    (tp12)
17  }
```

Listing 5.5: BGP q4

BGP **q5**, shown in Listing 5.6, contains 6 triple patterns forming 2 star-shaped pattern on variables *review* and *reviewer*. Triple patterns tp1, tp2, tp3, tp4, tp5 are executed using triple map *ReviewMapping* since it contains their respective predicates.
Triple pattern p6 is executed using triple map *ReviewerMapping*, since it contains its respective predicate.

```
1  select
2  ?title ?text ?reviewDate ?reviewer ?reviewerName
3  where {
4  ?review bsbm:reviewFor bsbm-inst:Product555 .          (tp1)
5  ?review dc:title ?title .                              (tp2)
6  ?review rev:text ?text .                               (tp3)
7  ?review bsbm:reviewDate ?reviewDate .                  (tp4)
8  ?review rev:reviewer ?reviewer .                       (tp5)
9  ?reviewer foaf:name ?reviewerName .                    (tp6)
10 }
```

Listing 5.6: BGP q5

BGP **q6**, shown in Listing 5.7, contains only one triple pattern with unbound predicate. The algorithm identifies that the subject belongs to triple map ReviewMapping, therefore all its predicates are considered for the execution of tp1.

```
1  select
2  ?predicate ?valueObject
3  where {
4  bsbm-inst:Review7 ?predicate ?valueObject.             (tp1)
5  }
```

Listing 5.7: BGP q6

BGP **q7**, shown in Listing 5.8, contains 6 triple patterns forming 2 star-shaped patterns on variables *offer* and *vendor*. Triple patterns tp1, tp2, tp3 and tp6 are executed using triple map *OfferMapping*, since it contains their respective predicates.

Triple patterns tp4 and tp5 are executed using triple map *VendorMapping*, since it contains their respective predicates.

```
1  select
2   *
3  where {
4   ?offer bsbm:product ?product.                                          (tp1)
5   ?offer bsbm:price ?price .                                             (tp2)
6   ?offer bsbm:vendor ?vendor .                                           (tp3)
7   ?vendor rdfs:label ?vendorTitle .                                      (tp4)
8   ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#GB> . (tp5)
9   ?offer bsbm:validTo ?validDate .                                       (tp6)
10 }
```

Listing 5.8: BGP q7

BGP **q8**, shown in Listing 5.9, contains 6 triple patterns forming 2 star-shaped patterns on variables *review* and *reviewer*. Triple patterns tp1, tp2, tp3, tp4, tp5 are executed using triple map *ReviewMapping* since it contains their respective predicates.
Triple pattern p6 is executed using triple map *ReviewerMapping*, since it contains its respective predicates.

```
1  select
2   *
3  where {
4   ?review bsbm:reviewFor bsbm-inst:Product10 .          (tp1)
5   ?review dc:title ?title .                             (tp2)
6   ?review rev:text ?text .                              (tp3)
7   ?review bsbm:reviewDate ?reviewDate .                 (tp4)
8   ?review rev:reviewer ?reviewer .                      (tp5)
9   ?reviewer foaf:name ?reviewerName .                   (tp6)
10 }
```

Listing 5.9: BGP q8

BGP **q9**, shown in Listing 5.10, contains 2 triple patterns forming 2 star-shaped patterns on variables *product* and *offer*. Triple pattern tp1 is executed using triple map *ProductMapping*, since it is the only triple map containing its predicates.

	Scale factor					
	284826		**1200000**		**4800000**	
BGPs	**Exec. time(ms)**	**Nr. results**	**Exec. time(ms)**	**Nr. results**	**Exec. time(ms)**	**Nr. results**
q1	8330.67	288	14628.33	1111	70314.00	4040
q2	7064.67	132	21445.33	96	66419.33	132
q3	3184.67	2	9875.00	4	36912.67	12
q4	35491.00	5	148742.67	2	443082.67	5
q5	34330.67	4	121381.00	2	337374.00	2
q6	129692.67	11	350329.00	11	1237844.00	11
q7	30678.33	613150	72671.33	2412795	229940.33	9628477
q8	37695.67	5	110857.33	2	304691.33	5
q9	29087.33	16263926	73243.67	68419782	227011.00	274008778

TABLE 5.3: Query Execution Time

On the other hand, the predicate of triple pattern tp2 is found in triple maps *VendorMapping*, *ProductTypeMapping*, *ProductMapping*, *ProductFeatureMapping*, *ReviewerMapping*, *ProducerMapping*, *OfferMapping* and *ReviewMapping*, therefore tp2 is executed using all of them, we make a union of their results for the final answer of the execution of tp2.

```
1 select
2  ?product ?p ?offer
3 where {
4  ?product bsbm:producer ?p .            (tp1)
5  ?offer dc:publisher ?p .               (tp2)
6 }
```

LISTING 5.10: BGP q9

In order to obtain the query execution time of the 9 queries, we ran each query 3 times and reported the mean of them, and this for all the scale factors used. Table 5.3 shows the results of each execution in terms of its mean. Figure 5.2 shows the results graphically.

We can observe that our Data Wrapper continues to deliver quite reasonable [5] performances even with increasing sizes of the test data. The query execution time is not proportional with the increasing data scales. Each scale is 4 times bigger than the previous one, but the query execution time is less than 4 times slower. This confirms the claim that our approach is scalable and performs quite well in large-scales.

[5] We deem reasonable any query results that remain in most cases in the scale of few minutes ($<$ 10 min).

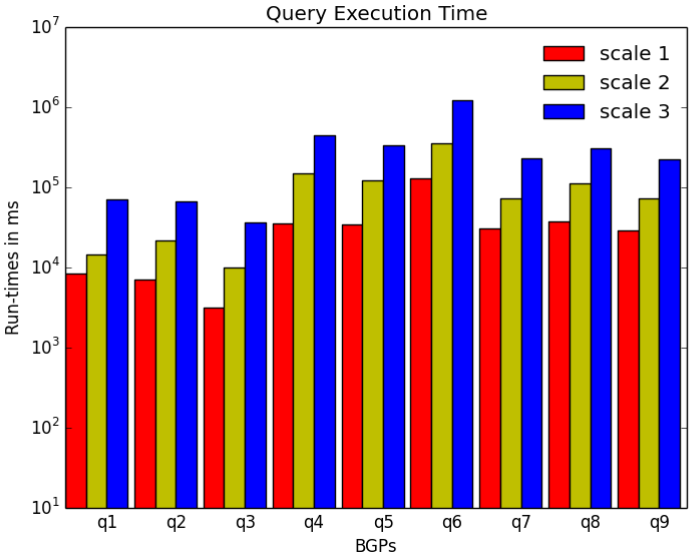

FIGURE 5.2: Query Execution Time. Scale 1, 2 and 3 refer to scale factor 284826, 1200000 and 4800000 respectively.

	Scale Factor					
	1		666		1332	
Source	**Size(KB)**	**#Records**	**Size(MB)**	**#Records**	**Size(MB)**	**#Records**
ProductFeature	104.3	289	1.04	2860	1.72	4745
ProductType	2.61	7	0.0197	55	0.05613	151
Producer	0.335	1	0.0056	14	0.01131	29
Product	1.26	1	0.91973	666	1.82	1332
ProductTypeProduct	0.032	3	0.01203	1998	0.03508	5328
ProductFeatureProduct	0.127	19	0.1295	16768	0.22308	27260
Vendor	0.375	1	0.00282	7	0.00625	15
Offer	3.08	20	2.04	13320	4.14	26640
Person	0.125	1	0.02338	326	0.04863	677
Review	14	10	8.85	6660	17.58	13320
TOTAL	126.244	352	13.04276	42674	25.64048	79497

TABLE 5.4: The CSV files generated using BSBM for the comparison with RML Processor.

5.3 Comparison with RML Processor

In this section we discuss the results obtained comparing RML Processor with our *Static* Data Wrapper. Following the same procedure as in Section 5.1, we generated 3 data sets: scale 1, scale 666 and scale 1332. Each data set produces 1844, 250279 and 491176 distinct triples. Table 5.4 shows the characteristics of the generated data.

For this experiment we used only one machine with the following characteristics:

- MacBook Pro (Mid 2012)
- 2.9 GHz Intel Core i7

	Scale Factor		
	1	**666**	**1332**
RML Processor	1.82	712.61	2121.97
Data Wrapper	12.29	16.62	20.13

TABLE 5.5: Query Execution Time in comparison with RML Processor (in secs.).

- 16 GB RAM
- 500 GB Solid State SATA

In order to obtain the execution time, we have run the process of transforming the whole datasets into an RDF format 3 times and calculated the mean. Table 5.5 shows the results and we can see the graphical comparison in Figure 5.3.
We can observe that RML Processor outperforms our approach by one order of magnitude only when the data is very small (scale factor 1). On the other hand, our Static Data Wrapper outperforms RML Processor by one and two orders of magnitude, for the scale factor 666 and 1332, respectively. This leads us to conclude that our approach performs well with large datasets.

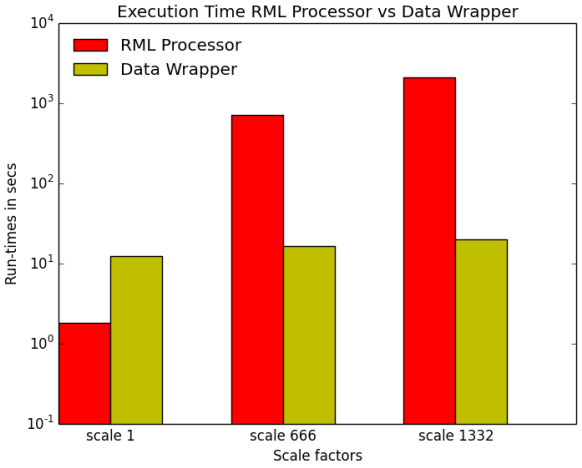

FIGURE 5.3: Query Execution Time.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, we propose an approach for *semantically* querying large-scale non-semantic structured data, by querying non-semantic data using SPARQL BGPs. We devise the Data Wrapper, which is an independent data extraction component able to query multiple data sources, with only a minimal change. The latter is achieved by exploiting Spark's distributed data structures (DataFrames), which abstracts away the differences between the data formats. The semantic enrichment, the Data Wrapper accepts as input alongside the data, are expressed using RML mapping language [5]. We suggest two types of Data Wrappers: the *Dynamic* Data Wrapper uses RML Mappings to query (multiple) non-semantic data; and *Static* Data Wrapper that uses the RML Mappings to obtain the *semantified* (RDF) version of the whole non-semantic data.

The remainder of this chapter concludes the work of this Master's thesis by presenting a list of the limitations that the current version has, and a set of ideas that could enhance and improve the capabilities of the current proposal, i.e. Data Wrapper.

6.2 Limitations

At this moment our approach is limited by the following aspects:

- Our approach is tightly coupled to the usage of RML mapping language. Even though very promising, this latter is not a standard yet and its adoption is not very popular at the moment of writing this thesis.

- We do not support all features of SPARQL language, such as Optional Pattern Matching, Filter evaluation.
- Non-semantic semi-structured data is not supported in the current proposal.

6.3 Future Work

During the development of this thesis, the following ideas arose. We believe that the realization of them would extend/improve the results achieved so far, and tackle the aforementioned limitations.

- **Support of Semantic Data:** This could be achieved using a schema-agnostic approach in which RDF triples are loaded into a large triple table, i.e., a DataFrame consisting of 3 columns *subject*, *predicate*, *object*. The challenge of query optimization could be addressed by implementing the work presented by Tsialiamanis et al, Heuristics-based SPARQL planner (HSP) [33]. They propose the use of a set of heuristics based on syntactic and structural characteristics of SPARQL queries, producing plans that maximize the number of merge joins and reduce intermediate results by choosing triples patterns most likely to have high selectivity, thus reducing the query execution time.
- **Extend the support of non-semantic data:** The goal would be to focus on semi-structured data, specifically in XML and JSON. In the following, we give some suggestions on how can this be accomplished:
  - **XML:** There is a library[1] for parsing and querying XML data with Apache Spark (proposed by Spark developers themselves), for Spark SQL and DataFrames. This package allows to process format-free XML files (via XPath queries). This would fit perfectly with the use of DataFrames that our approach employs.
  - **JSON:** Spark already supports JSON, therefore the schema used by the DataFrame should be constructed using JSONPath to refer to the JSON structure.

  Both formats are supported by RML mapping language, it is only necessary to specify the data type and how to access it.
- **Extend the support of SPARQL features:** The goal would be focus on supporting Optional Pattern Matching, Filter evaluation and Operator Mapping. Following we give suggestions on how to achieve these features:

[1] Databricks spark-xml

  - **Optional Pattern Matching:** Since our approach translates SPARQL to SQL, the aim would be to obtain its SQL equivalent. Optional matching adds information to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. `left outer join` has the same effect in SQL, it performs a regular inner-join but does not eliminate solutions if the join constraints are not met.
  - **Filter evaluation:** Filter statements could be pushed down directly to SQL since this latter supports most SPARQL operators or there is an SQL equivalent to it.

- **Expose Data Wrapper as a REST service:** Spark-jobserver[2] provides a RESTful interface for submitting and managing Apache Spark jobs, JARS, and job contexts. It is only necessary to implement the *SparkJob* trait, overriding methods *runJob* and *validate*.

[2] Spark Job Server

Appendix A

Appendix

A.1 RML mapping for Product of the Berlin Benchmark

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/> .
@prefix bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rev: <http://purl.org/stuff/rev#> .
@prefix schema: <http://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<#ProductMapping>
  rml:logicalSource [
    rml:source "hdfs://localhost:9000/thesis/csv_results/product.csv";
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
    rr:template "http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/Product{nr}";
    rr:class bsbm:Product
  ];
  rr:predicateObjectMap [
    rr:predicate rdfs:label;
```

```
25      rr:objectMap [ rml:reference "label" ]
26    ];
27    rr:predicateObjectMap [
28      rr:predicate rdfs:comment;
29      rr:objectMap [ rml:reference "comment" ]
30    ];
31    rr:predicateObjectMap [
32      rr:predicate bsbm:producer;
33      rr:predicate dc:publisher;
34      rr:objectMap [
35        rr:parentTriplesMap <#ProducerMapping>;
36        rr:joinCondition [
37          rr:child "producer";
38          rr:parent "nr"
39        ];
40      ]
41    ];
42    rr:predicateObjectMap [
43      rr:predicate bsbm:productPropertyNumeric1;
44      rr:objectMap [
45        rml:reference "propertyNum1";
46        rr:datatype xsd:integer
47      ]
48    ];
49    rr:predicateObjectMap [
50      rr:predicate bsbm:productPropertyNumeric2;
51      rr:objectMap [
52        rml:reference "propertyNum2";
53        rr:datatype xsd:integer
54      ]
55    ];
56    rr:predicateObjectMap [
57      rr:predicate bsbm:productPropertyNumeric3;
58      rr:objectMap [
59        rml:reference "propertyNum3";
60        rr:datatype xsd:integer
61      ]
62    ];
63    rr:predicateObjectMap [
```

```
64      rr:predicate bsbm:productPropertyNumeric4;
65      rr:objectMap [
66        rml:reference "propertyNum4";
67        rr:datatype xsd:integer
68      ]
69    ];
70    rr:predicateObjectMap [
71      rr:predicate bsbm:productPropertyNumeric5;
72      rr:objectMap [
73        rml:reference "propertyNum5";
74        rr:datatype xsd:integer
75      ]
76    ];
77    rr:predicateObjectMap [
78      rr:predicate bsbm:productPropertyNumeric6;
79      rr:objectMap [
80         rml:reference "propertyNum6";
81         rr:datatype xsd:integer
82      ]
83    ];
84    rr:predicateObjectMap [
85      rr:predicate bsbm:productPropertyTextual1;
86      rr:objectMap [
87        rml:reference "propertyTex1";
88        rr:datatype xsd:string
89      ]
90    ];
91    rr:predicateObjectMap [
92      rr:predicate bsbm:productPropertyTextual2;
93      rr:objectMap [
94        rml:reference "propertyTex2";
95        rr:datatype xsd:string
96      ]
97    ];
98    rr:predicateObjectMap [
99      rr:predicate bsbm:productPropertyTextual3;
100     rr:objectMap [
101       rml:reference "propertyTex3";
102       rr:datatype xsd:string
```

```
103     ]
104   ];
105   rr:predicateObjectMap [
106     rr:predicate bsbm:productPropertyTextual4;
107     rr:objectMap [
108       rml:reference "propertyTex4";
109       rr:datatype xsd:string
110     ]
111   ];
112   rr:predicateObjectMap [
113     rr:predicate bsbm:productPropertyTextual5;
114     rr:objectMap [
115       rml:reference "propertyTex5";
116       rr:datatype xsd:string
117     ]
118   ];
119   rr:predicateObjectMap [
120     rr:predicate bsbm:productPropertyTextual6;
121     rr:objectMap [
122       rml:reference "propertyTex6";
123       rr:datatype xsd:string
124     ]
125   ];
126   rr:predicateObjectMap [
127     rr:predicate dc:date;
128     rr:objectMap [
129       rml:reference "publishDate";
130       rr:datatype xsd:date
131     ]
132   ].
```

Listing A.1: RML mapping for Product of the Berlin Benchmark

Bibliography

[1] R2RML: RDB to RDF Mapping Language. URL http://www.w3.org/TR/r2rml/.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[3] Frank Manola and Eric Miller. RDF Primer, February 2004. URL www.w3.org/TR/rdf-primer/.

[4] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001. URL http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

[5] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the 7th Workshop on Linked Data on the Web*, April 2014. URL http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_01.pdf.

[6] Apache Spark Project. URL http://spark.apache.org.

[7] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[8] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[9] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[10] Tim Berners-Lee. Design Issues: Linked Data. *W3C Design Issues*, 2006. URL http://www.w3.org/DesignIssues/LinkedData.html.

[11] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009. ISSN 0362-5915. doi: 10.1145/1567274.1567278. URL http://doi.acm.org/10.1145/1567274.1567278.

[12] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 Turtle, February 2014. URL www.w3.org/TR/turtle/.

[13] David Beckett, Gavin Carothers, and Andy Seaborne. RDF 1.1 N-Triples, February 2014. URL www.w3.org/TR/n-triples.

[14] Gavin Carothers. RDF 1.1 N-Triples, February 2014. URL www.w3.org/TR/n-triples.

[15] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. JSON-LD 1.0, January 2014. URL www.w3.org/TR/json-ld/.

[16] Gandon. Fabien, Guus Schreiber, and Dave Beckett. RDF 1.1 XML Syntax, February 2014. URL www.w3.org/TR/rdf-syntax-grammar/.

[17] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, January 2008. URL www.w3.org/TR/rdf-sparql-query/.

[18] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.

[19] James Dixon. Pentaho, Hadoop, and Data Lakes, October 2010. URL http://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/.

[20] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on Spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.

[21] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. Sempala: interactive sparql query processing on hadoop. In *International Semantic Web Conference*, pages 164–179. Springer, 2014.

[22] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web*, pages 397–400. ACM, 2012.

[23] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *Proc. VLDB Endow*, 4 (12):1426–1429, 2011.

[24] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. Pigsparql: a sparql query processing baseline for big data. In *Proceedings of the 2013th International Conference on Posters & Demonstrations Track-Volume 1035*, pages 241–244. CEUR-WS. org, 2013.

[25] Kurt Rohloff and Richard E Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the shard graph-store. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, pages 35–44. ACM, 2011.

[26] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *Proc. VLDB Endow*, 4 (12):1426–1429, 2011.

[27] Mohamed Nadjib Mami, Simon Scerri, Sören Auer, and Maria-Esther Vidal. Towards Semantification of Big Data Technology. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 376–390. Springer, 2016.

[28] Christoph Quix, Rihan Hai, and Ivan Vatov. Gemms: A generic and extensible metadata management system for data lakes. In *CAiSE forum*, 2016.

[29] Coral Walker and Hassan Alrehamy. Personal data lake with data gravity pull. In *Big Data and Cloud Computing (BDCloud), 2015 IEEE Fifth International Conference on*, pages 160–167. IEEE, 2015.

[30] Mario Arias, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. *arXiv preprint arXiv:1103.5043*, 2011.

[31] Christian Bizer and Andreas Schultz. The Berlin Sparql Benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[32] Peter Pin-Shan Chen. The Entity-Relationship Model-Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976. ISSN 0362-5915. doi: 10.1145/320434.320440. URL http://doi.acm.org/10.1145/320434.320440.

[33] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based query optimisation for sparql. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 324–335. ACM, 2012.