Master Thesis

Distributed Evaluation of SPARQL queries with Impala and MapReduce

Antony R. Neu

24.03.2014

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum

27. 09. 2013 – 27. 03. 2014

Erstgutachter

Prof. Dr. Georg Lausen

Zweitgutachter

Prof. Dr. Peter Fischer

Betreuer

Alexander Schätzle

Abstract

The vast growth of the Semantic Web requires efficient storage of large RDF documents and evaluation of SPARQL queries on this data. This thesis deals with the feasibility of efficient querying RDF data stored in the distributed database *Cloudera Impala*, which is based on the open-source framework Hadoop. For this purpose, different storage schemas were tested for RDF data in Impala using the columnar storage format Parquet. Sempala, the software implementation developed for this thesis, stores RDF data in a property table and provides a translation for SPARQL 1.0 queries to the SQL dialect used by Impala. During the evaluation, the performance of Sempala was tested on a computer cluster using three known Semantic Web benchmarks: LUBM, BSBM and SP^{2}Bench. The runtimes of Impala are compared with the runtimes of Hive, a data warehouse software used in Big Data environments. This thesis has two main results: On the one hand, in-memory joins lead to limitations in Impala for unselective queries, on the other hand, the evaluation shows an improvement of query runtimes by one order of magnitude for Impala compared with Hive.

Keywords: Cloudera Impala, Parquet, SPARQL, Semantic Web, Hadoop, MapReduce

Zusammenfassung

Die immer stärker wachsenden Daten des Semantic Webs erfordern eine effiziente Speicherung und Möglichkeiten der Auswertung von SPARQL-Anfragen auf diesen Daten. Gegenstand dieser Arbeit ist die effiziente Auswertung von SPARQL Anfragen auf RDF-Daten, die in der verteilten Datenbank *Cloudera Impala* gespeichert werden. Hierfür werden verschiedene Speicherschemata getestet und deren Eignung für das spaltenorientiere Datenformat *Parquet* diskutiert. Sempala, die Implementierung dieser Arbeit, speichert RDF-Daten in einer *property table* und bietet die Möglichkeit der Übersetzung von SPARQL-Anfragen in den SQL-Dialekt, der von Impala verwendet wird. Die bekannten Semantic Web benchmarks, LUBM BSBM und SP^{2}Bench, werden verwendet um die Leistungsfähigkeit Sempalas auf einem Computercluster zu evaluieren. Die Laufzeiten werden mit dem Data-Warehouse-System Hive verglichen, das bereits im Big-Data-Bereich eingesetzt wird. Die Ergebnisse zeigen zum einen Impalas Limitierung auf In-Memory-Joins und die daraus resultierenden Probleme bei unselektive Anfragen. Andererseits zeigt der Einsatz von Impala eine signifikante Verbesserung der Laufzeiten verglichen mit dem Einsatz von Hive.

Keywords: Cloudera Impala, Parquet, SPARQL, Semantic Web, Hadoop, MapReduce

Contents

1. Introduction

1.1. Motivation

The *Semantic Web* describes the idea of complementing the current world wide web, which consists mostly of unstructured documents, with a "web of data" [1]. The idea was made public in 2001 by Tim Berners Lee, the director of the W3C. In this web of data, the documents feature machine-readable annotations, allowing machines to understand the meaning of the data. Thus, machines can act as intelligent agents, while combining information from different data sources. However, in the first few years after the initial publication, the Semantic Web only existed in academic and scientific environments as separate pools of data. Berners-Lee demanded *linked data* in 2006 [6] and provided a set of rules in order to achieve this. The use of open standards - namely *RDF* and *SPARQL* - allows others to reference published data, resulting in links between the separate data pools.

The *Resource Description Framework (RDF)* is an open W3C-recommended data format for the Semantic Web. The information is broken down to a set of statements which each consists of subject, predicate and object called a triple. These triples are stored in so-called *triple stores*. The query language *SPARQL* was developed, in order to gain knowledge from the data. Two requirements must be met for the Semantic web vision to become feasible: the large amounts of RDF data must be on the one hand stored economically and on the other hand accessible for efficient query evaluation. Single-place RDF stores do not offer the scalability required for the growing sets of RDF data.

Over the last few years, large IT companies such as Facebook, Google and Twitter have developed technologies to store and process *Big Data*, a term describing data too large for conventional data analysis. Even though these technologies were developed without Semantic Web in mind, many technologies were able to be adapted

for Semantic Web data processing. Along with NoSQL-databases, *MapReduce* – especially its open-source implementation *Apache Hadoop* – has become a well-known standard for Big Data processing [2]. As Hadoop runs on commodity hardware, large scale data analysis becomes affordable - even for small companies and academic institutions. Hadoop and its related projects, such as Hive and Pig, only allow batch processing where real-time queries are not a requirement. However, this is not always sufficient. In order to overcome this, Cloudera presented the open-source software *Cloudera Impala* in October 2012 to complement the existing ecosystem [3]. It allows users to run interactive SQL queries against data stored in Hadoop. Interactive queries are slower than real-time queries and have a duration ranging from seconds to a few minutes. In many use-cases, SPARQL queries are interactive queries.

1.2. Goal

The goal of this thesis is to investigate whether Cloudera Impala is suitable for the storage of large RDF data and efficient evaluation of SPARQL queries on this data. For this purpose, a software called *Sempala* was implemented, which consists of two components. The first component loads the RDF data into the relational Impala database. During the development of Sempala, different storage strategies were compared in order to find the most efficient strategy. The second component provides a translation of SPARQL queries to the SQL dialect used by Impala. After the implementation, three known benchmarks were used to test the performance of Sempala. As Impala was designed to complement other Hadoop technologies, the performance of Impala was compared with the performance of Hive, a widely used data warehousing software for Big Data running the same queries. Thus, the advantages of Impala on the one and hand and its limitations one the other hand became apparent. Apart from that, the runtimes of the queries were compared for different data set sizes to test the scalability and the suitability for large Semantic Web data processing.

1.3. Structure

The remainder of the thesis is structured as follows: **Chapter 2** gives an introduction to Semantic Web technologies and presents the Apache Hadoop Framework, which is used in the software implementation. Cloudera Impala is introduced in the following **Chapter 3** by presenting its architecture and how it relates to Hadoop and other SQL databases. Before the final implementation of Sempala, three different storage strategies for RDF in Impala were tested and evaluated. Their concepts and evaluation findings are described in **Chapter 4**. The final implementation of Sempala, which loads RDF data into Impala and translates SPARQL queries to SQL queries, is presented in **Chapter 5**. In order to test performance of the implementation, three known SPARQL benchmarks, LUBM, SP^{2}Bench and BSBM benchmark, were used to generate data and queries and evaluate the performance of the system on a computer cluster. The runtimes and further results are presented in **Chapter 6**. In **Chapter 7**, related works and similar database technologies are compared with Impala and the results of this thesis whose findings are summarized in **Chapter 8**.

2. Background

In this chapter, the necessary background information is given, in order to understand the following chapters of this thesis. First of all, the ideas, goals and concepts of the Semantic Web are described in section 2.1, followed by introductions to RDF and SPARQL, two standards of the Semantic Web. After that, Apache Hadoop is presented in section 2.4 with an example program, as this technology is used in the implementation to load the datasets into Cloudera Impala. Cloudera Impala itself is introduced in the following chapter, Chapter 3.

2.1. Semantic Web

The idea of the Semantic Web was first described in Scientific America [1] in 2001 by Tim Berners-Lee, the director of the W3C. His vision of a machine-readable extension of web documents would enable intelligent computer systems to request and gather information autonomously without any user interaction. Without this semantic extension, machines can only use the limited means of natural language processing in order to understand the meaning of information stored in a page [4]. For example, machines would not be able to distinguish whether a web document contains information about "jobs", meaning work, or information on the founder of Apple, i.e. Steve Jobs.

In 2006, Berners-Lee asserted that proprietary formats were still favored and open data formats rarely used [5]. As a consequence, the web consisted of separate pools of data which were not connected. In order to change this, Berners-Lee argued for an open data format [6] and unique identifiers, which could be used to reference resources from other dataset and thus link the data. Newer developments show efforts of major IT companies towards open data standards. *schema.org*[1] is an example of

[1] http://www.schema.org/

such newer developments. As a collaboration of the major search engine providers Google, Yahoo and Microsoft, *schema.org* provides a shared markup vocabulary for webmasters. In this way, webmasters can annotate their pages, knowing that all major search engines will use and understand the data. The semantic markup is provided as RDFa, which embeds RDF data in HTML pages.

Another example is *Linking Open Data*, a W3C community project, aiming at interlinking data which is freely available [7]. The interconnected datasets of linked open data, also referred to as *LOD-cloud*, became popular and grew rapidly. In 2010, the LOD-cloud consisted of 38 billion RDF triples. Parts of it are shown in Figure 2.1.

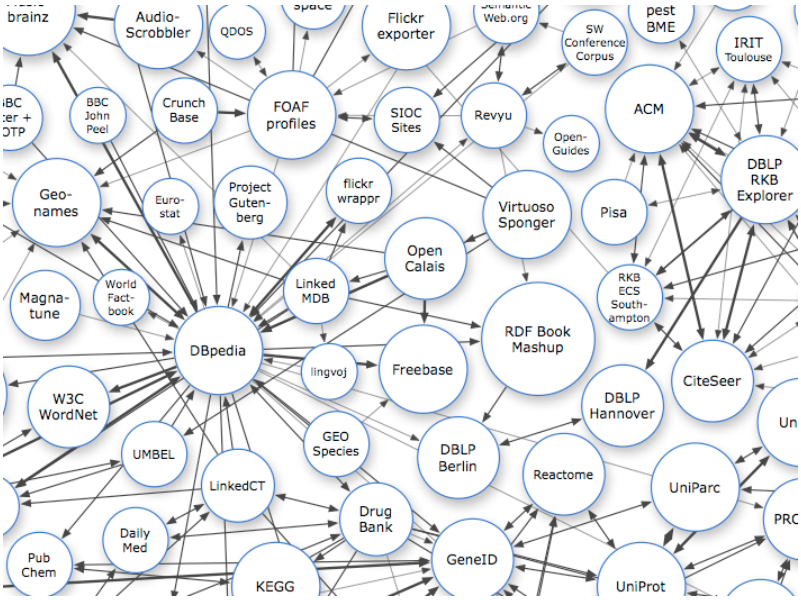


Figure 2.1.: Semantic web data (source: `http://linkeddata.org/`)

2.2. RDF

The Resource Description Framework (RDF) is a formal language to describe information in a machine-readable data format without losing the meaning of this information. It was originally developed to add meta information to web documents, such as author and publication date. However, it can be used to describe all "things that can be identified on the Web" [8]. In the following, the essential elements of the RDF language [4] [8] are described.

- **URIs and blank nodes:** The RDF data format uses Uniform Resource Identifiers(URIs) to accurately and unambiguously reference resources. These identifiers are unique in the entire web and can be created by different persons or organizations independently. An example of a URI are URLs, e.g. the URL `<http://www.example.org/index.html>`. This URL can be used to uniquely identify the example resource in any RDF document. To shorten the length of the URIs, prefixes can be used. With `ex:` defined as prefix for `http://www.example.org/`, the example URI is shortened to `ex:index.html`. *Blank nodes* can be used to uniquely identify a node within a certain RDF document without explicitly naming it with a URI. However these identifiers cannot be used in other documents to refer to a certain resource outside the respective document .

- **RDF triple:** An RDF document is a set of statements consisting of three elements: subject, predicate and object. Such a statement is hence also referred to as triple. Each statement describes a property (predicate) of a resource (subject) by giving it a value (object). The subject and object are either a URI or a blank node if the resource cannot or should not be identified explicitly. The object can be a literal, if a specific property value is to be described instead of a relationship between two resources. These literals are strings that can additionally be typed by a data type given as URI. `"Adamanta Schlitt"^^xsd:string` is an example of a literal typed by the data type `xsd:string`.

- **RDF Encoding formats:** There are different syntax encodings for RDF documents. XML, Turtle and N3 syntax are the most widely used formats. The simplest syntax lists RDF triples one after another. Each statement is followed by ”.” – similar to the natural written language. Prefixes can be declared at the beginning of the document in order to shorten URIs. Listing 2.1 shows an example RDF document.

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix per: <http://localhost/persons/> .

per:Paul_Erdoes rdf:type foaf:Person.
per:Paul_Erdoes foaf:name "Paul Erdoes"^^xsd:string.
per:Amanda_Knight rdf:type foaf:Person.
per:Amanda_Knight foaf:name "Amanda Knight"^^xsd:string.
<http://localhost/Article3> rdf:type dc:Document.
<http://localhost/Article3> dc:creator per:Paul_Erdoes.
<http://localhost/Article3> dc:creator per:Amanda_Knight.
<http://localhost/Article3> dc:title "Example"^^xsd:string.
<http://localhost/Article4> rdf:type dc:Document.
<http://localhost/Article4> dc:creator per:Amanda_Knight.
```

Listing 2.1: Example RDF data

- **RDF graphs:** RDF documents can also be viewed as directed graphs. Each triple describes a directed edge from a node labeled by the subject to a node labeled by the object. The edge itself is labeled by the predicate. In this case, the RDF documents are referred to as RDF graphs. Figure 2.2 shows the corresponding RDF graph for Listing 2.1.

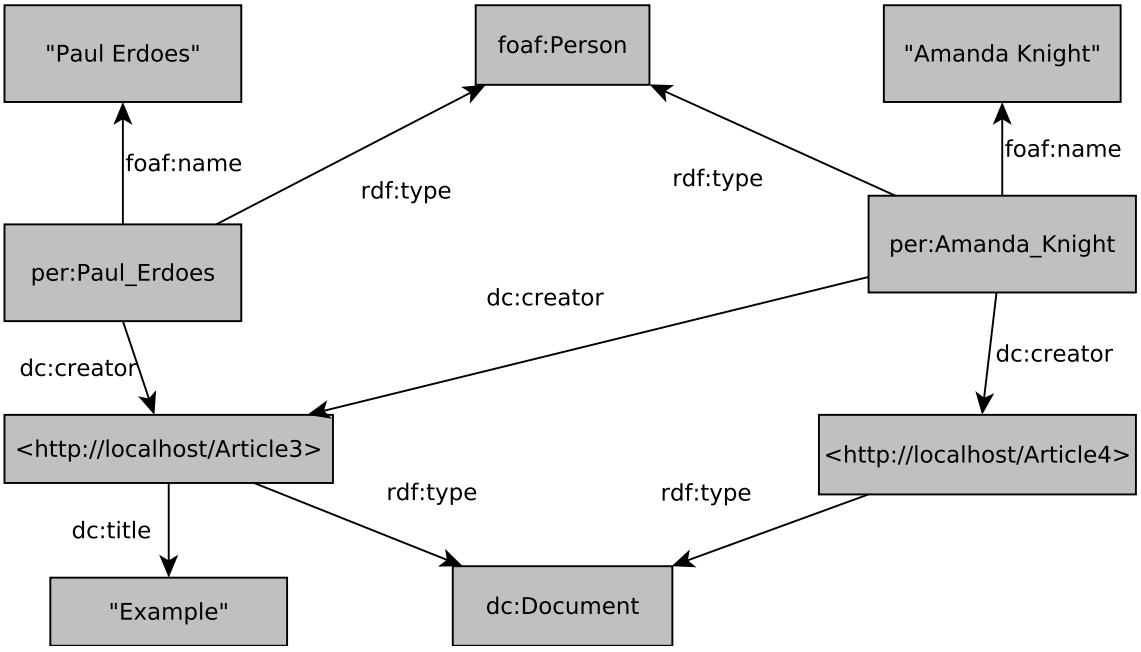


Figure 2.2.: Example RDF document from Listing 2.1 as graph representation

2.3. SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a query language for RDF data similar to SQL for data stored in relational databases. The first version of the language, SPARQL 1.0, was declared an official W3C recommendation on January 15 2008 [9]. The language was extended by features such as aggregates and property paths resulting in the SPARQL 1.1 W3C recommendation (March 21 2013) [10].

This introduction will focus on those language features supported by Sempala, namely the SPARQL 1.0 standard. A comprehensive list of all language features can be found on the official W3C (world wide web consortium) recommendation page [9]. First, the language itself is introduced informally by examples, followed by an introduction into the semantics of SPARQL [4][11].

A SPARQL query shares syntactical elements with RDF data in Turtle format. Listing 2.2 shows a SPARQL *SELECT* query. There are other query types, such as *DESCRIBE* and *CONSTRUCT* queries, which are not supported by Sempala and will therefore be omitted at this point.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE {
  ?creator rdf:type foaf:Person.
  ?creator foaf:name ?name.
  <http://localhost/Article3> dc:creator ?creator. }
```

Listing 2.2: Example SPARQL query

Every SELECT query consists of three parts separated by the keywords *PREFIX*, *SELECT* and *WHERE*:

- In order to shorten URIs, prefixes can be defined at the beginning of the SPARQL query using the keyword *PREFIX* which is also similar to the Turtle RDF format. As an alternative, a base URI can be defined by the keyword *BASE* which adds the prefix to all nodes which do not have a prefix defined.
- The result format is defined by the keyword *SELECT* followed by those variables that should be considered in the output. This is very similar to *SELECT*

?name
"Paul Erdoes"^^xsd:string
"Amanda Knight"^^xsd:string

Table 2.1.: Superset of solution mappings for example SPARQL query in Listing 2.2.

in *SQL* queries.

- The keyword WHERE followed by a *basic graph pattern* encapsulated in curly brackets represents the actual query and defines which nodes and edges of an RDF graph are valid for the result.

2.3.1. Basic graph patterns

Each SPARQL query consists of at least one basic graph pattern enclosed in brackets after the keyword *WHERE*. A basic graph pattern consists of at least one triple. Not only, these triples contain URIs and blank nodes like RDF documents but they can also include variables, which are marked by *'?'*. Variables can be subject, predicate or object of a statement. A variable is a placeholder for any value in the RDF document that fulfills the rest of the statement. For example, the first triple in Listing 2.2 maps the variable *?creator* to those subjects which appear in a triple with the predicate *rdf:type* and object *foaf:Person*. Each SPARQL query results in a set of all possible mappings of the variables in the query called *solution mappings*. If a variable appears in another triple, it has to be mapped to the same value in both triples to be a valid solution for the query. With the *SELECT* keyword, a subset of the variables, which should be returned as result, can be defined. The example in Listing 2.2 queries the names of all creators of the given article identified by the URI `<http://localhost/Article3>` which are of type *foaf:Person*. The query yields the superset of solution mappings in Table 2.1 for the RDF document in Listing 2.1. Hereby, each table cell is a solution mapping, each row forms a set of these mappings and the table hence represents a superset of solution mappings.

?name	?creator	?title
<http://localhost/Article3>	per:Paul_Erdoes	"Example"^^xsd:string
<http://localhost/Article3>	per:Amanda_Knight	"Example"^^xsd:string
<http://localhost/Article4>	per:Amanda_Knight	

Table 2.2.: Solution mappings for example SPARQL query in Listing 2.3.

2.3.2. Optional mappings

In logical terms, every triple acts as a condition that must be fulfilled by the corresponding triple in the RDF document to be a valid solution. If two triples share variables, then these triples act as a logical conjunction of those conditions. However, with the keyword *OPTIONAL* followed by a basic graph pattern, optional mappings can be defined, i.e. mappings which do not necessarily contain a value. The query engine checks if mappings exist for the basic graph pattern enclosed by *OPTIONAL*. In this case, the solution is extended by this mapping. If such a mapping does not exist, the mapping is not extended. For example, Listing 2.3 lists all documents, their creators and their title, if there is a title available.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?article ?creator ?title
WHERE {
        ?article rdf:type dc:Document.
        ?article dc:creator ?creator.
        OPTIONAL{
                ?article dc:title ?title
        }
}
```

Listing 2.3: Example SPARQL query with OPTIONAL

This yields the superset of solution mappings illustrated in Table 2.2. The column *?title* only contains a value for those articles with title.

per:Paul_Erdoes
per:Amanda_Knight
per:Paul_Erdoes

Table 2.3.: Solution mappings for SPARQL query in Listing 2.4.

2.3.3. Combining solution mapping sets

Two solution mapping supersets can be combined with the keyword *UNION*, with each superset encapsulated by curly brackets. The operator acts like a logical non-exclusive or. Thus, each mapping set in the resulting solution superset is either present in one of the two solutions supersets or in both. The query in Listing 2.4 searches for all authors who are creator of the resource identified by `<http://localhost/Article3>` or `<http://localhost/Article4>` or are creator of both. As can be seen in Table 2.3, the result sets of a SPARQL query can contain duplicates, unless the keyword *DISTINCT* is present.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?author
WHERE {
  { <http://localhost/Article3> dc:creator ?creator .}
        UNION
  { <http://localhost/Article4> dc:creator ?author .} }
```

Listing 2.4: Example SPARQL query with *UNION*

2.3.4. Filtering results

Every triple in a basic graph pattern acts as a filter, as only these mappings are considered which match all triples in the basic graph pattern. The keyword FILTER followed by a condition can be used to match variables by further conditions not limited to equality. Many logical operators and built-in functions are available, such as *'<', '>', '='*, and *'!'*. The function *bound(?var)* can be used in combination with *OPTIONAL* to check whether the variable *?var* is bound to a value. The query in Listing 2.5 selects all those entities with a value defined for the property *swrc:pages* and this value is less than 20.

```
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?article
WHERE {
        ?article swrc:pages ?pages.
        FILTER(?pages <20) }
```

Listing 2.5: Example SPARQL query with *FILTER*

2.3.5. Solution modifiers

Solution modifiers do not change the number of solution mappings, i.e. the number of rows in the table, but the format how these rows are returned to the user. The *SELECT* statement allows to select which variables should be considered in the resulting superset of mappings. The keywords *ORDER BY*, *LIMIT* and *OFFSET* order the result set or limit it to the first top results with optional offset, respectively. The query in Listing 2.6 selects the first ten titles of all articles with page numbers in sorted order, starting at the sixth article (offset 5).

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?title
WHERE { ?article dc:title ?title
        ?article swrc:pages ?pages. }
ORDER BY ?pages
LIMIT 10
OFFSET 5
```

Listing 2.6: Example SPARQL query with *ORDER BY*, *LIMIT* and *OFFSET*

2.3.6. SPARQL semantics

The semantics of SPARQL queries are specified by the SPARQL algebra [11][4], which relates to SPARQL in the same way as the relational algebra relates to SQL. The algebra consists of well-defined operators allowing to calculate the result of a SPARQL query. In the following, the translation of those SPARQL language elements to SPARQL algebra operators supported by the implementation is described. This is followed by an example. A more comprehensive overview can be found in [11] and [4].

operator	SPARQL syntax	meaning
BGP(T)	list of triples separated by ”.”	Matching triples in the RDF document are found.
Distinct (M)	DISTINCT	Only distinct values are selected.
Filter(F, M)	FILTER	The mappings from M which satisfy F are valid.
LeftJoin (M_1, M_2, F)	OPTIONAL	The mappings M_1 and M_2 are combined optionally, if the filter condition F is satisfied.
Slice (M, start, length)[i]	LIMIT or OFFSET	Solution modifiers which influence how the result is returned.
Project(M, Vars)	Variables in SELECT	Only those mappings with variables from Vars are selected.
Union (M_1, M_2)	{P} UNION {P}	The two solution mapping sets M_1 and M_2 are united.

Table 2.4.: SPARQL language elements and their corresponding SPARQL algebra operators

Table 2.4 shows those SPARQL language elements, which are relevant for the implementation, and their corresponding SPARQL algebra operators.

Every SPARQL operator except for *BGP* receives a *solution* and/or a filter expression as input and yields a new *solution* as result, which can be visualized as a table (see Table 2.2 as example). In this table, each row represents a set of mappings of variables to RDF terms (empty node, URI or literal), whereas some variables are not bound to a value. Formally this mapping can be described by a partial function μ, whose domain is the set of all variables in the SPARQL query. Every SPARQL query therefore results in a table, which formally can be described as a sequence of partial functions, called *solution*. Z is a solution which contains a set, which does not assign a value to any variable.

Let G be an RDF graph. Let Ψ, Ψ_1 and Ψ_2 be resulting solutions and F a filter expression. The solution of the operators can then be calculated as follows [4]:

expression	result
$\mu_1 \cup \mu 2$	- $\mu_1(x)$ if x is in the domain of $\mu_1(x)$ - $\mu_2(x)$ if x is in the domain of $\mu_2(x)$ - not defined in all other cases
$Filter(\Psi, F)$	$\{\mu \vert \mu \in \Psi$ and $\mu(F)$ yields *true* as result $\}$
$Join(\Psi_1, \Psi_2)$	$\{\mu_1 \cup \mu_2 \vert \mu_1 \in \Psi_1, \mu_2 \in \Psi_2$ and μ_1 compatible to $\mu_2\}$
$LeftJoin(\Psi_1, \Psi_2, F)$	$\{\mu_1 \cup \mu_2 \vert \mu_1 \in \Psi_1, \mu_2 \in \Psi_2$ and μ_1 compatible to μ_2 and $\mu_1 \cup \mu_2(F)$ yields *true* as result $\}$ $\cup$ $\{ \mu_1 \vert \mu_1 \in \Psi_1$ and for all $\mu_2 \in \Psi_2$ holds: μ_1 not compatible to μ_2 or $\mu_1 \cup \mu_2(F)$ yields *false* $\}$
$Union(\Psi_1, \Psi_2)$	$\{ \mu \vert \mu \in \Psi_1$ or $\mu \in \Psi_2 \}$

Table 2.5.: Caption

2.3.7. Translation to SPARQL algebra syntax tree

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?title ?author
WHERE{
  { ?article dc:title ?title
    ?article dc:author ?author.
    ?author foaf:name ?name
    FILTER(?name = "Paul Erdoes")
  } UNION {
    ?article dc:title ?title .
    ?article swrc:pages ?pages.
    FILTER(?pages > 20)
    OPTIONAL{ ?article dc:author ?author. }
  }
}
ORDER BY ?pages
LIMIT 10
OFFSET 5
```

Listing 2.7: Example SPARQL query

Listing 2.7 shows an example SPARQL query with many different nested operators. The result of the query are all titles and authors of those articles which are either written by Paul Erdoes or have more than 20 pages. In the latter case, the author is only optional and therefore need not have a value. The translation of the query into a SPARQL algebra syntax tree can be found in Figure 2.3. The nodes represent either a triple, a SPARQL algebra operator, a filter condition or a list of variable names. If a node m is connected to an operator node n with an edge pointing towards n, the node n receives the solution or value of the node m as input. The triple nodes form the leaves of the tree and form the input for the BGP operator nodes which they belong to, which are located one level above the leaves. These BGP nodes yield a solution superset as result. This result is passed on to the operator node which lies above it. This process can be continued recursively until the root of the tree is reached.

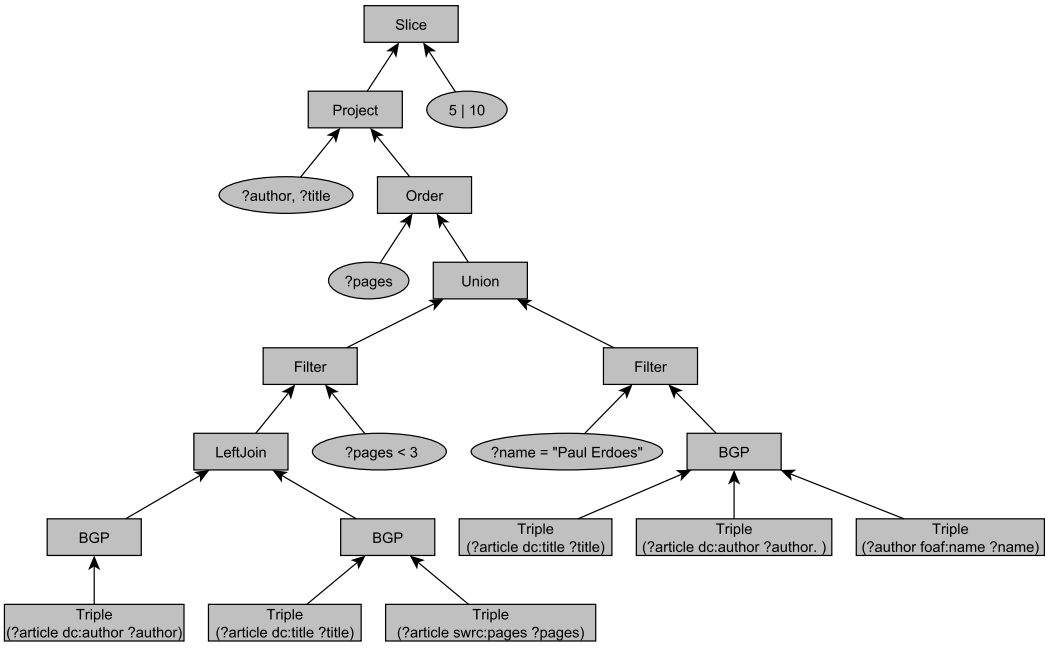


Figure 2.3.: SPARQL syntax tree for query in Listing 2.7

2.4. Apache Hadoop

Apache Hadoop[2] is an open-source framework implemented in Java for distributed processing of large data. It was originally developed by Doug Cutting, the creator of Apache Lucene, and has its roots in the Apache Nutch project, an open-source search engine [2]. In 2003 and 2004, Google published papers on Google Filesystem (GFS) [12] and Google MapReduce [13], in which the developers described how they solved the problem of processing Google's large-sized web search indexes. Thereafter, Hadoop became its own project implementing its own versions of Google's technologies: Hadoop MapReduce and Hadoop Distributed File System (HDFS). Besides these two modules, the Hadoop project consist of two supporting modules Yarn and Hadoop Common. Hadoop is designed to run on computer cluster sizes from just a few to thousands of machines and also runs on Amazon's Elastic Compute Cloud (EC2)[2].

HDFS allows the user to store data in blocks which are distributed in the computer cluster. By default, the data is replicated three times in the cluster. As Hadoop is designed to run on commodity hardware, the replication allows higher data security in the event of hardware failure. The second reason for this data replication is the data locality principle. When large-scale data is processed in a computer cluster, the network bandwidth can become a bottleneck, if large amounts of data must be copied to other machines. Hadoop MapReduce solves this problem by initially allowing each machine to process only that data available on its local storage device. Through data replication on HDFS, the data is available to more machines in the cluster, which allows parallel processing.

MapReduce programs follow a special paradigm and must essentially implement two methods, a map and a reduce method. These programs are called jobs and consist of a map and a reduce phase which correspond to the implemented methods and are referred to as map and reduce tasks when run. Figure 2.4 depicts the MapReduce workflow.

At the beginning, each node in the cluster runs map tasks and reads that part of the input data which is locally available to the node. This input split consists of key-value pairs, *MapInputKey* and *MaputInputValue*. They are processed by the

[2]http://hadoop.apache.org/

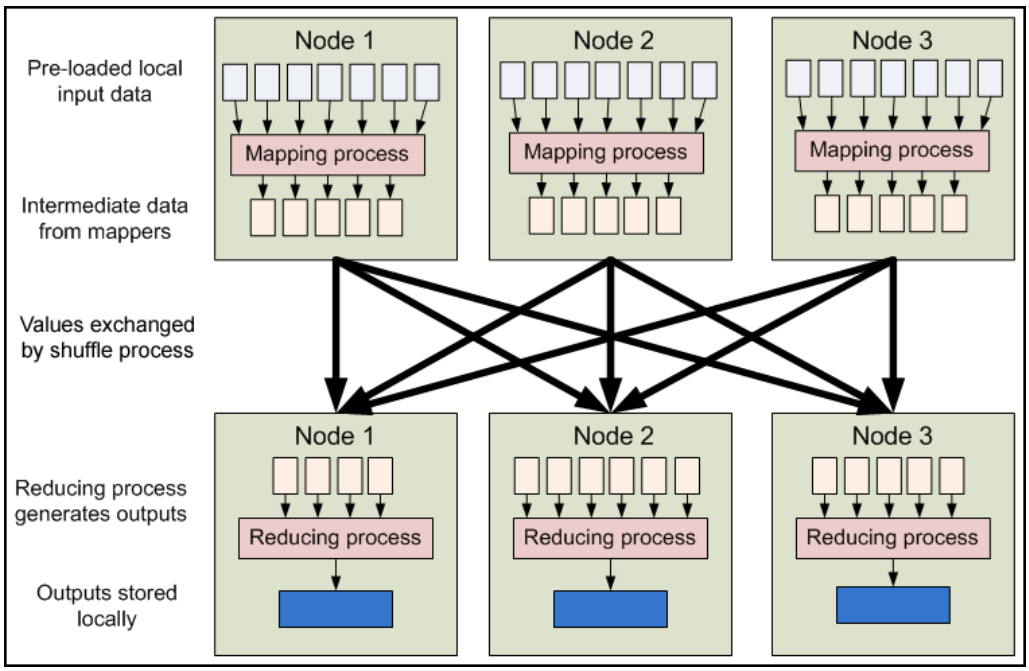


Figure 2.4.: MapReduce job workflow (source: [14])

map method. The result of the method is also a key-value pair – *MapOutputKey* and *MapOutputValue*. The map phase is followed by the shuffle phase, in which the map output pairs are sorted and grouped by the MapOutputKey. They are then sent as input to reduce tasks, which may run on different machines. A reduce task receives all values which have the same *MapOutputKey* and were grouped in the shuffle phase. Consequently the ReduceInputKey is of the same format as the MapoutputKey. Each reduce task and its corresponding method receives a key (ReduceInputKey) and a set of values (ReduceInputValues) as input. Its result is a set of key-value pairs (ReduceOuputKey, ReduceOutputValue). At the end of the MapReduce job, all values are written to HDFS and intermediate results are deleted. It is possible to set the number of reduce tasks to zero. In this case, both shuffle and reduce phase are skipped, which means that the map results are written unsorted to HDFS [2][14].

A common example of a Hadoop is the word count example [14]. Figure 2.5 shows an example run of the MapReduce job for two nodes. The input is a text file stored in HDFS, while the desired output is a distinct list of all words and the number of their occurrences in the input file. Every mapper receives an input split of the original file and calls the map method on each line of this split. Iterating over the words in the line, the mapper emits the word as key and the integer "1" as value.

Once all mappers have finished processing their input splits, the emitted pairs are sorted and grouped by the key, i.e. the word. Consequently, when the reducer calls the reduce method for a key, it has a list of all "1"s emitted by different mappers available to it. It guarantees this word is only processed on a single reducer and hence in a single call of the reduce method. The reducer sums up the values and emits the key (word) and its summed up values. Thus, the result is a sorted list of all words and their respective occurrences.

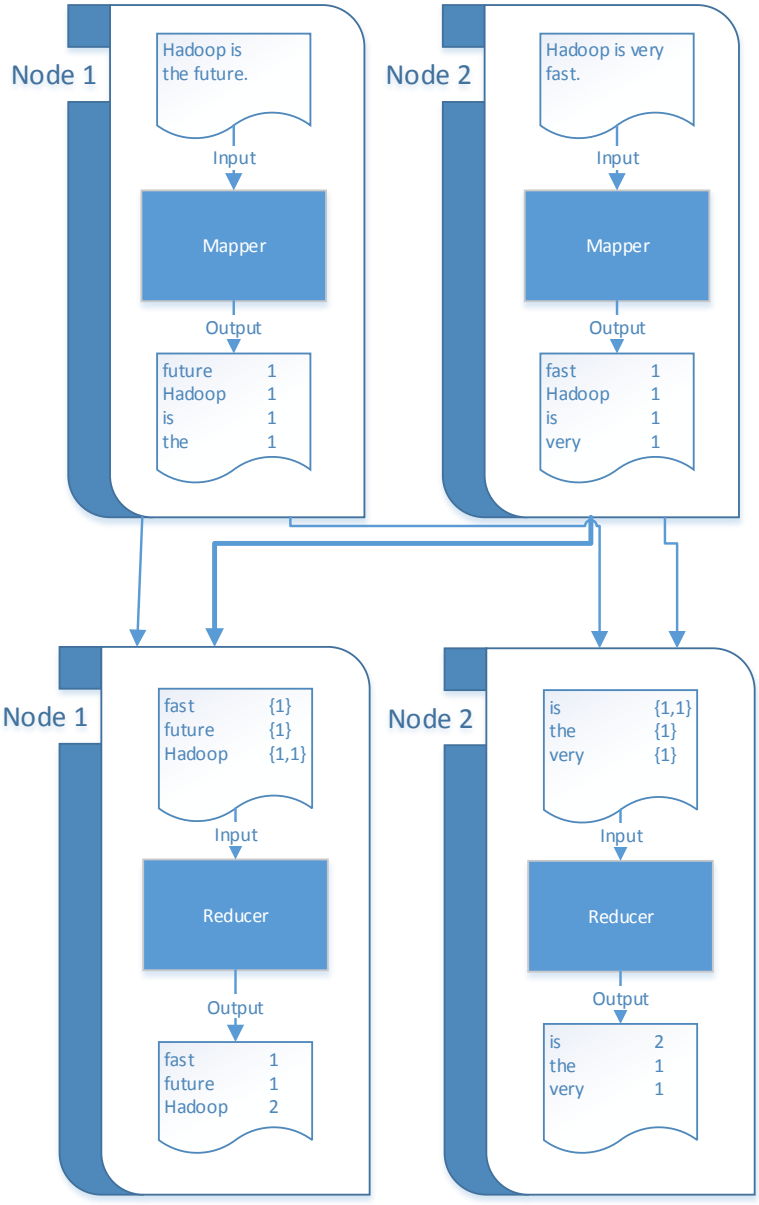


Figure 2.5.: MapReduce wordcount example

Some problems are too complex to be solved in a single MapReduce job. In this case, several jobs are chained to a sequence of jobs. Each job receives the resulting key-value pairs of its preceding job as input.

2.4.1. Secondary sort

Unlike the keys, which are sorted in the shuffle phase, the values belonging to a specific key are not ordered by Hadoop by default. Its order is determined by the order in which the map tasks finish [2]. If an application requires an order of the values, this can be imposed by implementing a *partitioner*, *group comparator* and *comparator* classes, which is known as secondary sorting. Figure 2.6 shows an example of secondary sort with weather data. The key has been transformed to a composite key consisting of the year and the temperature. A comparator for this class is used, which uses both parts of the composite key, i.e. year **and** temperature to sort the keys. The partitioning determines which values are sent to the same reducer and is controlled by the partitioner class. If both components are used for partitioning, some rows would be sent to a different reducer, as they may have the same year but different temperatures. Ergo, the partitioner for the composite key has to be set to only use the year for partitioning. Once the keys have arrived at the reducer, the group comparator defines which records belong to one call to the reduce method. The group comparator, like the partitioner, must only consider the year, else the records are split into too many groups. Once this is set correctly, all records sharing the same year are within the same partition and reduce group and are then sorted by the initially defined comparator (right side).

Further details and Java code implementation for the discussed classes can be found in [2].

2.4.2. Related projects

In order to make MapReduce job creation easier, several projects have been developed which build on Hadoop's MapReduce and HDFS. Based on the principles of Google Bigtable[15], *Apache HBase* is a non-relational open-source distributed database running on top of HDFS. Data stored in HBase is accessible through Java or REST Api or can serve as input to MapReduce jobs. *Apache Pig*[3] allows the

[3]https://pig.apache.org/

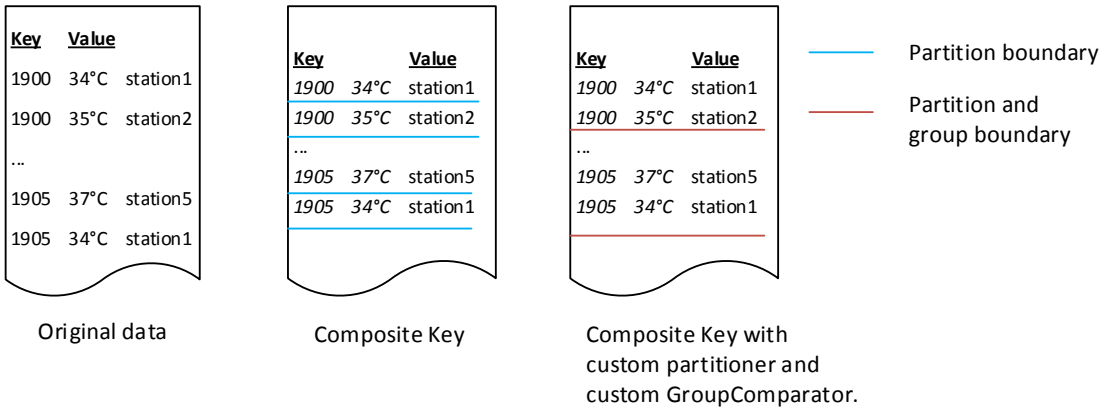


Figure 2.6.: Example of secondary sort. (Extended example from: [2])

user to describe data transformations in its own textual language called Pig Latin which are compiled and run as MapReduce jobs. The queries are automatically translated into a sequence of MapReduce jobs which can also use custom map and reduce methods. Many common operations, such as joins are already implemented efficiently and need not be implemented by the user. Similarly, *Apache Hive*[4] is a data warehouse software which allows the running of queries in a SQL-like language called HiveQL on data stored in HDFS. It also relies on MapReduce to execute the queries by translating the HiveQL queries into a sequence of MapReduce jobs. Its other features include: metadata storage in a relational database (Hive Metastore), indexing and various data sources and formats from the Hadoop Ecosystem, such as HBase. Like Hive, Cloudera Impala allows the user to run SQL statements against data stored on the HDFS, however, it does not rely on MapReduce. Cloudera Impala is described in more detail in the following chapter.

[4]http://hive.apache.org/

3. Cloudera Impala

Cloudera Impala[1] is an open-source software project written in C++ developed to perform interactive analysis of large data stored in Hadoop Distributed File System (HDFS) [16]. It provides a Massively Parallel Processing (MPP) engine for in-memory processing of unexpected queries written in SQL. Therefore, it can enable data analysts, who are familiar with SQL, to work on big data stored in HDFS without having to learn *MapReduce*, *Pig* or other technologies from the Hadoop and Big Data ecosystem. Based on Cloudera's Distribution for Hadoop (CDH), it also benefits from Hadoop's key features, such as data redundancy, authentication and security features [3]. As Impala supports all Hadoop data input formats, data can be processed directly on the HDFS without any middleware components. Unlike MapReduce and other software projects based on MapReduce, Impala is designed for interactive query evaluation. As described in subsection 2.4.2, Apache Hive is just such a data warehouse application based on MapReduce. Impala was designed to be fully compatible to Hive and even uses the Hive Metastore for storing meta information on the stored tables. This way, the stored tables can be queried by Impala and Hive without conversion of the data. However, the evaluation showed that HiveQL and the SQL dialect used by Impala are not completely compatible. Furthermore, there were problems with Hive reading Parquet files created by Impala (cf. chapter 6 for further details).

3.1. Core components

The Impala daemon, a local component on every data node, is used to process the data directly where it resides within the computer cluster. This avoids time-intensive transfer of data. SQL queries can be submitted to Impala via JDBC, ODBC, Apache Hue or the impala-shell. Each impala daemon in the cluster can

[1] http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html

accept a SQL query and distributes the computation work to other Impala daemons in the cluster. Each node returns its partial results back to the coordinator node, the node which initially accepted the query [3]. Based on the partial results, the coordinator node calculates the final result and returns this back to the user.

The second key component is the Impala statestore. Its process, *statestored*, runs on a single node in the cluster and observes the health state of all Impala daemons. The daemons can use this information when distributing work to other daemons. So, in an event of failure, the distributing daemons no longer consider the failed nodes for their workloads. Should the Impala statestore become unavailable, the daemons continue work as normal, however without knowing whether a specific daemon is available. This results in a less robust system [3] but does not necessarily cause Impala to fail. The third component is the Impala catalog service, which has a great impact on the performance of query evaluation. It was introduced to Impala in version 1.2.2. The catalog service automatically distributes metadata information on the tables stored in Impala to all nodes in the cluster. Whenever a table is created, deleted, altered, updated or data is inserted into a table, the catalog service demon, *catalogd*, broadcasts this information to all other nodes. Hence, manual updates of metadata, such as *REFRESH* – necessary in Impala versions prior to 1.2.2 – are no longer required. The metadata plays an important role for optimizing memory-intensive queries such as joins (cf. section 3.4) [17]. The catalog service is represented by a single process, catalogd, running on one node in the cluster. Figure 3.1 visualizes the core components of the Impala architecture.

3.2. Query language elements

One of the development goals of Impala is to allow data analysts to work with familiar tools, i.e. *SQL*, on large-scale data stored in HDFS. It therefore implements all language elements of the SQL-92 standard and partially elements from later standards SQL-99 and SQL-2003. This introduction can only give an overview of the SQL operators supported by Impala. It will therefore concentrate on those operators used in the translation of SPARQL queries. The SQL dialect of Impala is compatible to HiveQL, which allowed running the same queries with minor modifications in Hive and Impala during the evaluation (cf. chapter 6).

- Impala supports **SELECT**-queries with support for subqueries which can be

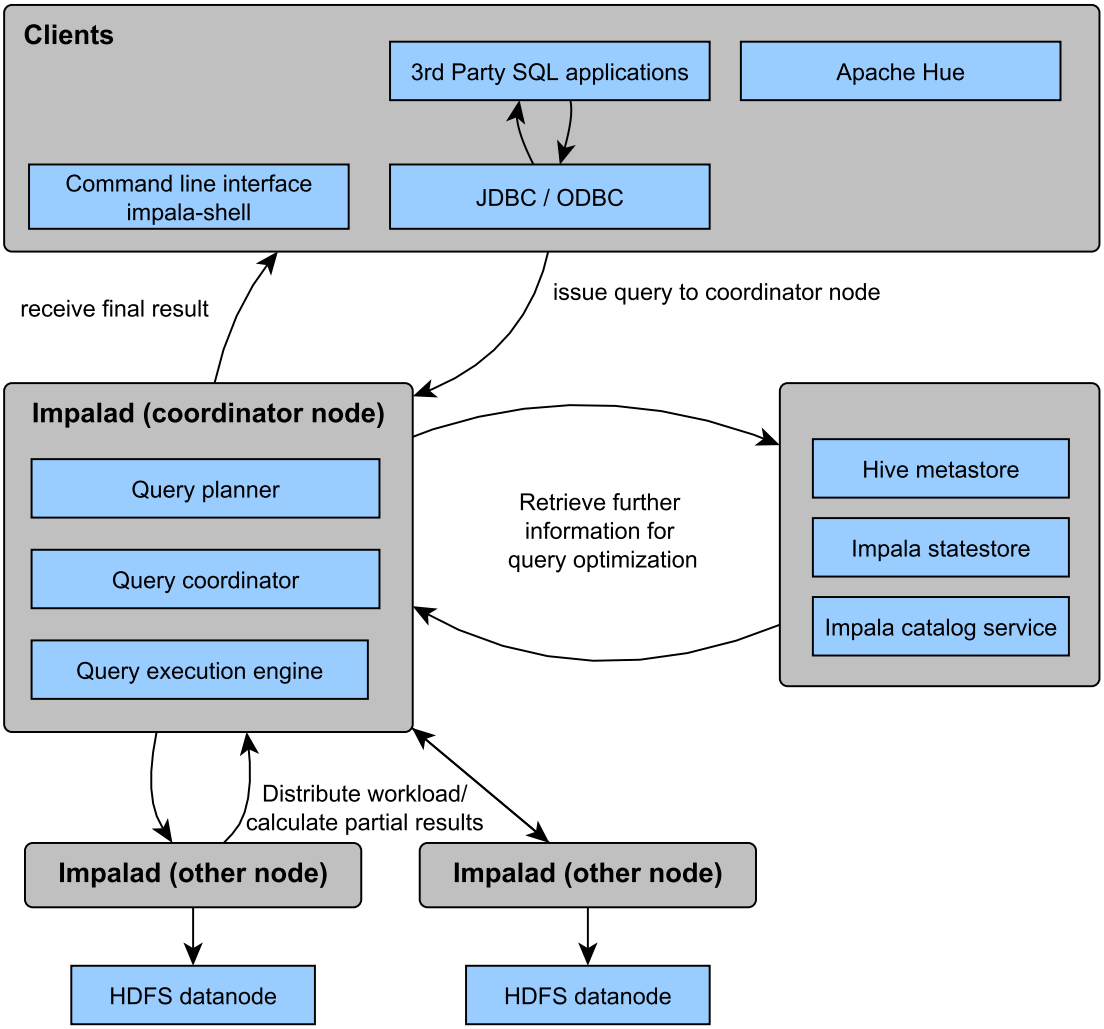


Figure 3.1.: Impala core components. Based on [3] but updated for Impala versions 1.2.2. and higher.

referenced like tables through aliases.

- The **ORDER BY** clause of a SELECT statement can be used to order the output records by given columns. As the data output can be very large and a sort operation may have a severe impact on performance, the *ORDER BY* clause can only be used with a *LIMIT* clause.
- **LIMIT** and **OFFSET** can be used to reduce the amount of output records to a certain size.
- **JOIN** can be used with *ON* clause to join tables on columns. If, however, a cartesian product is to be selected, the **CROSS JOIN** statement has to be

used explicitly.

- The **INSERT** command can be used to load records into tables. It is not recommended to use the *VALUES* clause when loading bulk data. Instead, values can be inserted by selecting them from external tables, which have been loaded from separated files in HDFS.
- **NULL values** are possible as column values and are different from empty strings. The token \N is used for null values in delimited files.

3.3. Data formats and compression types

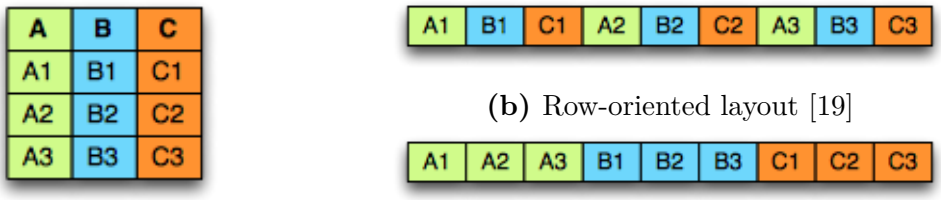
Every database in Impala represents a folder in HDFS. Existing files can be loaded into Impala as external tables. In consequence, the data is not moved and resides in the original location in HDFS. However, Impala can also create new tables, which are stored by default under the folder */user/hive/warehouse/*. The default file format is plain text, which generates a csv-file per table in HDFS. Other supported Hadoop file formats are: Avro, RCFile, SequenceFile and Parquet. Not only, the user can set the data format but can also choose from different compression algorithms, such as LZO, GZIP, Snappy, BZIP2 and deflate [3]. With Parquet specifically recommended by Cloudera for Impala, Parquet with Snappy compression was used for *Sempala*. In section 4.4, the performance of Parquet with Snappy compression is compared with tables stored in plain text format and Parquet format with GZIP compression.

3.3.1. Parquet data format

Initially developed by Twitter and Cloudera, Parquet[2] is a columnar file format inspired by Google's protocol buffers [18] for all projects of the Hadoop ecosystem, including Impala. Its column-orientated storage layout of data allows column-specific access to data, which enables performance benefits for Impala on warehouse-like queries [3] [17] [19]. This is achieved by keeping I/O to a minimum when selecting only specific columns. Figure 3.2 shows an example table stored in row-oriented layout (Figure 3.2b) and column-oriented layout (Figure 3.2c). Traditional RDBMs store data in a row-oriented layout, i.e. storing each row one after another. In

[2]http://parquet.io/

columnar file formats, the values for each column are stored together as a group. This way, when a query only selects specific columns, these columns can be read directly as opposed to row-oriented databases, where the entire row has to be read in order to access the column value. However, row-oriented databases perform better than column-oriented databases if the query selects most of the columns of the row, as column-oriented databases have an overhead when reconstructing an entire row. Also, if the row does not have many columns, the row can be read completely in one single disk access. In this case, row-oriented layouts do not have any overhead, even if only a few columns are selected. Column-oriented databases are therefore optimal for queries which aggregate few columns in wide tables. As columnar file formats store column values of a column together, homogeneous data blocks are created. This allows better performance of encoding and compression techniques. Ergo, the choice of database depends on the frequently used queries and the overall table size.



(a) Example table [19]

(b) Row-oriented layout [19]

(c) Column-oriented layout [19]

Figure 3.2.: Example table in different storage layouts.

Impala does not store a separate file per column. Instead, the rows of data are stored in files of fixed size (typically one gigabyte). Within these files, the data is stored in columnar layout. When Impala retrieves values for a specific column, all data files are opened but only those parts are read where the specified column can be found, which results in much less I/O.

The Parquet format supports nested data structures. An algorithm similar to the algorithm used by Google Dremel [18] flattens these data structures into columnar storage layout. This feature, however, is not yet supported by Impala 1.2.3.

Two automatic compression techniques further minimize the I/O: dictionary encoding and run-length-encoding (RLE). With dictionary encoding, Impala stores 2-byte large identifier values instead of the actual values. This is applied to each column which contains 2^{16} or fewer distinct values. RLE searches for repeating values in a sequence of rows and encodes these patterns efficiently. After the application of

the automatic compression techniques, user-specified compression, e.g. *Snappy*, is applied.

3.4. Performance optimization

In version 1.2.3, Cloudera Impala offers several possibilities to optimize queries, which is essential for memory-intensive queries, e.g. join operations:

- **Statistics** Impala offers two types of statistics: table statistics and column statistics. The former contains the number of rows, number of files, size and data format, while the latter stores type, number of distinct and null values and the maximum and average size of each column. As of version 1.2.2, Impala allows users to compute these statistics for a table and each of its columns with one single command *COMPUTE STATS*. The gathered meta information is used to decide how to load the data and distribute work of a query over the network.

- **Join reordering** Automatic join order optimization was introduced in version 1.2.2. The order of the joined tables is rearranged based on file and table sizes. If a manual ordering is desired, the user can specify this with the keyword *STRAIGHT JOIN*. The official documentation recommends loading the largest table first and join the remaining tables in ascending order of table sizes.

  Impala's default join strategy is the broadcast join where the right table is copied to each node in the cluster. This is assuming the right table is smaller than the left table. However, it also supports another approach, the partitioned join. With this strategy, only subsets of rows of the table are sent to specific other nodes involved in the query, allowing parallel processing of these rows on the respective nodes. This strategy is meant to provide better performance for large tables of approximately equal size. With statistics available, Impala chooses the appropriate strategy for each join. If statistics are unavailable for a table, the table is placed on the right side of the join and broadcast join strategy is used. The chosen join order and join strategies can be revealed with the *EXPLAIN* statement.

- **Partitioning** Each Impala table is stored in a single directory on the HDFS by default. The partitioning feature physically divides the data into subfolders

by either defining data ranges or specific values for a column. For example, a table containing data with a column *country*, can be partitioned by its values into subfolders for each country. Therefore, a query which only works on a specific country only has to load the data within the subfolder, which can result in a significant performance benefit. Other commonly used examples would be to partition by year or language, depending on the data. Table partitioning can be enabled by the keyword *PARTITIONED BY* followed by column names and types with a *CREATE* or *ALTER TABLE* statement. In the documentation, files sizes under 1GB are not recommended and the partitioning columns should be set accordingly in order to avoid too small partitions. The *PROFILE* statement gives detailed information on how a query was evaluated and the workload processed. It also states how many partitions were read for the query.

3.5. Current limitations in Impala 1.2.3

There are, however, several differences and limitations in Impala (version 1.2.3) compared with other relational database systems [3]:

- DDL statements, such as *DELETE*, *UPDATE* or *ROLLBACK* are not supported. The only supported DML (data manipulation language) statements are *INSERT* statements and CREATE, ALTER or DELETE of databases and tables.
- Constraints, such as primary and foreign key relations, are not available, as they create too much overhead for big data environments. For similar reasons triggers have not been implemented.
- Impala does not rely on column indexes but instead uses table and column statistics (see below) for query optimization.
- Impala supports the following column types: BOOLEAN, SMALLINT, INT, BIGINT, DOUBLE, FLOAT and STRING (instead of VARCHAR or VARCHAR2). Non-scalar data types, for example lists or maps, are not supported yet, although they can already be used in Hive and its corresponding language HiveQL. This feature is planned to be implemented for Impala version 2.0 in combination with the support of nested data types in Parquet, as described in subsection 3.3.1.

- Joins are limited to in-memory joins, which requires both tables which should be joined to fit into the memory of the cluster node performing the join. Unlike MapReduce-based systems, increasing the number of nodes in the cluster does not solve this problem, as it depends on the physical memory available to the node performing the join. Alternative software systems similar to Impala are presented in section 7.2. They overcome the limitation by using either in-memory or on-disk joins, depending on the input table sizes. On-disk joins are planned for Impala 2.0.

4. Storage schemas for RDF data in Impala

As introduced in section 2.1, RDF data is stored in RDF stores, which are essential for the semantic web. There are different types of RDF stores [20]:

- **Native stores** – These stores use a custom binary format for storing RDF data. Examples are 4store[21] or Jena TDB[22].
- **Relationally-backed stores** – The RDF data is stored in traditional relational databases. As opposed to native stores, relationally-backed databases offer advanced features, e.g. industrial-strength transaction support, security and authentication mechanisms. Examples are Jena SDB [22] or cStore [23].
- **Stores based on NoSQL-databases** – Latest developments include RDF stores which rely on either NoSQL (not only SQL) databases, such as HBase [24], or other big data technologies, for example Hadoop [25].

Sempala is a hybrid of the two latter types, as it is based on Impala. On the one hand Impala supports SQL and works with tables, on the other hand it is based on Apache Hadoop, and therefore shares features with NoSQL-based RDF stores.

Since the beginning of the Semantic Web, the possibility of storing RDF data in classical relational databases has been explored [23][26]. However, the distributed design of Impala and its column-orientated data format Parquet require a fresh evaluation of these storage strategies, which were mostly developed for single-place installations with row-oriented storage formats. Before the final implementation of Sempala, three different storage schema strategies for storing RDF data were implemented as prototypes. Each strategy was tested with different data formats and compression types and evaluated using nine predefined queries on data generated by the SP^{2}Bench [27] benchmark. The choice of queries is described in section 4.4. The three strategies are presented in section 4.1 to section 4.3, while the prototype evaluation results can be found in section 4.4.

subject	predicate	object
per:Paul_Erdoes	rdf:type	foaf:Person
per:Paul_Erdoes	foaf:name	"Paul Erdoes"^^xsd:string
per:Amanda_Knight	rdf:type	foaf:Person
per:Amanda_Knight	foaf:name	"Amanda Knight"^^xsd:string
publications:Article4	rdf:type	dc:Document
...	...	...

Table 4.1.: Example RDF data stored in triple-table relation.

4.1. Triple table

The triple table storage strategy requires only one relation to store an entire RDF document [20]. Each RDF statement, consisting of subject, predicate and object, is stored in a separate row of the table, resulting in a three-column table. Advantages of this approach are its simplistic design and its capability of easily adding further statements, as they can simply be appended to the table. However, the data is not partitioned which results in self-joins of a potentially very large table, even for simple SPARQL queries. All columns have the same type (usually *String*). This does not allow the use of literal type information and as a consequence results in overhead.

Table 4.1 shows an excerpt from the example RDF graph in Listing 2.1 on page 16 stored in triple-table relation.

4.2. Vertical partitioning

The goal of the vertical partitioned storage scheme [23] is to avoid loading the entire RDF document when a query only requires a subset of this data. In order to achieve this, the set of statements is divided into subsets which share the same predicate. Each subset results in a relation with two columns, subject and object, whereas the shared predicate is the table name. Each triple in the basic graph pattern of a query, would result in a SELECT query on the table identified by the predicate. Thus, it is possible to load only the subset of the RDF document relevant to the query and

ignore the rest of the graph. This optimization works especially well for structured RDF data [23]. Table 4.4 shows two partial tables for the RDF dataset used for the example in Table 4.1.

subject	object
per:Paul_Erdoes	foaf:Person
<http://localhost/Article3>	dc:Document
per:Amanda_Knight	foaf:Person
...	...

Table 4.2.: rdf:type table

subject	object
per:Amanda_Knight	"Amanda Knight"^^xsd:string
per:Paul_Erdoes	"Paul Erdoes"^^xsd:string
...	...
...	...

Table 4.3.: foaf:name table

Table 4.4.: Example RDF data stored in tables following vertical partitioning scheme.

In contrast to the triple table approach, the vertical partitioning schema allows type information for the object column, as this column could have a different type for each table. However, if the predicate in a SPARQL query triple is not bound, the vertical partitioning schema requires many joins. In this case, all tables must be joined to evaluate the query, as there is one table per predicate and all predicates need to be considered. Depending on the number of distinct predicates in an RDF document, vertical partitioning can lead to many tables of small file size. This can have a negative impact on performance as Impala is optimized for working with large files.

4.3. Property table

The Parquet data format, introduced in subsection 3.3.1, is optimized for wide tables, meaning tables with many columns, as its columnar storage format allows direct access to the values of a column without having to read the entire row. Both previous storage strategies are designed for row-oriented relational databases, e.g. Oracle, MySQL or PostgreSQL. As a consequence, they construct narrow tables with two or three columns and consequently many rows. The last approach, called property table storage schema, aims at creating a wide table. The table has a row for every resource which occurs at least once as a subject in a triple. Each row

ID	rdf:type	foaf:name	dc:creator	dc:title
per:Paul_Erdoes	foaf:Person	"Paul Erdoes"^^xsd:string	<http://localhost/Article3>	\N
per:Amanda_Knight	foaf:Person	"Amanda Knight"^^xsd:string	<http://localhost/Article3>	"Example"^^xsd:string
per:Amanda_Knight	foaf:Person	"Amanda Knight"^^xsd:string	<http://localhost/Article4	\N
<http://localhost/Article3>	dc:Document	\N	\N	\N
<http://localhost/Article4>	dc:Document	\N	\N	\N

Table 4.5.: Example RDF data stored in property table relation.

contains a column "ID" containing the URI identifying the resource. Additionally, it contains a column for each predicate (property) in the document containing the corresponding object value.

The schema is based on schemata presented in [26] but has been adapted to a single table. In [26], multi-value-properties are stored in separate tables, next to a table for single-value properties and a triple store. Instead, a single table is used with multiple rows for a specific resource, if multi-value-properties are contained in the RDF data for this resource. Another possible solution would be to store a list of values for multi-value properties. This would enable having one row per resource in the property table. However, as described in section 3.5, Impala does not support nested data structures[1] as column values. Multi-value properties are therefore stored as follows: For each value of a given multi-value predicate, a duplicate of the row is saved containing all other column values. Table 4.5 shows the property table for the RDF graph in Listing 2.1 on page 16. The property *dc:creator* has multiple values for the resource *per:Amanda_Knight*, namely `<http://localhost/Article3>` and `<http://localhost/Article4>`. Ergo, two rows are required for this resource with identical values for all other columns apart from *dc:creator*. If a second column were to have more than one value for this resource, the cross product of both these columns would be calculated, resulting in additional rows. Null values ($\backslash N$) are saved for those predicates where a value does not exist for the given resource.

The property table schema is especially effective for queries scanning for multiple properties of a single specific subject, so-called star-queries. With the other two storage strategies, these queries result in many joins. However, all possible predicates of an RDF document have to be known before creating the property table- This presents a loss of flexibility compared with a triple table. As multi-value predicates lead to duplicate values in a column. For example, in Table 4.5, Amanda_Knight

[1]This feature is planned to be implemented in Impala 2.0.

occurs twice in the ID column. As a consequence, each query against the property table includes the keyword *DISTINCT*, in order to avoid these duplicates. This way, queries which yield duplicates as result cannot be processed correctly. Only the distinct values are returned in this case.

Due to the null values and additional rows for multi-value properties, this schema creates a lot of overhead, potentially leading to larger tables and performance issues. As this schema was developed with the Parquet data format in mind, the overhead is minimized with the automatic compression techniques enabled. While null values only have a minimal effect on file size in Parquet, the dictionary and run-length encoding with Snappy compression handles the overhead of saving duplicate values for multi-value predicates. Table 4.7 in section 4.4 shows the non-existent overhead impact on file size is for the tested dataset.

4.4. Prototype evaluation

Nine SPARQL queries were defined to test the general performance of the three storage strategies. They are based on RDF data generated by the SP^{2}Bench SPARQL benchmark[27]. Join evaluation has the strongest impact on the complexity of a query. Ergo, the queries are designed to test different types of join patterns and thus uncover which type of queries perform especially well or perform badly. Listing A.1 through Listing A.8(cf. Appendix) show the queries written in SPARQL. Q1 through Q3 consist of only one triple pattern varying the variable position within the pattern. These will be referenced as look-up queries in the following. Q4, Q7 and Q8 consists of triple patterns sharing the same subject. This type of query is called star-query. Queries Q5 and Q6 each have a different variable in subject position and share exactly one variable with the previous triple pattern. In terms of RDF graphs, these queries generate long paths similar to a depth-first-search and are called linear queries. The final query Q9 is the only query which has a variable in predicate position.

The tables are loaded into Impala as external tables created from tsv-files (tab-separated-values). From there, this external table is copied into internal tables with the data formats and compression types that are to be tested, i.e. plain text and Parquet file format with Snappy and GZIP compression, respectively. This process was automated using scripts written in Python using *impala-shell* to interact with

the Impala database. The tsv-files needed for the initial external table were created by a MapReduce program which takes an RDF graph in N3-format as input and generates the tsv-files onto the HDFS. The tests were performed on a computer cluster described in greater detail in section 6.1.

As the queries consisted near-exclusively of basic graph patterns, the translated SQL queries consist of a sequence of join operations. The queries were transmitted to Impala via the *impala-shell* interface.

4.4.1. Prototype evaluation results

Data format comparison

Running Impala version 1.2.1, all three storage strategies were tested on a dataset of 100 million triples, which resulted in the runtimes in Table 4.6. The data format and compression only has minimal impact on the runtimes, although GZIP compression seems to slow down the system and Parquet with Snappy compression runs slightly faster for some queries. However, the impact on storage capacity is significant, as depicted in Table 4.7. The orginal RDF document has a size of 10.5 gigabyte. In plain text format, the overhead of the property table strategy becomes visible, resulting in 14.2 gigabyte. It profits most from the Parquet file format, which compresses the data to about 20% of its original size. As GZIP compression has a negative influence on performance, it was therefore decided to use Parquet with Snappy compression for the final implementation.

Comparison of all storage strategies

All three storage schemas were compared on two different dataset sizes to get a notion of how the storage strategies perform. Table 4.8 and Table 4.9 show the results of the nine queries running Impala version 1.2.1 for datasets with 50 million and 100 million triples, respectively. This version did not yet support automatic join optimization, as this feature was introduced in Impala version 1.2.2.

The triple table approach only yielded better runtimes for query Q9 and even failed to run query Q8 on the 100 million triple dataset. Query Q9 was expected to perform better with the triple table strategy, as it can be answered by a single scan of all rows in the triple table, whereas the other strategies require joins (vertical

		Vertical Partitioning		**Triple table**		**Property table**		
		Text	Parquet	Text	Parquet	Text	Parquet	Parquet
query	results		(Snappy)		(Snappy)		(Snappy)	(GZIP)
Q1	1	2.06s	1.13s	10.99s	3.71s	1.41s	1.57s	2.22s
Q2	1	0.43s	0.55s	0.75s	2.16s	2.77s	1.00s	2.14s
Q3	1589621	16.55s	17.25s	17.11s	17.18s	16.73s	16.68s	17.34s
Q4	1	0.73s	1.36s	1.32s	4.20s	1.15s	1.08s	2.34s
Q5	151300	19.54s	16.73s	37.55s	39.34s	78.34s	73.66s	79.33s
Q6	2	15.37s	14.15s	34.19s	35.35s	74.87s	75.50s	76.00s
Q7	819	39.95s	37.13s	40.80s	42.17s	2.11s	2.00s	3.80s
Q8	9097526	533.18s	535.28s	MEM	MEM	317.26s	315.45s	313.77s
Q9	656	18.02s	9.86s	0.76s	2.15s	23.37s	23.95s	27.74s

Table 4.6.: Runtimes of queries Q1 through Q9 on SP^{2}Bench dataset with 100 million triples. Different data formats and compressions were tested. *MEM* indicates a memory error during execution. (Impala version 1.2.1)

	RDF document	**Triple table**	**Property Table**	**Vertical Partitioning**
Text (uncompressed)	10.5 G	9.7G	14.2G	8.6G
Parquet (Snappy)	-	2.0G	1.8 G	2.3 GB
Parquet (GZIP)	-	1.2G	1.1 G	-

Table 4.7.: Table sizes of 100 million triple SP^{2}Bench dataset. (Impala version 1.2.1)

query	results	**Vertical Partitioning**	**Triple table**	**Property table**
Q1	1	0.90s	5.74s	1.31s
Q2	1	0.55s	0.56s	0.72s
Q3	1003810	10.90s	10.95s	10.72s
Q4	1	0.95s	0.85s	0.85s
Q5	73594	8.43s	17.85s	32.67s
Q6	2	6.90s	16.66s	32.52s
Q7	498	15.89s	20.96s	1.54s
Q8	4230949	247.61s	332.57s	150.19s
Q9	656	7.44s	0.58s	14.71s

Table 4.8.: Runtimes of queries Q1 through Q9 on SP^{2}Bench dataset with 50 million triples. Property table and vertical partitioning was used with Parquet format and Snappy compression. Triple table is stored as plain text. (Impala version 1.2.1)

partitioning) or unions (property table). Property table and vertical partitioning performed quite similar for queries Q1-Q4. However, vertical partitioning performs much better for chain-queries Q5 and Q6, while the property strategy has shorter runtimes for star-queries Q8 and Q9. The evaluation of star-queries does not require any join operations with the property table approach, which results in a significant performance advantage.

With the introduction of automatic join optimization in version 1.2.2 of Impala, the queries Q5-Q8 were rerun on the datasets consisting of 100 and 200 million triples. Table 4.10 shows the results running Impala version 1.2.3 and data format set to Parquet with Snappy for all three storage strategies.

The vertical partitioning and property table strategy both benefit from the join optimization compared with the results running Impala version 1.2.1. Comparing the property table and vertical partitioning strategy, the property table profits more from the optimization and no longer has significantly higher runtimes for chained queries Q5 and Q6 compared with vertical partitioning. It yields the best runtimes for the star queries Q7 and Q8 - even more evident in the 200 million triples dataset,

query	results	**Vertical Partitioning**	**Triple table**	**Property table**
Q1	1	1.14s	10.99s	1.66s
Q2	1	0.55s	0.75s	1.00s
Q3	1589621	16.57s	17.11s	16.58s
Q4	1	1.24s	1.32s	1.08s
Q5	151300	18.88s	37.55s	68.03s
Q6	2	15.74s	34.19s	75.64s
Q7	819	42.25s	40.80s	2.08s
Q8	9097526	535.08s	MEM ERR	315.77s
Q9	656	9.28s	0.76s	24.04s

Table 4.9.: Runtimes of queries Q1 through Q9 on SP^{2}Bench dataset with 100 million triples. Property table and vertical partitioning was used with Parquet format and Snappy compression. Triple table is stored as plain text. (Impala version 1.2.1)

query	results		**Vertical Partitioning**		**Triple table**	**Property table**	
	100M	200M	100M	200M	100M	100M	200M
Q5	151300	272039	8.91s	15.27s	41.90s	10.09s	15.75s
Q6	2	2	8.41s	14.24s	38.62s	7.63s	11.16s
Q7	819	1485	12.06s	18.98s	45.09s	1.76s	1.93s
Q8	9097526	18763170	467.29s	MEM ERR	MEM ERR	283.16s	574.63s

Table 4.10.: Runtimes for queries Q5 - Q8 with automatic join optimization (Impala 1.2.3) on SP^{2}Bench datasets with with 100M and 200M triples

where vertical partitioning strategy lead to memory errors.

With the introduced join optimization, the property table strategy did not show any significant shortcomings compared with the vertical partitioning strategy, other than queries which included unbound predicates, e.g. query Q9. It was therefore decided to use a combination of the property table and triple table strategy, whereas the latter would only be used in the case of an unbound predicate in a SPARQL query. Overall, the Parquet data format with Snappy compression in combination with the property table schema showed the best performance in the initial prototype evaluation and was consequently used in the final implementation of Sempala.

5. Implementation

Figure 5.1 gives an overview of the implementation architecture of Sempala. The software implementation consists of two main components. First, the loader component loads input RDF document located in HDFS into Impala as a property table and a triple table. MapReduce was used to transform the RDF document into a table enabling Impala to read it. Secondly, the SPARQL2SQL component translates SPARQL queries into the equivalent SQL queries, which operate on the generated Impala tables. Due to the property table design, discussed in section 4.3 on page 41, the SQL queries can only yield distinct results. This may differ from the actual results of the SPARQL query if duplicates are present in the dataset. The SQL queries can be passed as input to the Impala shell.

5.1. Loader

The loader consists of a driver implemented in Python, which handles the MapReduce job execution and the creation of the Impala tables via *impala-shell*, while the MapReduce jobs themselves were implemented in Java. Two MapReduce jobs are required to convert a given RDF document to a property table in tsv (tab-separated values) file format. As all properties have to be known at the time of the property table creation, the first job scans over the entire dataset and writes a list of distinct predicates and their types (in case of literal objects) as a file in HDFS. The pseudocode of this job can be found in Listing 5.1. In the map phase, the predicate name and the type of the literal in object position (if available) are emitted as map output key and map output value, respectively (lines 6 and 8). For example, the type of `"Amanda Knight"^^xsd:string` is `xsd:string`. In the reduce phase, the reduce input key and first value are emitted as output key and output value. This assumes that all values are of the same type as the first value, which is not verified by the implementation. If not true, the following creation of the table would fail.

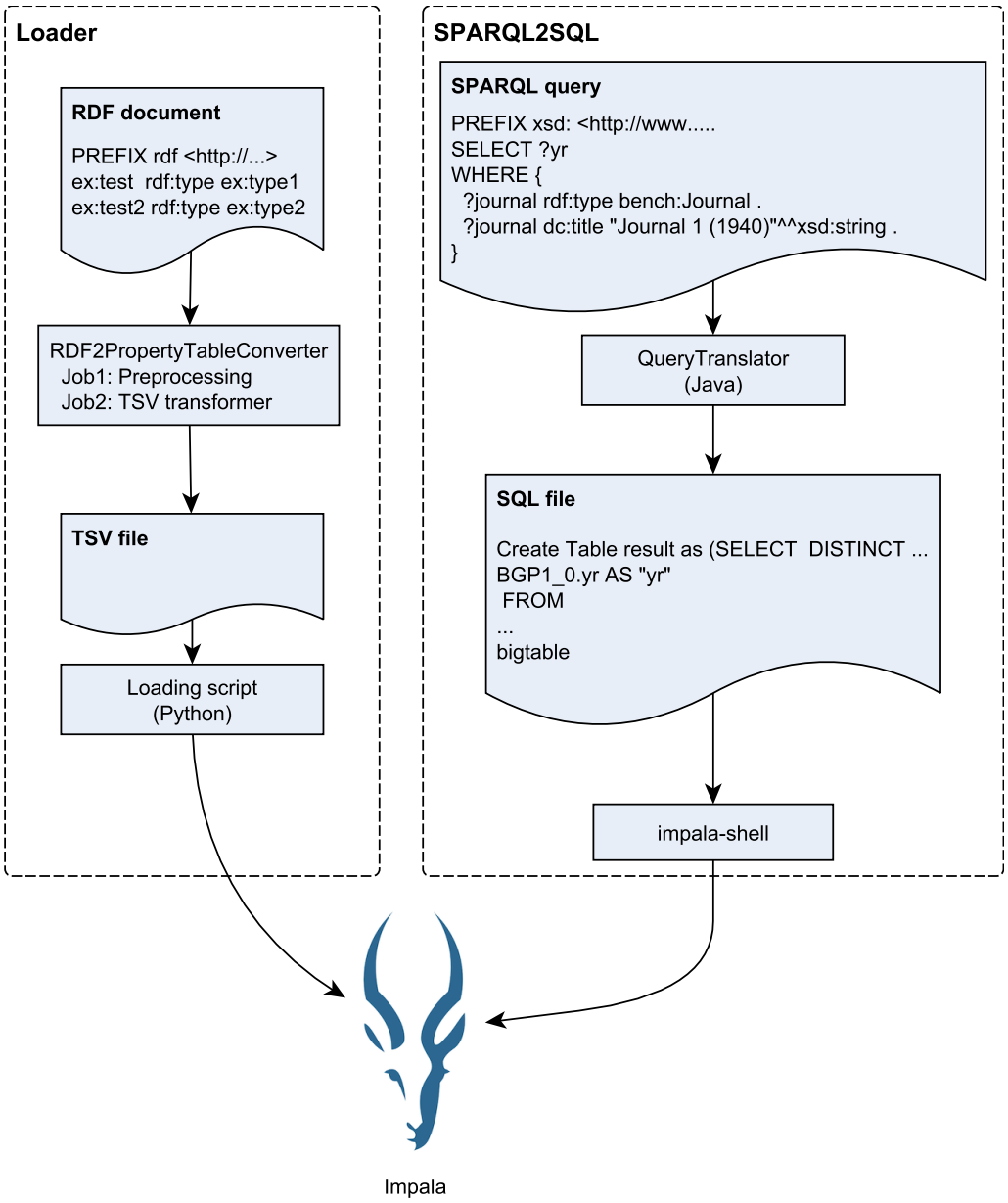


Figure 5.1.: Architecture of Sempala

```
1  void map(key, value){
2          triple = TripleParser.parseline(value);
3          predicate = triple.getPredicate();
4          object = triple.getObject();
5          if (object.isTypedLiteral()) {
6                  emit(predicate, object.getLiteralType());
7          } else {
8                  emit(predicate, "xsd:string");
9          }
10 }
11 void reduce (Text key, List values){
12         emit(key, values[0]);
13 }
```

Listing 5.1: Preprocessing map and reduce functions

```
1  void map(key, value){
2    triple = TripleParser.parseline(value);
3    emit(new CompositeKey(triple.getSubject(), triple.getPredicate()),
4             new CompositeKey(triple.getPredicate(), triple.getObject()));
5  }
```

Listing 5.2: TSV transformer map function

```
1  void reduce(key, value){
2    List<List<Row>> groupedRows = new List<List<Row>>();
3    String prev = "";
4    List currentGroup = new List<Row>();
5    for (CompositeKey value : values) {
6      subject = key.getFirst();
7      property  = value.getFirst();
8      object = value.getSecond();
9          // predecessor is different from current value
10         if (!value.getFirst().equals(prev)) {
11                 groupedRows.add(currentList);
12                 currentGroup = new List<Row>();
13         }
14         list.add(new Row(property, object));
15         prev = value.getFirst();
16   }
17   // add last group
18   groupedRows.add(currentGroup);
19   // calculate cross product of row groups
20   writeCrossProductAsRows(groupedRows);
21 }
```

Listing 5.3: Reduce method of TSV transformer

After this preprocessing step, the actual conversion job, which implements a secondary sort (cf. subsection 2.4.1), receives the RDF document as input. In the map phase (Listing 5.2), each triple is parsed into subject, predicate and object and the composite keys *(subject, predicate)* and *(predicate, object)* are emitted as map output key and map output value respectively. As explained in subsection 2.4.1, the following shuffle phase groups the output values by the value of subject, while sorting them according to the value of predicate of this group.

Thus, in the reduce phase (Listing 5.3), each call to the reduce function receives the subject and all the properties (predicates) and property values belonging to it sorted by property. From here, all property values are grouped into lists of rows by property name, which can easily be achieved by exploiting the sorted order of the values (lines 4 to 18). As described in section 4.3 on 41, the values of multi-value properties have to be cross joined with the other property values, i.e. all other columns in the property table. Hence, the cross product of every row group is calculated (line 20).

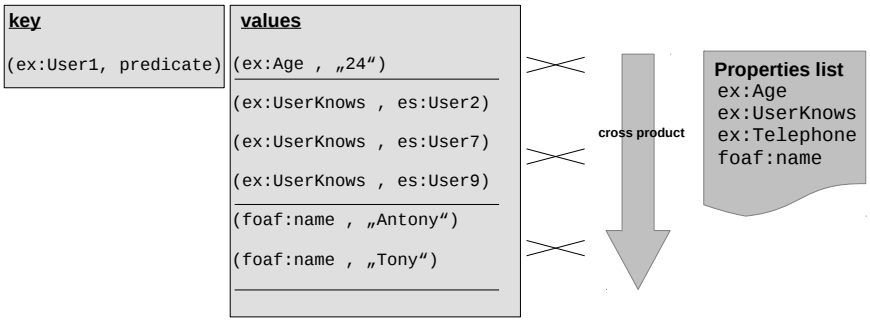
The columns of every row written by *writeCrossProductAsRows* have to follow a fixed order. At this point, the ordered list of distinct predicates from the preprocessing job is used. In the setup-phase of the reducer, the predicate list is loaded and is used to create the rows of the property table as depicted in Listing 5.4. For each property in the sorted list, it is checked whether a corresponding group exists. If not, the null value $\backslash N$ is appended to the resulting row. As a result, each emitted row in the tsv file follows the same order defined by the sorted list of predicates.

```
List knownPredicates = loadPreprocessingFromHDFS();
// used by writeCrossProductAsRows() method
String createRow(subject, row){
        String result = subject;
        // fixed order
        for (predicate : knownPredicates) {
          if (row.containsKey(predicate)) {
            result.append(row.getValue(predicate));
          } else {
            result.append("\\N");
          }
          result.append("\t");
        }
        return result;
}
```

Listing 5.4: createRow() method used in the Reducer for creating rows of the property table

Figure 5.2 shows the workflow of a reduce call with example values. In the example, the properties *ex:UserKnows* and *foaf:name* have multiple values. Other known properties are ex:Age and ex:Telephone. The reduce call generates all rows of the property table for the resource *ex:User1*, which is therefore the first part of the composite reduce input key. The second part of the composite key is *ex:Age*, which is only needed for the secondary sort and is the first of all map output keys. This entry is not used in the reduce function at all. To avoid confusion, the value *predicate* is used instead. The values are grouped into three groups by the first part of the composition, i.e. *ex:Age*, *ex:UserKnows* and *foaf:name*. As a consequence of the cross product calculation, the reduce call generates 6 property table rows. The resource *ex:User1* does not have a value for the property *ex:Telephone*, which results in the null value ($\backslash N$) in all rows for this column.

Once the tsv file has been generated in HDFS, the loader program reads the sorted list of predicates and their object literal types from the preprocessing job and derives the table schema from that. The table is loaded as external table into Impala from



ID	ex:Age	ex:UserKnows	ex:Telephone	foaf:name
ex:User1	„24“	es:User2	\NULL	„Antony“
ex:User1	„24“	es:User2	\NULL	„Tony“
ex:User1	„24“	es:User7	\NULL	„Antony“
ex:User1	„24“	es:User7	\NULL	„Tony“
ex:User1	„24“	es:User9	\NULL	„Antony“
ex:User1	„24“	es:User9	\NULL	„Tony“

Figure 5.2.: Reduce phase of RDF converter MapReduce job.

the generated tsv file (see example in Listing 5.5). As certain special characters cannot be used as column names in Impala, these characters have to be replaced in the property names. The external table is then copied and stored as internal table in Parquet format with Snappy compression enabled (lines 11 ff.). Finally, the table and column statistics are computed. As the triple table is required for queries where the predicate is unbound (cf. chapter 4), a map-only MapReduce job is run to create the tsv file, which is followed by the same steps for table creation.

```
1  CREATE EXTERNAL TABLE property_table_ext(
2    ID STRING,
3    ex_age INT,
4    ex_UserKnows STRING,
5    ex_Telephone STRING,
6    foaf_name STRING
7  )
8  ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
9  LOCATION "/data/test/";
10
11 CREATE TABLE property_table LIKE property_table_ext STORED AS PARQUETFILE;
12 SET PARQUET_COMPRESSION_CODEC=snappy;
13 insert overwrite table property_table SELECT * FROM property_table_ext;
14 COMPUTE STATS property_table;
```

Listing 5.5: SQL script used to load property table.

Sempala also supports partitioning the data by *rdf:type* property, if this property

is available. Hereby, the same generated tsv file can be used and parts of the SQL script are changed as shown in Listing 5.6. The example assumes an additional column for *rdf:type* in the property table.

```
CREATE TABLE property_table_partitioned(
  ID STRING,
  ex_age INT,
  ex_UserKnows STRING,
  ex_Telephone STRING,
  foaf_name STRING
)
PARTITIONED BY (rdf_type STRING)
STORED AS PARQUETFILE ; -- SNAPPY is default compression

INSERT INTO property_table_partitioned partition(rdf_type)
        SELECT ID, ex_age, ex_UserKnows, ex_Telephone, foaf_name,
        rdf_type  -- select type last
        FROM property_table_ext;
```

Listing 5.6: SQL script used to load external partitioned property table.

5.2. SPARQL translator

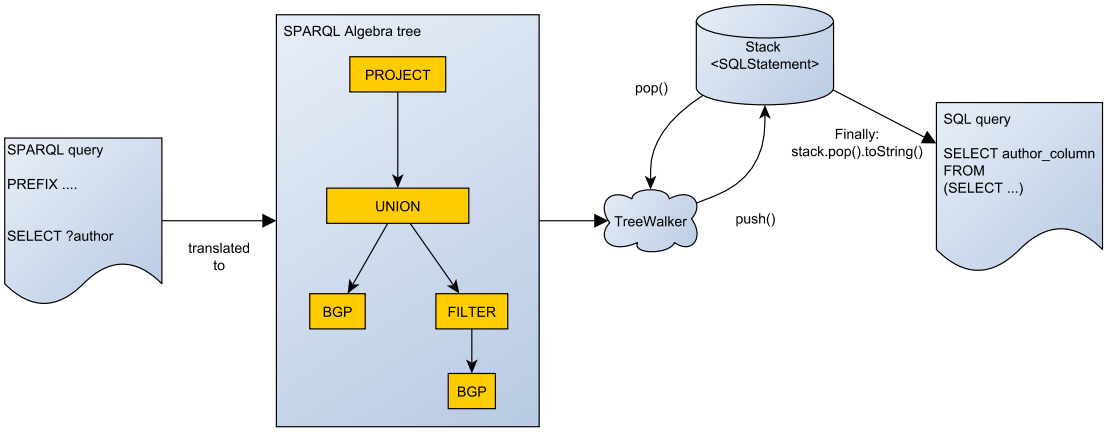


Figure 5.3.: Translator component

Sempala is based on parts of the implementation of *PigSPARQL* [28], whose code was provided by its author Alexander Schätzle. Like PigSPARQL, Sempala only supports SPARQL SELECT queries, which is the most common SPARQL query type. This is due to the fact that Impala is designed for write-once-read-many operations, which does not yet offer support for DML queries, such as *UPDATE*

queries on rows. Because SPARQL and SQL queries can be quite complex and deeply nested, the SPARQL queries are not translated directly into SQL but instead are transformed to an algebra tree, which consists of operators from the SPARQL algebra (introduced in subsection 2.3.6 on 21).

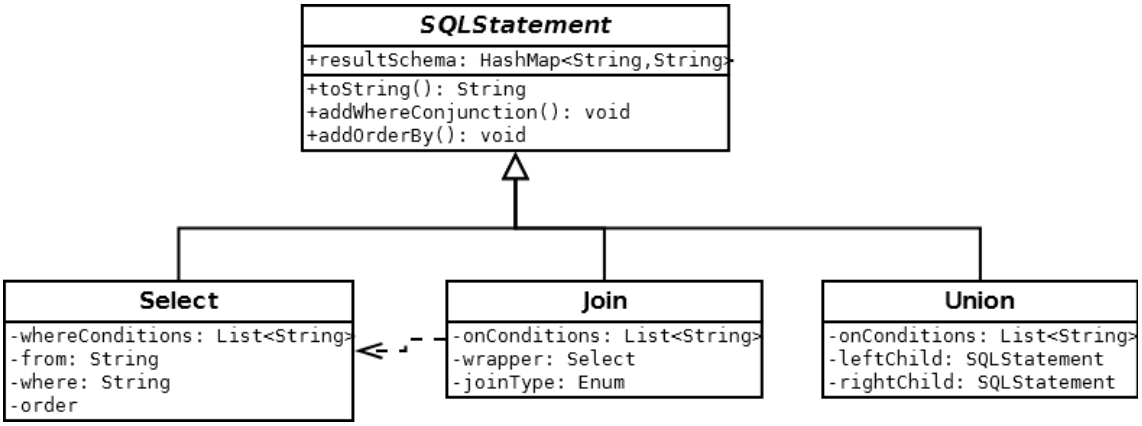


Figure 5.4.: UML diagramme of SQL statement classes.

This approach was adopted from PigSPARQL, as it also provides implemented optimization techniques on the algebra level and therefore can be applied before translating to PigLatin or SQL respectively. From there, the algebra tree is walked from the bottom up by a tree walker object. PigSPARQL translates the queries to Pig Latin, a procedural language, while Sempala translates to SQL, a declarative language. This is why Sempala cannot follow the pattern of PigSPARQL, which translates each algebra tree node into a string appended to the current PigLatin program. Instead, a stack-based approach is used by the translator module of Sempala. While walking the algebra tree from the bottom up, the algebra tree walker translates the nodes into objects representing the SQL operators: *SELECT*, *UNION* or *JOIN* (for all types of joins). These classes (illustrated in Figure 5.4) are all derived from the abstract class *SQLStatement*. For some algebra operators, such as FILTER, a new object is not created. Instead, the current SQLStatement object at the top of the stack is altered, e.g. by adding a *WHERE* clause. How each SPARQL algebra operators is treated by the walker is described in subsection 5.2.1.

When the walker visits a node, it therefore retrieves the latest translation of the subtree underneath the currently visited node from the stack. This translation is stored as an object of the class *SQLStatement* on the stack. With the information stored in the visited node and the retrieved SQLStatement object, the currently visited node can be translated into a new SQLStatement object. After that, the

translation is pushed to the stack. This process is continued until every node in the algebra tree has been translated. Section subsection 5.2.2 gives an example and illustrates how the translation is performed for an example SPARQL query.

Each *SQLStatement* object implements a *toString()* method which is finally called when the tree walker reaches the root node and thus yields the SQL translation of the SPARQL query as a result. Each child element hereby results in a subquery in the resulting SQL query.

5.2.1. SPARQL to SQL

As defined by the W3C, every SPARQL query can be transformed into an equivalent SPARQL algebra tree. Therefore, the translation for the SPARQL algebra operators are presented in the following:

BGP (basic graph pattern)

Optimization of the BGP translation could not be adopted from PigSPARQL, as the underlying data model is different from the property table based model used by Sempala. The translation of a basic graph pattern depends on the contained triples. For BGPs which only contain the same variable in subject position for each triple an object of class *Select* is created. It is translated to a single SQL SELECT statement, which selects those columns in the property table defined by the predicates in the triples. Listing 5.7 shows a BGP with ?*art* as the only subject in all triples. Its translation can be seen in Listing 5.8. The test for null values is necessary, as the property table design stores null values for those resources that do not own a certain property.

```
{
 ?art rdf:type ?type.
 ?art dc:title ?title
 ?art dc:pages ?pages
}
```

Listing 5.7: Example BGP

```
SELECT  ID as "art",
        rdf_type as "type",
        dc_title as "title",
        dc_pages as "pages"
FROM bigtable
WHERE ID is not null
AND rdf_type is not null
AND dc_title is not null;
AND dc_pages is not null;
```

Listing 5.8: code 2

If the BGP consists of more than one distinct variable in subject position, the BGP is translated into a Join object which joins every triple group which share a common variable in subject position. In order to automate this process, the triples are grouped by subject using a *HashMap* with the subject as key. If necessary, a Join is created, whereas each group of triples results in a Select object which is translated as subquery. The *ON-clause* of the join is determined by calculating the shared variables between the groups of triples. If two triple groups do not share any variables, the cartesian product is calculated by the SQL operator CROSS JOIN. In order to avoid unnecessary cross joins due to bad join order, a greedy approach is implemented (see Listing 5.9): Starting with an arbitrary group of triples, the number of shared variables between a selected group and all other groups which have not been joined yet, is calculated. Then, the group with the highest number of shared variables is selected and the join conditions are calculated. These steps are repeated until there are no groups of triples left to join.

```
15 int findBestJoin(Triplegroup group, ArrayList<TripleGroup> groups) {
16         int best = -1;
17         int index = 0;
18         for (int i = 0; i < groups.size(); i++) {
19                 int sharedVars = JoinUtil.getSharedVars(group, groups.get(i)).size();
20                 if(sharedVars > best){
21                         best = sharedVars;
22                         index = i;
23                 }
24         }
25         return index;
26 }
27 // generate Join object
28 Join generateJoin(TripleGroup group, List<TripleGroup> groups){
29         List<String> onConditions = new List<String>();
30         List<SQLStatement> rights = new List<SQLStatement>();
31         // Greedy approach: Find join partner with most shared vars.
32         while (groups.size() > 0) {
33           // calculate join
34           int index = findBestJoin(group, groups);
35           TripleGroup right = groups.get(index);
36           onConditions.add(getOnConditions(group, right)));
37           rights.add(right.translate());
38           groups.remove(index);
39         }
40         Join join = new Join(getResultName(), group.translate(), rights, onConditions,
               JoinType.natural);
41         return join;
42 }
```

Listing 5.9: Greedy algorithm to avoid unnecessary cross joins

Listing 5.10 gives an example of a BGP which requires a join in the SQL translation (Listing 5.11).

```
{
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?person foaf:name ?name .
}
```

Listing 5.10: Example SPARQL BGP

```
SELECT DISTINCT BGP1_1.person  AS "person",
                BGP1_0.article AS "article",
                BGP1_1.name    AS "name"
FROM (SELECT DISTINCT id        AS "person",
                      foaf_name AS "name"
      FROM   property_table
      WHERE  id IS NOT NULL
             AND foaf_name IS NOT NULL) BGP1_1
      JOIN (SELECT DISTINCT dc_creator AS "person",
                            id         AS "article"
      FROM   property_table
             WHERE  id IS NOT NULL
                    AND rdf_type IS NOT NULL
                    AND rdf_type = 'bench:Article'
                    AND dc_creator IS NOT NULL) BGP1_0
      ON( BGP1_1.person = BGP1_0.person )
```

Listing 5.11: Translation of SPARQL query in Listing 5.11

The first two triples form a triple group sharing the subject *?article* and are translated into the SELECT-subquery referenced by BGP1_0. The second triple group with *?person* as subject is translated into the subquery BGP1_1. The join is created with the appropriate *ON*-clause and wrapped into a *Select*-Statement in order to reference the result more easily, when it is a subquery of a more nested clause. This algorithm may not always find the optimal join order for the actual data, as a very selective clause, which should be joined early, may only share few variables with the rest. However, Sempala uses the automatic join optimization of Impala based on table statistics to optimize the join order. Therefore, it is enough to avoid cross joins and let the query engine of Impala reorder the joins.

A special case occurs if the predicate is a variable. Here, a separate Select object is created which retrieves the records from the triple store table stored separately.

Filter

A filter expression consists of nested arithmetic and logical expressions. The expression is therefore parsed into an expression tree which like the algebra tree is walked from the bottom up and the following translation rules are applied:

- The operators =, <, >, >=, <=, +, − and ! = are left unchanged in the SQL translation.
- The logical operators && and || are translated to *AND* and *OR*.
- Variables are translated to their corresponding subquery columns.
- The function *bound(?var)* is translated to *varcolumn IS NOT NULL*.
- *langMatch(expr,lang)* is translated to *expr LIKE '%lang'*.
- Literals are checked if they are numeric and if so left unchanged in the translation, while strings are embedded in inverted commas, e.g. "Amanda Knight".

Every SQL element implements the method *addWhereConjunction*. If called on a Select object, it adds the translated expression to a list, which is translated into a *SQL WHERE* clause. Similarly the Join object adds the filter condition to a list which is used for its *ON* clause in the SQL translation. Union recursively calls the method for its two children.

Listing 5.12 shows an example of a basic graph pattern with *FILTER* expression. The two triple groups do not share any variables and therefore results in a cross product. As the BGP is translated into a join, the translated filter expression is added to the *WHERE* clause of the enclosing *SELECT*.

```
{
bsbm:Prod245 bsbm:prop ?prop .
bsbm:Prod245 bsbm:name ?name .
FILTER regex(?name, "%word1%")
}
```

Listing 5.12: Example BGP with filter

```
SELECT DISTINCT bsbm_name AS "name",
                bsbm_prop AS "prop"
FROM   property_table
WHERE  id = 'bsbm:Prod245'
       AND bsbm_prop IS NOT NULL
       AND bsbm_name IS NOT NULL
       AND ( bsbm_name LIKE '%word1%' )
```

Listing 5.13: SQL code for BGP with filter

LeftJoin

Left join reuses the Join class and sets a flag to translate it to a left outer join in SQL. If the left outer join includes a condition, it is translated with the same expression walker for filters and added to the list of on-conditions.

Project

The projection is translated into a SQL object with the translated rest of the tree as *FROM* clause. The variables of the projection are added as selected columns to the Select object.

Distinct

As the implementation only supports distinct selection of values, this operator does not require special translation, as all SELECT statements already include the keyword *DISTINCT*.

ORDER BY

The method *addOrderBy()* is used to add the SQL ORDER BY clause to the SQLStatement object at the top of the stack. As Impala requires a limit to be set, a arbitrary very large number is set as limit.

Union

The SPARQL algebra Union operator is translated into a Union object, receiving two SQLStatement objects from the stack, which are referred to as subqueries.

Slice

The Slice operator is a combination of OFFSET and LIMIT clauses. This is applied to the top SQLStatement on the stack by calling the *addLimit* or *addOffset* method respectively.

5.2.2. Example

In the following, an example of translating a SPARQL algebra tree to SQL is given. Listing 5.14 shows an example SPARQL query, which is translated to the algebra tree illustrated in Figure 5.5.

```
PREFIX ex:
  <http://example.org/>
PREFIX rdf:
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?book ?pages
WHERE {
  { ?book rdf:type ex:Book.
    ?book ex:pages ?pages.
    FILTER (?pages > 30) }
  UNION
  { ?author ex:wroteBook ?book .
    ?book rdf:type ex:Book.
    ?book ex:pages ?pages.
    FILTER regex(?author, "%Erdoes%") }
}
```

Listing 5.14: Example SPARQL query

The triples form the leaves of the tree at level 0, while a *Distinct* operator node is the root of the tree at level 5.

1. BGP 1 is visited first. It receives the triples as input. The triples have the same variable in subject position. Thus, BGP 1 is translated to a *Select* object, which is pushed onto the stack. The select object selects those columns corresponding to the variables contained in the BGP.

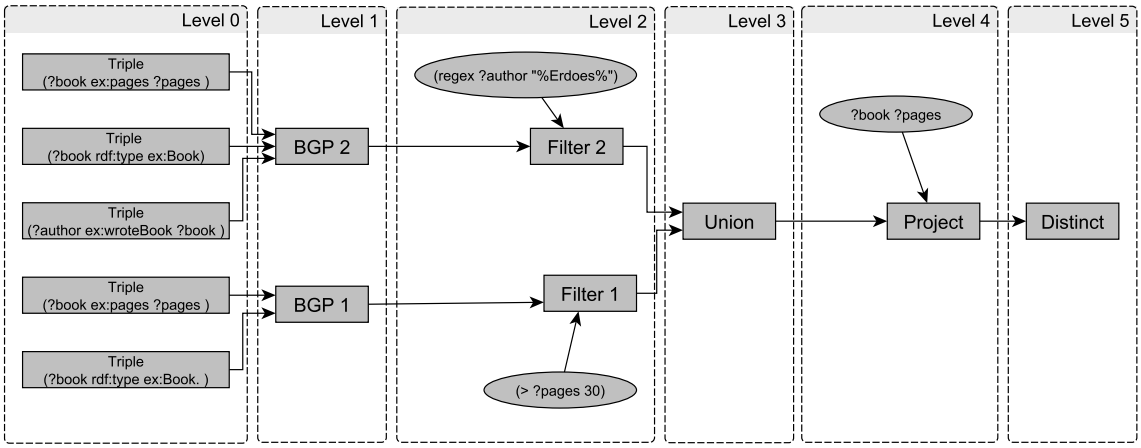


Figure 5.5.: SPARQL algebra sytax tree for query in Listing 5.14

2. After that, Filter 1 is processed next. The filter expression is translated into SQL by replacing the variable with its corresponding column name, i.e. (*ex_pages* > 30). After retrieving the *Select* object from the previous step from the stack, the filter condition is added to the WHERE clause conditions by calling the method *addWhereConjunction*. The modified *Select* object is then pushed back to the stack.

3. BGP 2 is visited by the tree walker. Unlike BGP 1, the triples have different variables in subject position. The first two triples have the variable *?book*, while the last triple has the variable *?author* in subject position. Therefore, the triples are grouped by subject into two groups. After calculating the shared variables of the two groups (i.e. *?book*) and calculating the resulting join condition, this information is used to create a new *Join* object. Each group is hereby translated into a SELECT subquery statement linked by *JOIN*. The subqueries are surrounded by a parent *SELECT* statement referred to as *wrapper* in the UML diagram in Figure 5.4. The field *joinType* is set to a value indicating a natural join. Listing 5.15 shows the corresponding SQL code of the *Join* object. As last step, the newly created *Join* object is pushed onto the stack. Hence, the stack contains a *Select* object and a *Join* object at this point.

```
(SELECT DISTINCT BGP2_1.author AS "author",
                 BGP2_0.pages  AS "pages",
                 BGP2_0.book   AS "book"
 FROM   (SELECT DISTINCT ex_pages AS "pages",
                         id       AS "book"
         FROM   property_table
         WHERE  id IS NOT NULL
                AND rdf_type IS NOT NULL
                AND rdf_type = 'ex:Book'
                AND ex_pages IS NOT NULL) BGP2_0
        JOIN (SELECT DISTINCT id           AS "author",
                              ex_wrotebook AS "book"
              FROM   property_table
              WHERE  id IS NOT NULL
                     AND ex_wrotebook IS NOT NULL) BGP2_1
          ON( BGP2_0.book = BGP2_1.book ))
```

Listing 5.15: Translation of BGP 2

4. Next, Filter 2 is processed following the same steps like Filter 1. The Join object from the previous step is retrieved from the top of the stack. The filter condition is added to the wrapper of the Join object, which is a *Select* object.

Analogously, the modified *Join* object is pushed back to the stack.

5. Subsequently, the Union node at level 3 is visited. After fetching the *Join* and *Select* objects from the stack, a *Union* object is created and pushed back onto the stack.

6. The following *Project* node is translated to a *Select* object, which receives the *Union* from the object as subquery for the *FROM* clause. Hereby only those columns are selected which correspond to the variables given as list to the *Project* node. After that, the *Select* object is pushed to the stack.

7. Due to the design of the property table, all *SELECT* queries must contain the keyword *DISTINCT*(see section 4.3 on page 41). Therefore, the *Distinct* node at level 5 is ignored and the stack left untouched.

8. Finally, the root node is reached and the *toString()* method of the current object at the top of the stack, i.e. the *Select* object resulting from the *Distinct* node, is called. It yields the SQL query in Listing 5.16.

```
SELECT DISTINCT UNION1.pages AS "pages",
                UNION1.book  AS "book"
FROM   ((SELECT DISTINCT NULL     AS "author",
                         ex_pages AS "pages",
                         id       AS "book"
         FROM   property_table
         WHERE  id IS NOT NULL
                AND rdf_type IS NOT NULL
                AND rdf_type = 'ex:Book'
                AND ex_pages IS NOT NULL
                AND ( ex_pages > 30 ))
        UNION
        (SELECT DISTINCT BGP2_1.author AS "author",
                         BGP2_0.pages  AS "pages",
                         BGP2_0.book   AS "book"
         FROM   (SELECT DISTINCT ex_pages AS "pages",
                                 id       AS "book"
                 FROM   property_table
                 WHERE  id IS NOT NULL
                        AND rdf_type IS NOT NULL
                        AND rdf_type = 'ex:Book'
                        AND ex_pages IS NOT NULL) BGP2_0
                JOIN (SELECT DISTINCT id           AS "author",
                                      ex_wrotebook AS "book"
                      FROM   property_table
                      WHERE  id IS NOT NULL
                             AND ex_wrotebook IS NOT NULL) BGP2_1
                  ON( BGP2_0.book = BGP2_1.book )
         WHERE  ( BGP2_1.author LIKE '%Erdoes%' ))) UNION1
```

Listing 5.16: Translation of SPARL algebra tree in Figure 5.5

6. Performance evaluation

In this chapter, the performance of Sempala is tested with three different SPARQL benchmarks, LUBM, BSBM and SP^{2}Bench. All benchmarks offer SPARQL queries which run on synthetically generated RDF data. The goal of the evaluation is to compare Impala, a MPP-engine based database, with MapReduce-based systems, as Impala was developed to complement these systems for interactive queries. Apache Hive is such a MapReduce-based system and was chosen for comparison, as it is compatible with the tables generated by Impala and the corresponding queries. Thus, the performance of the two engines could be compared without having to re-implement the components of Sempala for Hive. When comparing the results of Hive, it has to be considered that the table format - especially the choice of Parquet - may not be optimal for Hive. Nevertheless, a general notion which type of queries perform better than others can be gained from the evaluation.

In order to measure the time of a SQL query without the overhead of transferring the results back to the node which placed the query, the results were written into a result table (see Listing 6.1).

```
CREATE TABLE result AS
  SELECT DISTINCT BGP1_0.article AS article
  FROM   ... ;
```

Listing 6.1: Write SQL query results to a table

6.1. Benchmark environment

The evaluation was performed on a cluster of 10 Dell PowerEdge R320 servers equipped with a Xeon E5-2420 1.9 GHz 6-core CPU, 32 GB RAM, 4 TB disk space and connected via gigabit network. The software installation included Cloudera's Distribution of Hadoop (CDH4)[1] installed on Ubuntu Linux 12.04 LTS. Impala is

[1] http://www.cloudera.com/hadoop/

installed in version 1.2.3.

6.2. Compatibility issues with Hive

Generally, the Impala SQL dialect is compatible to Hive's query language, HiveQL. However, some queries had to be rewritten due to minor syntactic differences, e.g. the use of brackets. Apart from that, the operators *UNION* and *LEFT JOIN* had to be changed to *UNION ALL* and *LEFT OUTER JOIN*, respectively.

During the evaluation of Hive, problems arose when trying to read tables created by Impala in Parquet format with Hive. Tests showed that Impala had no problem reading tables in Parquet format which were created by Hive. The problem was narrowed down to a programming bug of the Parquet reader used by Hive, which leads to an error when a column only containing null values is read. The problem also affected other Impala users and was mentioned in this forum post[2]. As a result, the tables were reloaded using Hive, which solved the problem.

6.3. Lehigh University Benchmark

The Lehigh University Benchmark (LUBM) was first presented at the ISWC conference in 2004 [29]. Describing information from a university domain, *Univ-Bench* is the ontology used in the benchmark. LUBM allows to generate synthetic Semantic Web data defined in OWL-lite [30], providing 14 SPARQL queries based on this ontology to run on this data. All 14 queries are written in SPARQL 1.0 and consist only of basic graph patterns and are listed in section B.1 on page 99ff. The scaling factor is the number of universities in the dataset. As Sempala does not support OWL reasoning, the WebPIE (Web-scale Parallel Inference Engine) OWL reasoner [31] was used to generate reasoned versions of the OWL-lite data. As this process generates duplicates, the dataset was further pre-processed when it was loaded into Impala. Only distinct RDF triples were loaded, while also replacing URIs with prefixes. Datasets ranging from 5 to 3000 universities were chosen for the evaluation. Table 6.1 shows the loading times including the preprocessing of the reasoned RDF data with duplicates. It also shows the original file size and the

[2] `https://groups.google.com/d/topic/parquet-dev/Q-TPuA2rGk0/discussion`

universities	5	500	1000	1500	2000	2500	3000
loading time (Impala)	2min 25s	22min 25s	40min 8s	55min 50s	76min 41s	93min 11s	113min 27s
loading time (Hive)	n.a.	24min 03s	43min 42s	59min 40s	72min 57s	86min 54s	109min 37s
distinct triples	982,219	105,029,856	210,233,711	315,130,716	420,148,379	525,386,748	630,173,960
Sempala database size	19.1M	2.6G	5.1G	7.7G	11.0G	14.9G	17.8G
original file size	161M	17.0G	34.1G	51.3G	68.5G	85.7G	102.9G

Table 6.1.: LUBM loading times and dataset sizes

summed up size of the property and triple tables, i.e. Sempala database size. The Parquet encoding mechanisms, the elimination of duplicates and the Snappy compression substantially decrease the sizes of the tables compared with the original file sizes.

6.3.1. Benchmark runtimes

Table 6.2 and Table 6.3 show the runtimes of the translated queries Q1 through Q14 (see section B.2) running Impala and Hive. Impala outperforms Hive on every of the fourteen queries. For Hive, Q7 evaluated on the dataset with 3000 universities took the longest (417s), while the maximum runtime for Impala is around 66 seconds for Q6 on the same dataset.

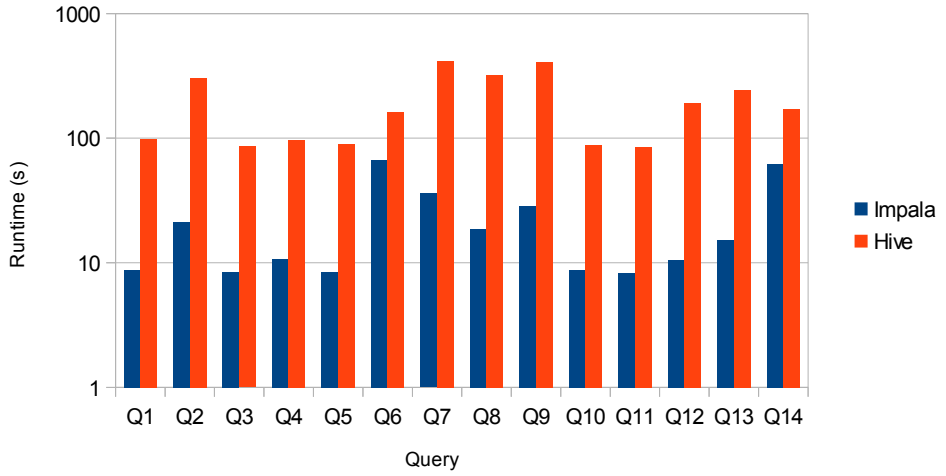


Figure 6.1.: Absolute runtimes of Hive and Impala on LUBM 3000 dataset (logarithmic scaling)

The runtimes are visualized in Figure 6.1 (logarithmic scale). The performance

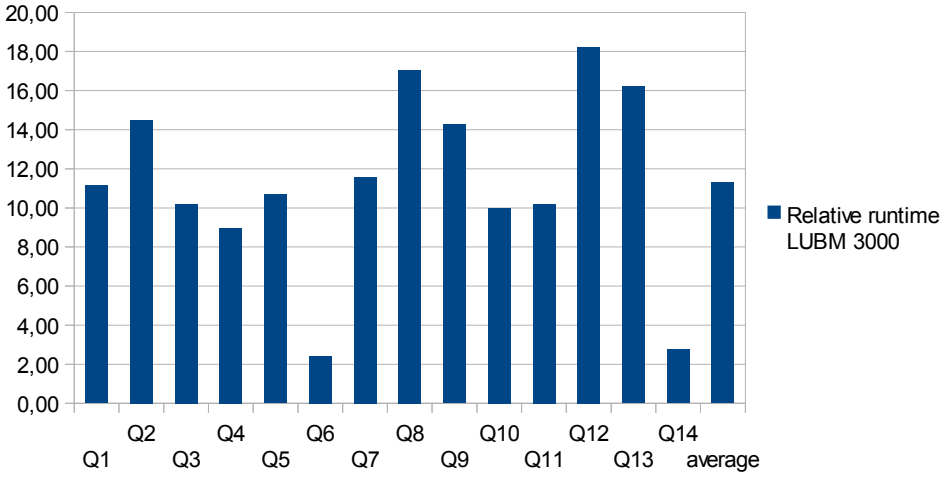


Figure 6.2.: Relative runtimes of Hive compared with Impala on LUBM 3000 dataset

advantage of Impala becomes even more apparent, when analyzing the relative runtimes of Hive compared to Impala in Figure 6.2. On average, Impala outperforms Hive by a factor of 11.3. Before the evaluation, it was expected for Impala to perform well for selective queries. This was confirmed by the evaluation results. For example, Q12 and Q8 are queries, where the performance difference is high, while the difference is low for Q6 and Q14. This can be explained by considering the types of queries. Q6 and Q14 are very unselective, as they query all students or all undergraduate students, respectively, while Q12 and Q8 are more selective. For example, Q8 queries all department members of a specific university (University0) who are students. Q12 selects those chairs which work for a subdepartment of University0. Ergo, the performance advantage is higher for Q8 and Q12, as these are the types of queries which Impala was designed for.

6.3.2. Scaling of data sizes

Figure 6.3 visualizes the runtimes for Q5, Q6 and Q14 running on different dataset sizes. The scaling behavior corresponds to the type of query, i.e. lookup query, star query or linear query (cf. chapter 4). On the one hand, Q5 shows nearly constant runtimes for 5 to 3000 universities, while the plots for Q6 and Q14 are linear. Q5 is very selective, as it selects those entities of type person which visit a specific university. The number of results does not increase for larger datasets for this query. As stated in the previous subsection, Q6 and Q14 are unselective and

therefore force the system to scan more of the dataset with increasing dataset size. The plots for the same queries running on Hive, show similar behavior but are not as definite as the plots for Impala. Especially the difference in runtimes for the dataset consisting of 5 universities, does not allow an interpretation of constant runtime for Q5. The scaling behavior of Q6 and Q14 are similar for Hive and Impala, whereby the plots for Impala show less fluctuation, which is normal for MapReduce.

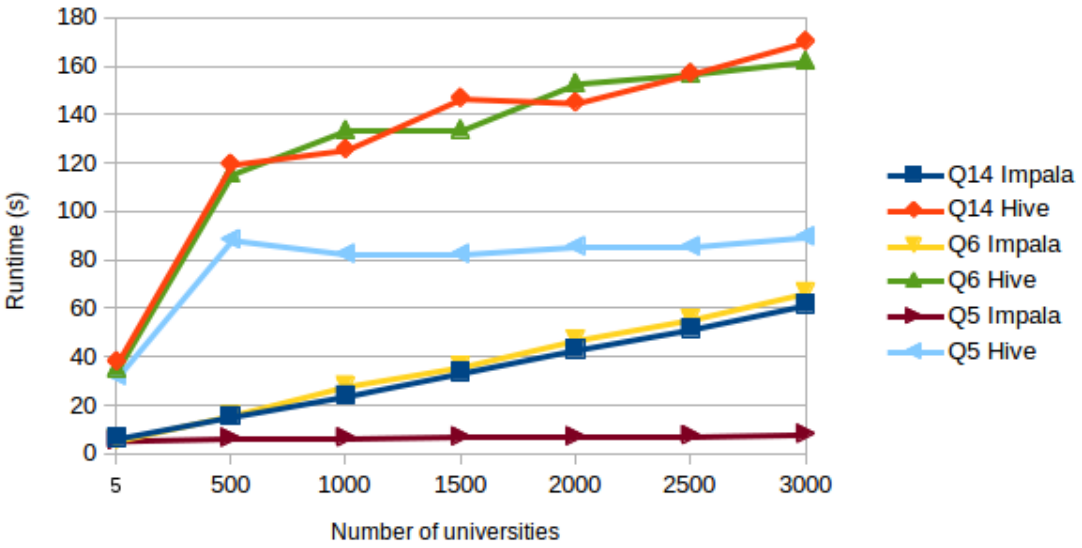


Figure 6.3.: Runtimes for selected queries on different dataset sizes

All in all, the results of LUBM show that Impala performs well on basic graph pattern evaluation. Compared with Hive, the queries run one order of magnitude faster on average. For the tested dataset sizes, a nearly constant runtime for selective queries was achieved.

universities	**5**	**500**	**1000**	**1500**	**2000**	**2500**	**3000**
Q1(Impala)	5.78	6.97	7.09	7.38	7.61	8.21	8.79
Q1 (Hive)	31.303	85.542	77.592	82.676	80.737	89.887	98.031
results	4	4	4	4	4	4	4
Q2 (Impala)	7.11	9.24	11.47	13.64	15.80	18.63	21.07
Q2 (Hive)	114.843	255.496	253.613	269.876	298.174	297.423	305.618
results	9	1284	2528	2528	2528	2528	2528
Q3 (Impala)	5.76	6.78	6.88	7.19	7.40	7.91	8.41
Q3 (Hive)	32.354	96.581	75.591	78.632	89.851	84.79	85.822
results	6	6	6	6	6	6	6
Q4 (Impala)	6.09	8.28	8.66	8.97	9.63	9.96	10.71
Q4 (Hive)	34.458	100.802	89.771	92.964	97.997	96.07	96.107
results	34	34	34	34	34	34	34
Q5 (Impala)	5.75	6.80	6.90	7.32	7.38	7.85	8.41
Q5 (Hive)	32.272	88.691	82.756	82.72	85.814	85.801	89.899
results	719	719	719	719	719	719	719
Q6 (Impala)	5.86	16.22	28.28	36.29	47.32	55.83	66.86
Q6 (Hive)	35.438	115.792	133.893	134.115	153.278	157.193	162.317
results	40087	4325023	8653646	12977778	17299925	21633320	25948698
Q7 (Impala)	6.78	12.05	17.59	22.49	27.02	31.02	36.07
Q7	96.285	293.153	310.567	318.01	353.23	380.674	417.934
results	61	61	61	61	61	61	61
Q8 (Impala)	6.78	9.75	10.89	12.95	14.57	16.77	18.81
Q8 (Hive)	79.584	229.39	243.545	236.551	284.052	290.169	320.465
results	6463	6463	6463	6463	6463	6463	6463
Q9 (Impala)	6.78	10.81	14.20	17.55	21.49	25.24	28.55
Q9 (Hive)	114.785	306.863	312.01	360.265	371.651	405.116	407.503
results	784	86085	172632	258622	344736	430987	516709

Table 6.2.: LUBM runtimes (in s) for queries Q1-Q9

universities	**5**	**500**	**1000**	**1500**	**2000**	**2500**	**3000**
Q10 (Impala)	6.78	6.98	7.09	7.48	7.69	8.22	8.81
Q10 (Hive)	32.318	89.774	77.749	82.767	82.698	85.863	87.934
results	0	0	0	0	0	0	0
Q11(Impala)	6.78	6.69	6.80	7.08	7.20	7.58	8.32
Q11 (Hive)	35.251	83.628	74.643	77.659	80.83	80.802	84.921
results	224	224	224	224	224	224	224
Q12 (Impala)	6.78	7.40	8.18	8.86	9.11	9.74	10.56
Q12 (Hive)	81.317	165.033	165.219	190.461	180.407	187.596	192.526
results	0	0	0	0	0	0	0
Q13 (Impala)	6.78	8.29	9.57	11.05	12.01	13.84	15.14
Q13 (Hive)	79.301	192.74	217.06	218.11	237.25	250.31	245.46
results	21	2370	4760	7049	9351	11702	14097
Q14	6.78	15.72	24.24	33.81	43.35	51.81	61.85
Q14	38.312	119.852	125.929	147.09	145.224	157.283	170.524
results	36682	3961133	7924765	11886084	15845036	19813828	23765746

Table 6.3.: LUBM runtimes (in s) for queries Q10-Q14

6.4. Berlin SPARQL Benchmark

First published in 2009, the Berlin SPARQL Benchmark (BSBM)[32] is designed to test SPARQL endpoint systems with synthetic RDF data. The BSBM includes a data generator and a test driver. Version 3.1 was used for the evaluation. The generated RDF graph describes an e-commerce use case including products, vendors, user ratings with comments and other related items. Generated data can be scaled to arbitrary sizes, by defining the number of products and can be stored in different formats. The nt-format was chosen for the evaluation, as the Sempala loader component includes a parser for this format.

While the test driver offers three different use cases, the *explore* use case is the only read-only use case and is therefore used for the evaluation of Sempala, as Impala does not support UPDATE SQL queries. The explore use case describes the scenario of users searching for products and consists of 12 query templates which comply to the SPARQL 1.0 standard. Q9 (DESCRIBE query) and Q12 (CONSTRUCT query) are not supported because Sempala only supports SELECT-queries. The test driver replaces the placeholders in the templates with actual values depending on the generated dataset. This process utilizes statistics collected during data generation. As Sempala does not implement the SPARQL protocol and cannot be used as a SPARQL endpoint, the test driver was used to generate 20 different instances of each query, summarizing to a total of 200 queries used for the evaluation. As the actual queries depend on the respective dataset on which the queries are run, this process was repeated for all six datasets used during the evaluation resulting in a total of 1200 queries that were translated and run by Sempala.

In [32], Bizer et. al. suggest three fundamental performance metrics for the benchmark: Query Mixes per Hour (QMpH), Queries per Second (QpS) and Load time (LT). A query mix hereby is a certain ordered set of queries predefined by the benchmark designers. As the benchmark was designed to test SPARQL endpoints with concurrent threads, QMpH and QpS are not appropriate for testing Sempala. Instead, the load time and the average query execution time (aQET) for each query for a fixed number of runs was measured, as suggested as performance metric for single queries.

Following the same procedure as with the LUBM benchmark in section 6.3, the datasets are loaded with Impala and Hive by the loader module. Six datasets were generated for the evaluation ranging from 500,000 to 3,000,000 products. Table 6.4

shows the loading times and dataset sizes for the six datasets. Comparing Hive and Impala, the loading times are similar but show shorter loading times for Impala. The Parquet file format with Snappy compression performs well and produces a significant reduction of the datasize compared with the original file size.

products	**500K**	**1000K**	**1500K**	**2000K**	**2500K**	**3000K**
loading time (Impala)	25min	70min	70min	92min	151min	138min
loading time (Hive)	32min	63 min	92min	120 min	155min	176min
triples	175,526,449	350,563,015	525,594,458	700,623,319	875,991,782	1,051,007,783
Sempala database size	17.5G	35.0G	52.6G	70.3G	87.9G	105.4G
original file size	42.9G	85.9G	129.2G	172.5G	215.9G	259.3G

Table 6.4.: BSBM loading times and dataset sizes.

6.4.1. Benchmark runtimes

Table 6.5 and Table 6.6 show the runtimes for each of the ten queries. There are different number of average results for Hive because only 10 different instances were run per query for Hive due to the expected longer runtimes. As with LUBM benchmark, Impala outperforms Hive on every query. Nevertheless, Q7 shows a memory error, which occurs when two tables cannot be joined in memory, as they are too large. Hive has no problem scaling the dataset size, however, it runs much longer.

Figure 6.4 visualizes the absolute runtimes of Hive and Impala respectively for the dataset consisting of 2.5 million products. This dataset was chosen, as all queries can be evaluated on Impala and Hive. The runtimes of Hive are set in relation to the runtimes of Impala in Figure 6.5. On average, Impala runs faster by a factor of 20. While all relative runtimes show a performance advantage of Impala, the advantage is lowest for Q11. Q11 selects all properties which either have a given resource in subject position, object position, or both. As this query requires the use of the triple table, which is not optimal for Impala, the performance gap is therefore lower for this query. At the other end of the scale, Impala outperforms Hive significantly for nearly all other queries, which can be explained by the selectivity of the queries. For example Q4 is well-suited for Impala, as it involves a union of products which are of a specific product type and must posses specific product features and properties. These properties are even further specified by filters. The two subqueries of the

union are star-queries and hence can each be translated as a single SQL SELECT query. Thus, it takes advantage of the property table layout, as it does not require any joins.

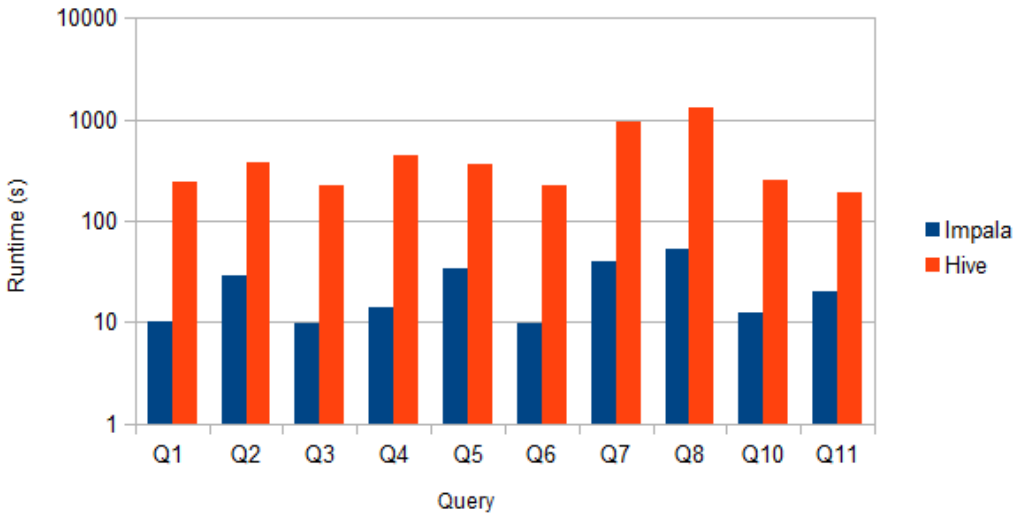


Figure 6.4.: Absolute runtimes of Hive and Impala on BSBM 2500K dataset (logarithmic scaling)

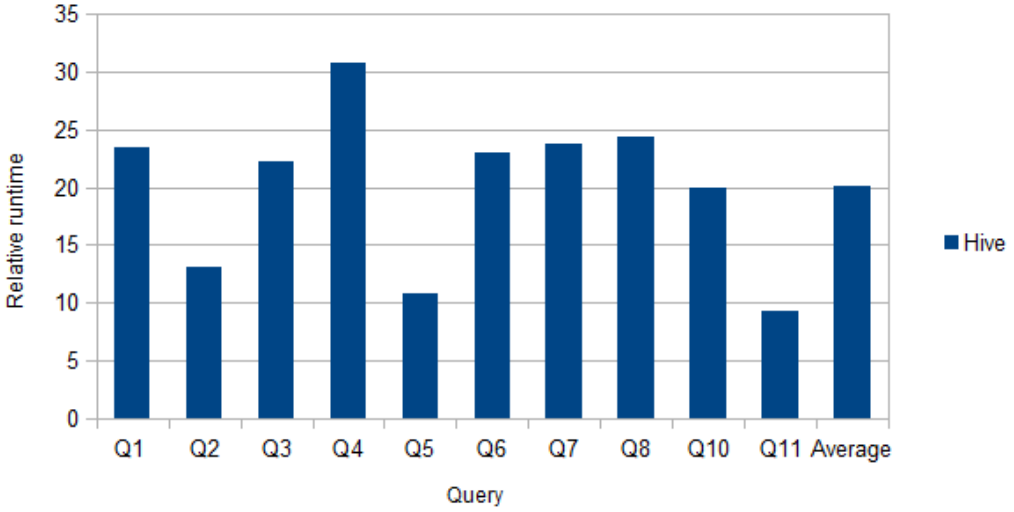


Figure 6.5.: Relative runtimes of Hive compared with Impala on BSBM 2500K dataset

6.4.2. Scaling of dataset sizes

Figure 6.6 shows the behavior of queries Q3, Q6 and Q11 for increasing dataset sizes. While Q3, Q6 and Q11 show linear behavior on Hive, Q3 and Q6 show nearly constant runtimes for Impala. This was expected, as Q3 and Q6 are star queries which are well-suited for the property table approach used by Sempala. Q11 requires the triple table and therefore shows linear scaling behavior for Impala and Hive.

In summary, the BSBM memory confirms the findings of the LUBM benchmark.

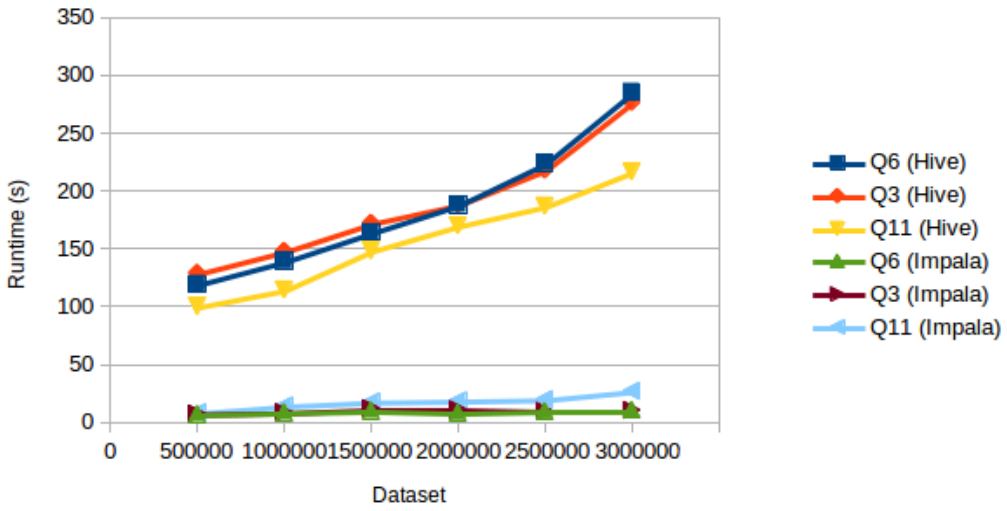


Figure 6.6.: Scaling behavior of BSBM benchmark queries Q3, Q6 and Q11

number of products	500K	1000K	1500K	2000K	2500K	3000K
Q1 (Impala)	7.42	8.7	12.6	11.01	10.19	10.70
avg. results	7.75	9.45	9.45	10	9.4	9.55
Q1 (Hive)	146.74	164.02	184.64	202.82	238.69	294.05
avg. results	7.75	8.7	8.9	9.2	8.8	9.1
Q2 (Impala)	13.81	16.92	21.39	23.9	28.517	30.76
avg. results	18.4	18.85	17.15	18.8	19.85	19.3
Q2 (Hive)	181.86	219.03	257.49	307.48	373.58	414.02
avg. results	17.9	18.2	14.7	18.5	19.1	17.9
Q3 (Impala)	7.68	8.83	12.35	11.34	9.81	10.72
avg. results	5.05	7.45	7.45	7.85	7.05	7.45
Q3 (Hive)	129.22	148.31	172.551	188.793	218.67	277.29
avg. results	4.2	6.8	7.2	7.4	6.3	6.8
Q4 (Impala)	9.37	11.7	13.82	16.06	14.17	15.63
avg. results	8.9	10	10	10	9.9	10
Q4 (Hive)	255.41	292.63	336.241	366.999	435.2	490.42
avg. results	8.9	10	10	10	10	10
Q5 (Impala)	14.21	17.77	22.59	27.79	33.53	37.6
avg. results	5	5	4.75	5	5	5
Q5 (Hive)	193.35	233.88	265.18	309.3	363.24	399.79
avg. results	5	5	5	5	5	5

Table 6.5.: BSBM runtimes in seconds

number of products	500K	1000K	1500K	2000K	2500K	3000K
Q6 (Impala)	7.2105	8.74	10.26	8.48	9.76	10.77
avg. results	1.9	5.6	5.6	7.25	9.65	11
Q6 (Hive)	119.72	139.62	164.52	188.48	224.58	285.8
avg. results	1.9	3.2	5.7	7.5	10.1	11.2
Q7 (Impala)	14.42	28.20	28.53	37.3	39.92	MEM
avg. results	9.3	15.85	11.65	8.1	8.6	MEM
Q7 (Hive)	528.36	631.12	732.259	828.332	948.2	1203.81
avg. results	9.3	13.2	17	12	7.1	18.1
Q8 (Impala)	15.39	36.12	36.31	48.5	52.59	63.76
avg. results	0	0	0	0	0	0
Q8 (Hive)	660.87	818.63	1001.46	1169.3	1284.43	1554.69
avg. results	0	0	0	0	0	0
Q10 (Impala)	12.76	11.52	13.86	12.37	12.63	13.39
avg. results	3.9	3.7	4.84	1.9	4.6	2.95
Q10 (Hive)	142.48	169.08	188.89	211.03	252.78	265.445
avg. results	3.6	3.2	4.2	3.2	5.8	2.2
Q11 (Impala)	8.67	14.58	18.00	18.97	20.08	27.3
avg. results	10	10	10	10	10	10
Q11 (Hive)	100.38	114.79	148.686	170.332	187.26	216.93
avg. results	10	10	9.8	10	10	10

Table 6.6.: BSBM runtimes in seconds

6.5. SP²Bench

Like LUBM and BSBM, SP²Bench [27] provides a data generator for arbitrary dataset sizes and pre-defined queries. Queries Q1 through Q7 and Q9 through Q11 were chosen for the benchmark and are listed in Appendix D on page 121ff. The translation of Q8 lead to memory issues during execution, as the intermediate results were too large. In [27], the authors suggest special optimization techniques to perform on this query. As the focus of this thesis is not the optimization of SPARQL queries, Q8 was omitted for the evaluation. The translation of query Q6 yielded an error which was specific to Hive and involved the way Sempala uses table aliases for subqueries. The query translated by Sempala could be run on Impala but threw a semantic error on Hive. Q12 and Q14 are not supported, as Sempala only supports SPARQL SELECT queries. It was therefore only run for Impala. Table 6.7 shows the loading times for the datasets and their respective sizes. SP²Bench is known for its complex queries, which can only be evaluated with advanced query optimization [33]. Therefore, SP²Bench was chosen to show the expected limitations of Sempala for certain queries.

	DBLP25M	**DBLP50M**	**DBLP100M**	**DBLP200M**	**DBLP300M**	**DBLP400M**	**DBLP500M**
loading time (Hive)	635s	727s	985s	1474s	2009s	2459s	2761s
triples	25,000,113	50,000,109	100,000,153	200,000,978	300,000,056	400,000,632	500,000,980
database size	3.9G	7.9G	15.7G	31.4 G	46.8G	62.4G	78.1G
original file size	2.6G	5.3G	10.5G	20.7G	30.9	41.1	51.3

Table 6.7.: SP²Bench loading times and dataset sizes

6.5.1. Table partitioning

SP²Bench was used to test the performance of table partitioning by RDF type property (cf. 5.1 on page 53) as the generated RDF data is well structured and features 13 distinct types (which is not too few and not too many). Table 6.8 gives an overview of all runtimes. Although the analysis of the profile log files states a correct usage of the partitions, there was no significant performance benefit from partitioning the tables and for some queries, e.g. Q5a, results in longer runtimes. Possible causes involve too small dataset sizes for partitioning to have an effect. It is also possible that the optimization of Impala's query engine performs very well

even without partitioning for these dataset sizes. The overhead of reading from multiple partitions may in these cases outweigh the performance benefit. Hence, table partitioning was not considered in the following comparison of Impala and Hive.

6.5.2. Runtime comparison

Figure 6.7 shows the absolute runtimes for all queries for the dataset consisting of 100 million triples. As can be seen in the diagram, Impala outperforms Hive for every query which finishes without error for this dataset size. The in-memory-joins cause more problems for larger dataset sizes, as visualized in Figure 6.8. Similar to the other benchmarks, Impala is one order of magnitude faster for certain queries. This becomes clear when considering the relative runtimes of Hive compared with Impala, as depicted in Figure 6.9. Q2 and Q7 have the smallest difference in performance. For Q2, this can be explained by the low selectivity and the high number of results. Q7 consists of a lot of joins and selects many columns, which is not optimal for Parquet and therefore results in longer runtimes for Impala. In contrast to this, Q11 only selects two columns and is consequently well-suited for the Parquet file format. Q4 does not finish on the 100 million triple dataset using Impala, however, it can be run using Hive. As described in section 3.5, Impala 1.2.3 requires both tables to fit in memory, when performing a join. Hive on the other hand does not have this limitation and therefore can process larger datasets.

Before running the benchmark, it was expected for Impala to yield better runtimes the more selective a query is. The runtimes of Q3a to Q3c prove this theory to be true. The selectivity increases from Q3a (many results) to Q3c (no results) and corresponds to the measured runtimes. Considering the relative runtimes in Figure 6.9, the performance advantage of Impala compared with Hive increases with higher selectivity.

Some runtimes could have been improved by more advanced query optimization but this was not the goal of this thesis and is a separate field of research [33].

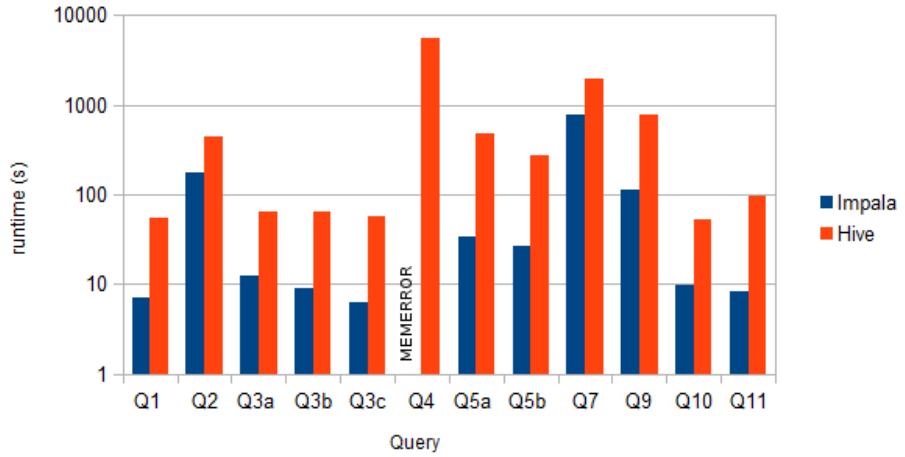


Figure 6.7.: Absolute runtimes for Hive and Impala (unpartitioned) on 100 million triples SP^{2}Bench dataset. Y-axis: logarithmic scale

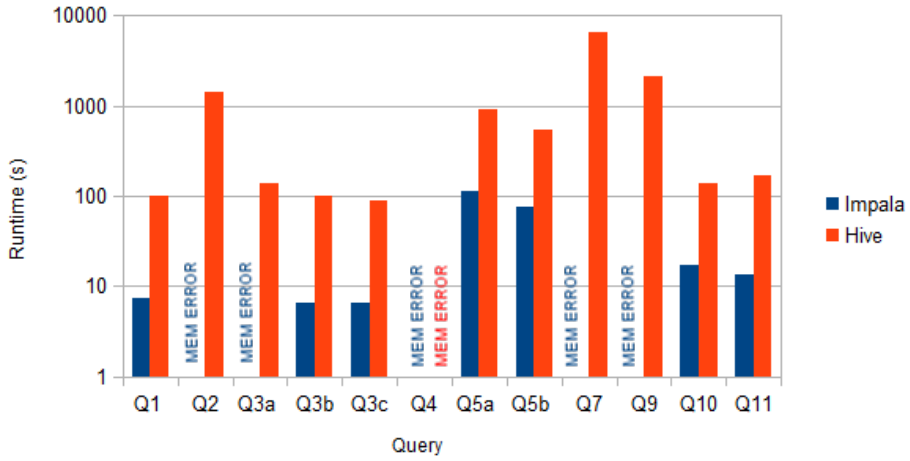


Figure 6.8.: Absolute runtimes for Hive and Impala (unpartitioned) on 500 million triples SP^{2}Bench dataset. Y-axis: logarithmic scale

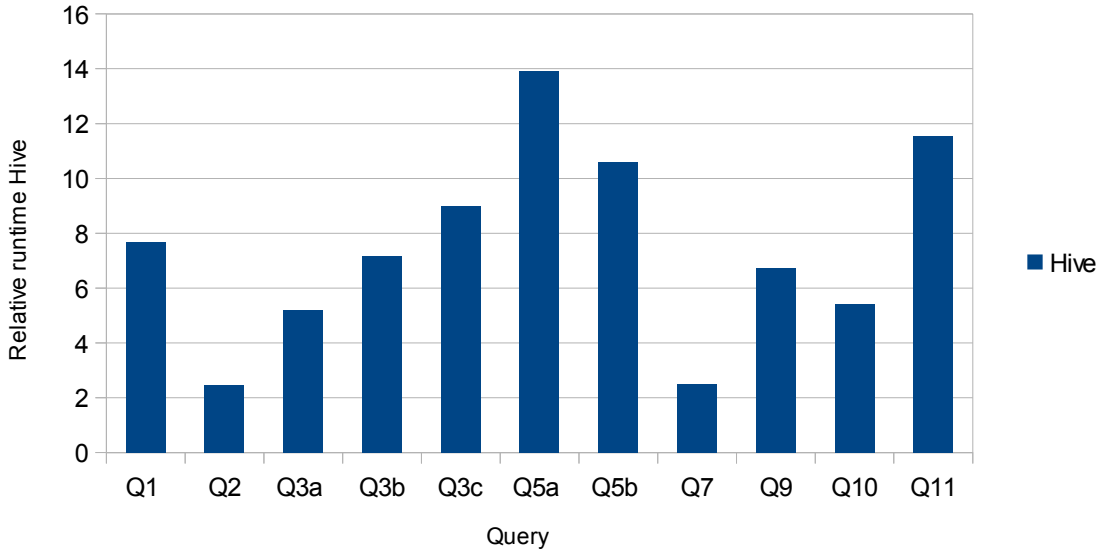


Figure 6.9.: Relative runtimes for Hive compared with Impala (unpartitioned on 100 million triples SP²Bench dataset)

6.5.3. Scaling of dataset sizes

Figure 6.10 and Figure 6.11 shows the runtimes for chosen queries on different dataset sizes ranging from 25 million to 500 million triples. Although the runtimes are much longer for Hive, the scaling behavior is very similiar for both Hive and Impala. Q5a shows linear runtimes, and hence shows a greater dependency between the runtimes and the dataset sizes. Apart from that, the query Q1 (like Q3b and Q3c) shows a nearly constant runtime for Impala. This is consistent with the query types, as queries Q1, Q3b and Q3c do not require any joins and are selective, which favors the design of Impala.

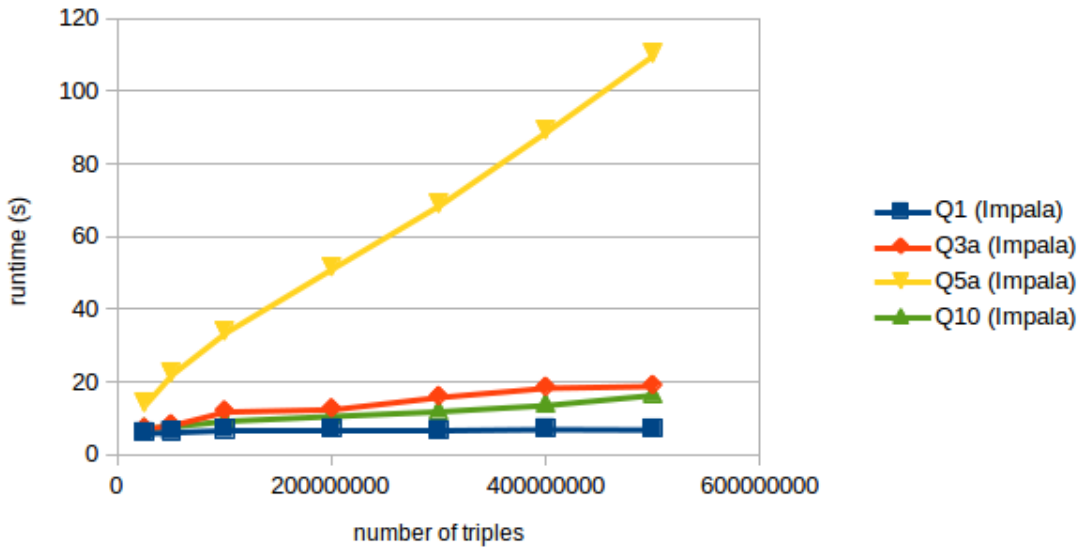


Figure 6.10.: Scaling behavior of Impala

In summary, SP^{2}Bench clearly shows the join limitations of Impala and the potential for optimization of the translation module of Sempala. Nevertheless, there exists a significant performance advantage of Impala compared with Hive for those queries which execute without failure.

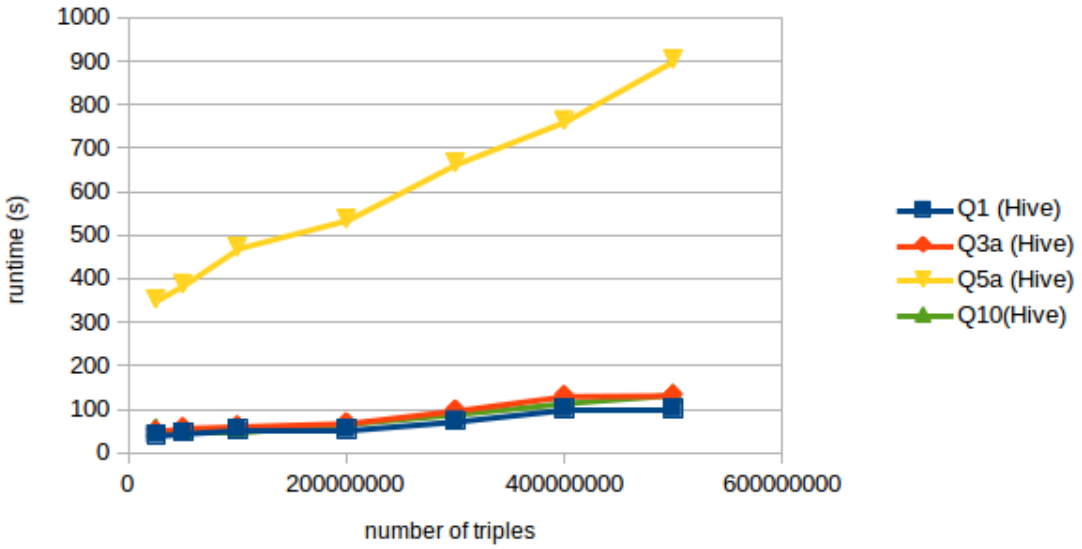


Figure 6.11.: Scaling behavior of Hive

	DBLP25M	DBLP50M	DBLP100M	DBLP200M	DBLP300M	DBLP400M	DBLP500M
Q1 (Impala)	6.27	6.55	7.14	7.34	6.97	7.37	7.27
Q1 (Impala part)	5.73	5.83	5.72	5.72	5.84	5.82	5.84
Q1 (Hive)	42.35	47.44	54.93	54.65	75.18	101.54	101.69
results	1	1	1	1	1	1	1
Q2 (Impala)	41.5	87.6	175.73	350.14	MEM	MEM	MEM
Q2 (Impala part)	50.02	96.67	180.77	362.1	MEM	MEM	MEM
Q2 (Hive)	296.58	353.11	435.41	646.1	857.46	1091.65	1387.29
results	1896297	4230949	9097526	18764382	28306882	37935543	47550273
Q3a (Impala)	7.49	8.47	12.23	12.85	16.24	18.77	19.27
Q3a (Impala part)	8.72	11.94	11.71	12.85	17.28	19.38	22.27
Q3a (Hive)	52.47	59.62	63.64	70.9	100.25	133.16	134.89
results	598936	929547	1471971	2456265	3414800	4380891	5346697
Q3b (Impala)	5.96	6.16	8.88s	6.82	6.52	6.6	6.58
Q3b (Impala part)	5.98	6.26	6.38	6.37	6.39	6.4	6.4
Q3b (Hive)	52.66	47.88	63.72	70.9	75.21	100.56	100.61
results	4196	6481	10287	2456265	23924	30756	37425
Q3c (Impala)	5.9	5.99	6.31	6.3	6.42	6.41	6.5
Q3c (Impala part)	5.99	6.19	6.37	6.21	6.21	6.22	6.24
Q3c (Hive)	42.46	49.72	56.65	64.9	72.08	87.48	89.67
results	0	0	0	0	0	0	0
Q4 (Impala)	401.74	MEM	MEM	MEM	MEM	MEM	MEM
Q4 (Impala part)	300.97	MEM	MEM	MEM	MEM	MEM	MEM
Q4 (Hive)	42.46	5343.38	5592.71	MEM	MEM	MEM	MEM
results	115349869	234720384	462943026	-	-	-	-
Q5a (Impala)	14.33	22.56	34.02	51.68	69.14	89.39	110.5
Q5a (Impala part)	20.79	34.31	47.4	86.1	122.72	156.34	199.55
Q5a (Hive)	353.67	389.04	473.14	538.06	666.94	764.64	904.4
results	815864	1384044	2049077	2970234	3795107	4563184	5371363
Q5b(Impala)	12.08	18.15	26.14	38.27	48.49	60.76	74.65
Q5b (Impala part)	16.7	25.77	34.25	49.28	70.63	84.36	117.79
Q5b (Hive)	229.61	255.68	277.23	336.64	435.15	453.68	540.05
results	815864	1384044	2049077	2970234	3795107	4563184	5371363

Table 6.8.: SP^{2}Bench runtimes (in s) - *MEM* indicates a memory error.

	DBLP25M	DBLP50M	DBLP100M	DBLP200M	DBLP300M	DBLP400M	DBLP500M
Q6 (Impala)	65.01	137.4	261.81	464.7	696.25	924.66	1148.92
Q6 (Impala part)	72.49	141.16	271.22	481.83	733.71	953.95	1216.24
Q6 (Hive)	TRANS	TRANS	TRANS	TRANS	TRANS	TRANS	TRANS
results	2226471	5349308	10541108	22254823	34843701	46843176	59071867
Q7 (Impala)	129.49	310.41	775.84	MEM	MEM	MEM	MEM
Q7 (Impala part)	135	312.12	777.19	MEM	MEM	MEM	MEM
Q7 (Hive)	1016.74	1289.58	1944.76	3804.71	4259.49	5744.06	6594.53
results	5170	8876	14526	18593	18927	20226	20244
Q9 (Impala)	34.48	56.37	114.13	223.56	MEM	MEM	MEM
Q9 (Impala part)	33.29	63.42	120.99	MEM	MEM	MEM	MEM
Q9 (Hive)	529.45	627.96	768.48	1025.25	1353.19	1728.29	2095.71
results	4	4	4	4	4	4	4
Q10 (Impala)	7.55	8.19	9.67	11.01	12.31	14.03	16.82
Q10 (Impala part)	7.29	8.16	9.91	10.67	13.59	15.81	17.33
Q10 (Hive)	57.17	52.6	52.44	69.44	91.9	117.18	136.63
results	656	656	656	656	656	656	656
Q11 (Impala)	6.37	7.01	8.19	9.78	10.59	11.99	13.43
Q11 (Impala part)	7.79	10.52	9.59	12.68	19.16	18.56	27.17
Q11 (Hive)	72.06	84.13	94.64	109.54	131.88	164.38	169.8
results	10	10	10	10	10	10	10

Table 6.9.: SP²Bench runtimes (in s) - MEM indicates a MEM error, while TRANS indicates a compatibility problem to HiveQL.

7. Related work and technologies

In this chapter alternatives to Cloudera Impala are presented, while also comparing this thesis with other scientific works dealing with SPARQL query evaluation.

7.1. Related work

At the time of publishing this thesis and to the best of the author's knowledge, there has not been any published work on using Cloudera Impala for RDF storage and SPARQL evaluation. There are, however, publications using other Hadoop technologies and classical databases.

7.1.1. RDF in RDBMS

Sesame [34] is a generic architecture independent of the actual RDBMS, which stores an RDF graph as triple store, as discussed in section 4.1. In [22], [26], the authors present the table schemas used in the Semantic Web framework Jena2. While the first version of Jena, *Jena1*, only supported triple tables, *Jena2* can use multiple schemas, including a property table. A modified version of this approach was used for Sempala and is discussed in section 4.3 on page 41. Abadi et. al. [23] describe a vertical partitioning for RDF documents (discussed in section 4.2) and compare its performance with the property table approach. The authors claim similar performance for their approach compared with a property table approach when using RBDMS and an order of magnitude better performance for column-oriented DBMS.

The presented papers do not focus on specific SPARQL language elements but instead present more general retrieval for certain RDF patterns.

7.1.2. Hadoop-based systems

[35] and [36] are among the first publications on the matter of using Hadoop MapReduce for RDF storage and retrieval. The authors present systems to store large amounts of RDF data using commodity hardware with Hadoop. Both papers focus on basic graph pattern evaluation and omit more complex SPARQL language elements. HadoopDB [37] aims at combining the speed of a SQL database with the scalability of MapReduce and presents a hybrid of the two technologies. Inspired by HadoopDB, HadoopRDF [38] uses the combination of a traditional triple store (instead of the SQL database) and MapReduce. However, this work also only focuses on BGP evaluation.

PigSPARQL [28] has a similar approach as Sempala. Instead of translating SPARQL queries to SQL queries, PigSPARQL translates a given SPARQL query to Pig Latin, the programming language of *Apache Pig*[1]. The system supports nearly the complete SPARQL 1.0 standard. These scripts are executed by Pig as MapReduce jobs. The system stores the RDF graphs in HDFS using a vertical partitioning storage strategy. As mentioned in chapter 5, the translation of Sempala is based in part on the software code of PigSPARQL. Especially the query optimization on SPARQL algebra tree level was adopted. Further optimization, e.g. join reordering, are explained in detail in [28], as well as a detailed explanation of the implementation.

There are several approaches which are based on storing RDF data in HBase, a NoSQL database built on Hadoop to build RDF stores. JenaHBase [24] provides a combination of the Semantic Web framework Jena and HBase, in order to overcome the lack of scalability of single-machine RDF-stores based on RDBMS. [39] and [40] follow similar approaches using MapReduce and Hive, respectively in combination with HBase. [41] focuses on providing MapSide-join implementation for efficient star-query-evaluation, while RDFChain [42] contributes efficient SPARQL evaluation of chained queries (linear queries mentioned in section 4.4 on page 43). Both approaches use a combination of MapReduce and HBase. HBase is a column-oriented non-relational database system and thus shares features with Parquet. Also, Impala supports HBase as input format. Therefore, further research could investigate the presented storage schemas in HBase and compare it with Parquet.

[1] https://pig.apache.org/

7.2. Alternatives to Cloudera Impala

Presto[2] [43] is an open source distributed SQL query engine for Hadoop, which provides very similar functionality like Cloudera Impala. It was developed by Facebook and made public and open-source in November 2013. Like Impala, it only uses the Hive metastore and uses its own in-memory techniques for interactive query evaluation, without relying on MapReduce. Presto is written in Java and offers a REST API. The input data is stored on HDFS. At the moment of writing this thesis, however, Presto does not support HBase, while offering support for data stored in the NoSQL-database Cassandra.

Shark[3] also provides SQL query execution against data stored with Hadoop. It is based on Apache Spark, "a fast and general engine for large-scale data processing"[4]. Compared with Impala and Presto, it allows both in-memory and on-disk query evaluation, which according to the creators is meant to perform 100 times (in-memory) and 10 times (on-disk) faster than Hive due to the Spark-based query execution. Shark uses a columnar storage and is optimized for temporal locality, which appears similar to partitioning techniques of Impala.

The *Stinger initiative*[5] has set its goal to improve the performance of Hive, while obtaining its scalability in respect to data sizes. It consists of three phases, while by the time of this writing, the second phase has been delivered and the third phase has a preview available. In this third phase, MapReduce is replaced with Apache Tez as underlying technology for Hive [6], which is also developed by Hortonworks. Another component of the initiative is ORCFile, a columnar data format, which is meant to replace the RCFile format commonly used by Hive. Like Parquet, it provides run-length-encoding and dictionary encoding. All these technologies are part of Hortonworks' own Hadoop distribution called *Hortonworks Data Platform (HDP)*.

All in all, the introduced technologies share the idea of using Hadoop for querying Big Data with SQL, while replacing certain modules to improve performance. Columnar data format with built-in encodings, is a common performance boost factor. All the software technologies introduced here are early stage projects at the time of

[2] http://prestodb.io/
[3] http://shark.cs.berkeley.edu/
[4] https://spark.incubator.apache.org/
[5] http://hortonworks.com/labs/stinger/
[6] http://hortonworks.com/hadoop/tez/

writing, showing just how cutting-edge the idea behind Sempala is. In future, the Hive-based execution engines may present a promising alternative to Impala, if they can combine the speed of in-memory-based execution engines with the data-size scalability of on-disk execution engines. At the present time, Sempala provides an efficient complement to MapReduce-based approaches like PigSparql or Hive for queries with are rather selective, whereas these systems are more suited for long-running ETL-like queries where the scalability of MapReduce proves an advantage.

8. Summary

This thesis presents a system for efficient storage and analysis of RDF data using Cloudera Impala. Sempala, the software implemented for this thesis, provides storage of RDF data in Impala using the columnar file format Parquet and a translation of SPARQL to the SQL dialect used by Impala. As described in chapter 4, the property table approach showed the best results compared with other storage schemas for RDF documents in Impala. In this schema, each property from the RDF document corresponds to a column in the table.

During the evaluation of Sempala on a computer cluster, the following results can be summarized:

- Impala shows best performance for selective lookup queries. Hereby, nearly constant runtimes independent from the respective dataset size can be achieved.
- Unselective queries can lead to longer runtimes, which can no longer be considered as interactive runtimes. In the worst case, these queries can lead to out-of-memory-errors, as Impala 1.2.3 is limited to in-memory joins.
- Partitioning did not prove beneficiary for the runtimes.
- Working with Impala, however, also showed its young age compared with other software projects in form of software bugs. Although officially supported by Impala 1.2.3, the tables created by Impala could not be read correctly by Hive in all cases.
- On average, Impala performs one order of magnitude faster compared with Apache Hive, a data warehouse application.

8.1. Future work

As soon as Impala supports nested data types, future work could refine the property table storage schema for storing multi-value properties more efficiently without the

overhead of duplicate rows. Additionally, the evaluation of SP^{2}Bench showed that the translation was not optimal for complex queries and therefore leaves room for specific query optimization.

Related software products show a hybrid of in-memory and on-disk join evaluation. This feature is planned for Impala and may overcome the limitation of Impala, which became visible in form of out-of-memory errors during the evaluation. In future works, Impala 2.0 could be compared with other hybrid system.

All in all, Sempala provides an efficient complement to MapReduce-based approaches for selective queries at the present time.

Bibliography

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web (Berners-Lee et. al 2001)," May 2001.

[2] T. White, *Hadoop: The definitive guide, 3rd Edition.* " O'Reilly Media, Inc.", 2012.

[3] A. Chauhan, *Learning Cloudera Impala.* Packt Publishing Ltd, 2013.

[4] P. Hitzler, M. Krötzsch, S. Rudolph, and Y. Sure, *Semantic Web: Grundlagen.* Springer, 2008.

[5] N. Shadbolt, W. Hall, and T. Berners-Lee, "The semantic web revisited," *Intelligent Systems, IEEE*, vol. 21, no. 3, pp. 96 –101, jan.-feb. 2006.

[6] T. Berners-Lee, "http://www.w3.org/designissues/linkeddata.html," http://www.w3.org/DesignIssues/LinkedData.html, 2006.

[7] A. Dengel, "Linked open data, semantic web datensätze," in *Semantische Technologien.* Spektrum Akademischer Verlag, 2012, pp. 183–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-8274-2664-2_7

[8] F. Manola and E. Miller, Eds., *RDF Primer*, ser. W3C Recommendation. World Wide Web Consortium, Feb. 2004.

[9] E. Prud'hommeaux and A. Seaborne, Eds., *SPARQL Query Language for RDF*, ser. W3C Recommendation. World Wide Web Consortium, Jan. 2008.

[10] S. Harris and A. Seaborne, Eds., *SPARQL Query Language for RDF*, ser. W3C Recommendation. World Wide Web Consortium, Mar. 2013.

[11] J. Perez, M. Arenas, and C. Gutierrez, "Semantics of sparql," Technical Report TR/DCC-2006-17, Universidad de Chile, Tech. Rep., 2006.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, M. L. Scott and L. L. Peterson, Eds. ACM, 2003, pp. 29–43.

[13] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[14] "Module 4: Mapreduce," http://developer.yahoo.com/hadoop/tutorial/module4.html, Online, visited 16.02.2014.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[16] J. Russel, *Cloudera Impala*. O'Reilly Media, 2013.

[17] "Installing and using cloudera impala," http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/Installing-and-Using-Impala.html, Online, visited 02.01.2014.

[18] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[19] J. Le Dem, "Dremel made simple with parquet," https://blog.twitter.com/2013/dremel-made-simple-with-parquet, 2013.

[20] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in *Proceedings of the 2013 international conference on Management of data*. ACM, 2013, pp. 121–132.

[21] "4store, an efficient, scalable and stable rdf database," http://www.http://4store.org/, Online, visited 19.02.2014.

[22] K. Wilkinson, C. Sayers, H. A. Kuno, D. Reynolds *et al.*, "Efficient rdf storage and retrieval in jena2." in *SWDB*, vol. 3, 2003, pp. 131–150.

[23] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 411–422.

[24] V. Khadilkar, M. Kantarcioglu, B. M. Thuraisingham, and P. Castagna, "Jena-hbase: A distributed, scalable and effcient rdf triple store." in *International Semantic Web Conference (Posters & Demos)*, 2012.

[25] M. F. Husain, P. Doshi, L. Khan, and B. Thuraisingham, "Storage and retrieval of large rdf graph using hadoop and mapreduce," in *Cloud Computing*. Springer, 2009, pp. 680–686.

[26] K. Wilkinson and K. Wilkinson, "Jena property table implementation," 2006.

[27] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Sp$\hat{2}$bench: a sparql performance benchmark," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 222–233.

[28] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "Pigsparql: Mapping sparql to pig latin," in *Proceedings of the International Workshop on Semantic Web Information Management*, ser. SWIM '11. New York, NY, USA: ACM, 2011, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/1999299.1999303

[29] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for {OWL} knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, pp. 158 – 182, 2005, selcted Papers from the International Semantic Web Conference, 2004 ISWC, 2004 3rd. International Semantic Web Conference, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570826805000132

[30] D. L. McGuinness, F. Van Harmelen *et al.*, "Owl web ontology language overview," *W3C recommendation*, vol. 10, no. 2004-03, p. 10, 2004.

[31] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal, "Owl reasoning with webpie: calculating the closure of 100 billion triples," in *The Semantic Web: Research and Applications*. Springer, 2010, pp. 213–227.

[32] C. Bizer and A. Schultz, "The berlin sparql benchmark," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 2, pp. 1–24, 2009.

[33] M. Schmidt, M. Meier, and G. Lausen, "Foundations of sparql query optimization," in *Proceedings of the 13th International Conference on Database Theory*. ACM, 2010, pp. 4–33.

[34] J. Broekstra, A. Kampman, and F. Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *The Semantic Web — ISWC 2002*, ser. Lecture Notes in Computer Science, I. Horrocks and J. Hendler, Eds.

Springer Berlin Heidelberg, 2002, vol. 2342, pp. 54–68. [Online]. Available: http://dx.doi.org/10.1007/3-540-48005-6_7

[35] M. F. Husain, P. Doshi, L. Khan, and B. Thuraisingham, "Storage and retrieval of large rdf graph using hadoop and mapreduce," in *Cloud Computing*. Springer, 2009, pp. 680–686.

[36] J. Myung, J. Yeon, and S.-g. Lee, "Sparql basic graph pattern processing with iterative mapreduce," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. ACM, 2010, p. 6.

[37] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 922–933, 2009.

[38] J.-H. Du, H.-F. Wang, Y. Ni, and Y. Yu, "Hadooprdf: A scalable semantic data analytical engine," in *Intelligent Computing Theories and Applications*. Springer, 2012, pp. 633–641.

[39] J. Sun and Q. Jin, "Scalable rdf store based on hbase and mapreduce," in *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, vol. 1. IEEE, 2010, pp. V1–633.

[40] A. Haque and L. Perkins, "Distributed rdf triple store using hbase and hive," 2012.

[41] A. Schätzle, M. Przyjaciel-Zablocki, C. Dorner, T. Hornung, and G. Lausen, "Cascading map-side joins over hbase for scalable join processing," *SSWS+ HPCSW*, p. 59, 2012.

[42] P. Choi, J. Jung, and K.-H. Lee, "Rdfchain: Chain centric storage for scalable join processing of rdf graphs using mapreduce and hbase."

[43] "Prestodb a distributed sql query engine for big data coming from facebook," https://www.xtendsys.net/blog/post-1, Online, visited 18.02.2014.

A. Queries for prototype evaluation

A.1. SPARQL queries

```
SELECT *
WHERE {
  ?journal dc:title  "Journal␣1␣(1940)"^^xsd:string .
}
```

Listing A.1: Q1 SPARQL query

```
SELECT *
WHERE {
  <http://localhost/publications/journals/Journal1/1940>
    dc:title ?title .
}
```

Listing A.2: Q2 SPARQL query

```
SELECT *
WHERE {
?a rdf:type bench:Article .
}
```

Listing A.3: Q3 SPARQL query

```
SELECT *
WHERE {
?s rdf:type ?type .
?s dc:title "Journal␣1␣(1940)"^^xsd:string .
}
```

Listing A.4: Q4 SPARQL query

```
SELECT *
WHERE {
 ?subject dcterms:references ?ref.
 ?ref rdf:_1 ?it1.
 ?it1 dc:creator ?creator.
 ?creator foaf:name ?name.
}
```

Listing A.5: Q5 SPARQL query

```
SELECT *
WHERE {
  <http://localhost/publications/inprocs/Proceeding567/2001/Inproceeding33402>
    dcterms:references ?ref .
  ?ref rdf:_1 ?it1 .
  ?it1 dc:creator ?creator .
  ?creator foaf:name ?name .
}
```

Listing A.6: Q6 as SPARQL query

```
SELECT *
WHERE {
  ?subject rdf:type ?type.
  ?subject swrc:pages ?pages.
  ?subject dc:creator ?creator.
  ?subject dc:title ?title.
  ?subject swrc:month ?month.
  ?subject swrc:note ?note.
}
```

Listing A.7: Q7 as SPARQL query

```
SELECT *
WHERE {
  ?subject ?predicate
     person:Paul_Erdoes.
}
```

Listing A.8: Q9 as SPARQL query

```
SELECT *
WHERE {
  ?inproc rdf:type bench:Inproceedings.
  ?inproc dc:creator ?author.
  ?inproc bench:booktitle ?booktitle.
  ?inproc dc:title ?title.
  ?inproc dcterms:partOf ?proc.
  ?inproc rdfs:seeAlso ?ee.
  ?inproc swrc:pages ?page.
  ?inproc foaf:homepage ?url.
  ?inproc dcterms:issued ?yr.
  OPTIONAL {
    ?inproc bench:abstract ?abstract.
  }
} ORDER BY ?yr
```

Listing A.9: Q8 as SPARQL query

B. Lehigh University Benchmark (LUBM)

B.1. SPARQL queries

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
  ?X rdf:type lubm:GraduateStudent .
  ?X lubm:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> . }
```

Listing B.1: LUBM Q1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE {
  ?X rdf:type lubm:GraduateStudent .
  ?Y rdf:type lubm:University .
  ?Z rdf:type lubm:Department .
  ?X lubm:memberOf ?Z .
  ?Z lubm:subOrganizationOf ?Y .
  ?X lubm:undergraduateDegreeFrom ?Y . }
```

Listing B.2: LUBM Q2

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
  ?X rdf:type lubm:Publication .
  ?X lubm:publicationAuthor
    <http://www.Department0.University0.edu/AssistantProfessor0> . }
```

Listing B.3: LUBM Q3

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y1 ?Y2 ?Y3
WHERE {
  ?X rdf:type lubm:Professor .
  ?X lubm:worksFor <http://www.Department0.University0.edu> .
  ?X lubm:name ?Y1 .
  ?X lubm:emailAddress ?Y2 .
  ?X lubm:telephone ?Y3 . }
```

Listing B.4: LUBM Q4

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
  ?X rdf:type lubm:Person .
  ?X lubm:memberOf <http://www.Department0.University0.edu> . }
```

Listing B.5: LUBM Q5

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X WHERE {
  ?X rdf:type lubm:Student . }
```

Listing B.6: LUBM Q6

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y
WHERE {
  ?X rdf:type lubm:Student .
  ?Y rdf:type lubm:Course .
  ?X lubm:takesCourse ?Y .
  <http://www.Department0.University0.edu/AssociateProfessor0>
        lubm:teacherOf ?Y . }
```

Listing B.7: LUBM Q7

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE
{?X rdf:type lubm:Student .
  ?Y rdf:type lubm:Department .
  ?X lubm:memberOf ?Y .
  ?Y lubm:subOrganizationOf <http://www.University0.edu> .
  ?X lubm:emailAddress ?Z}
```

Listing B.8: LUBM Q8

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE
{?X rdf:type lubm:Student .
  ?Y rdf:type lubm:Faculty .
  ?Z rdf:type lubm:Course .
  ?X lubm:advisor ?Y .
  ?Y lubm:teacherOf ?Z .
  ?X lubm:takesCourse ?Z}
```

Listing B.9: LUBM Q9

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type lubm:Student .
  ?X lubm:takesCourse
<http://www.Department0.University0.edu/GraduateCourse0>}
```

Listing B.10: LUBM Q10

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type lubm:ResearchGroup .
  ?X lubm:subOrganizationOf <http://www.University0.edu>}
```

Listing B.11: LUBM Q11

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y
WHERE
{?X rdf:type lubm:Chair .
  ?Y rdf:type lubm:Department .
  ?X lubm:worksFor ?Y .
  ?Y lubm:subOrganizationOf <http://www.University0.edu> . }
```

Listing B.12: LUBM Q12

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
  ?X rdf:type lubm:Person .
  <http://www.University0.edu> lubm:hasAlumnus ?X . }
```

Listing B.13: LUBM Q13

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE { ?X rdf:type lubm:UndergraduateStudent . }
```

Listing B.14: LUBM Q14

B.2. SQL queries

```
Create Table result as (SELECT  DISTINCT BGP1_0.X AS "X"
FROM
 (SELECT  DISTINCT ID AS "X"
  FROM
  propertytable
  WHERE
  ID is not null
  AND rdf_type is not null
  AND rdf_type = 'lubm:GraduateStudent' AND lubm_takesCourse is not null
  AND lubm_takesCourse = '<http://www.Department0.University0.edu/GraduateCourse0>') BGP1_0) ;
```

Listing B.15: LUBM Q1 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1.y AS "Y",
                   BGP1.x AS "X",
                   BGP1.z AS "Z"
   FROM   (SELECT DISTINCT BGP1_0.y AS "Y",
                           BGP1_0.x AS "X",
                           BGP1_0.z AS "Z"
           FROM   (SELECT DISTINCT lubm_undergraduatedegreefrom AS "Y",
                                   id                           AS "X",
                                   lubm_memberof                AS "Z"
                   FROM   propertytable
                   WHERE  id IS NOT NULL
                          AND rdf_type IS NOT NULL
                          AND rdf_type = 'lubm:GraduateStudent'
                          AND lubm_memberof IS NOT NULL
                          AND lubm_undergraduatedegreefrom IS NOT NULL) BGP1_0
                  JOIN (SELECT DISTINCT lubm_suborganizationof AS "Y",
                                        id                     AS "Z"
                        FROM  property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Department'
                               AND lubm_suborganizationof IS NOT NULL) BGP1_1
                    ON( BGP1_0.y = BGP1_1.y
                        AND BGP1_0.z = BGP1_1.z )
                  JOIN (SELECT DISTINCT id AS "Y"
                        FROM  property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:University') BGP1_2
                    ON( BGP1_1.y = BGP1_2.y )) BGP1)
```

Listing B.16: LUBM Q2 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.x AS "X"
   FROM
     (SELECT DISTINCT id AS "X"
      FROM  property_table
      WHERE  id IS NOT NULL
       AND rdf_type IS NOT NULL
       AND rdf_type = 'lubm:Publication'
       AND lubm_publicationauthor IS NOT NULL
       AND lubm_publicationauthor =
          '<http://www.Department0.University0.edu/AssistantProfessor0>')
     BGP1_0)
```

Listing B.17: LUBM Q3 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.y3 AS "Y3",
                   BGP1_0.y1 AS "Y1",
                   BGP1_0.y2 AS "Y2",
                   BGP1_0.x  AS "X"
   FROM   (SELECT DISTINCT lubm_telephone    AS "Y3",
                           lubm_name         AS "Y1",
                           lubm_emailaddress AS "Y2",
                           id                AS "X"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'lubm:Professor'
                  AND lubm_worksfor IS NOT NULL
                  AND lubm_worksfor = '<http://www.Department0.University0.edu>'
                  AND lubm_name IS NOT NULL
                  AND lubm_emailaddress IS NOT NULL
                  AND lubm_telephone IS NOT NULL) BGP1_0);
```

Listing B.18: LUBM Q4 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.x AS "X"
   FROM   (SELECT DISTINCT id AS "X"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'lubm:Person'
                  AND lubm_memberof IS NOT NULL
                  AND lubm_memberof = '<http://www.Department0.University0.edu>'
          )
          BGP1_0);
```

Listing B.19: LUBM Q5 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.x AS "X"
   FROM   (SELECT DISTINCT id AS "X"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'lubm:Student') BGP1_0)
```

Listing B.20: LUBM Q6 (SQL)

```
CREATE TABLE result AS
CREATE TABLE result AS
  (SELECT DISTINCT BGP1.y AS "Y",
                   BGP1.x AS "X"
   FROM   (SELECT DISTINCT BGP1_0.y AS "Y",
                           BGP1_2.x AS "X"
           FROM   (SELECT DISTINCT lubm_teacherof AS "Y"
                   FROM   property_table
                   WHERE  id =
                  '<http://www.Department0.University0.edu/AssociateProfessor0>'
                          AND lubm_teacherof IS NOT NULL) BGP1_0
                  JOIN (SELECT DISTINCT lubm_takescourse AS "Y",
                                        id               AS "X"
                        FROM   property_table
                        WHERE  id IS NOT NULL
                               AND lubm_takescourse IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Student') BGP1_2
                    ON( BGP1_0.y = BGP1_2.y )
                  JOIN (SELECT DISTINCT id AS "Y"
                        FROM   property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Course') BGP1_1
                    ON( BGP1_2.y = BGP1_1.y )) BGP1)
```

Listing B.21: LUBM Q7 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1.y AS "Y",
                   BGP1.x AS "X",
                   BGP1.z AS "Z"
   FROM   (SELECT DISTINCT BGP1_0.y AS "Y",
                           BGP1_0.x AS "X",
                           BGP1_0.z AS "Z"
           FROM   (SELECT DISTINCT lubm_memberof     AS "Y",
                                   id                AS "X",
                                   lubm_emailaddress AS "Z"
                   FROM   property_table
                   WHERE  id IS NOT NULL
                          AND rdf_type IS NOT NULL
                          AND rdf_type = 'lubm:Student'
                          AND lubm_memberof IS NOT NULL
                          AND lubm_emailaddress IS NOT NULL) BGP1_0
                  JOIN (SELECT DISTINCT id AS "Y"
                        FROM   property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Department'
                               AND lubm_suborganizationof IS NOT NULL
                               AND lubm_suborganizationof =
                                   '<http://www.University0.edu>'
                       ) BGP1_1
                    ON( BGP1_0.y = BGP1_1.y )) BGP1)
```

Listing B.22: LUBM Q8 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1.y AS "Y",
                   BGP1.x AS "X",
                   BGP1.z AS "Z"
   FROM   (SELECT DISTINCT BGP1_0.y AS "Y",
                           BGP1_0.x AS "X",
                           BGP1_0.z AS "Z"
           FROM   (SELECT DISTINCT lubm_advisor     AS "Y",
                                   id               AS "X",
                                   lubm_takescourse AS "Z"
                   FROM   property_table
                   WHERE  id IS NOT NULL
                          AND rdf_type IS NOT NULL
                          AND rdf_type = 'lubm:Student'
                          AND lubm_advisor IS NOT NULL
                          AND lubm_takescourse IS NOT NULL) BGP1_0
                  JOIN (SELECT DISTINCT id             AS "Y",
                                        lubm_teacherof AS "Z"
                        FROM   property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Faculty'
                               AND lubm_teacherof IS NOT NULL) BGP1_1
                    ON( BGP1_0.y = BGP1_1.y
                        AND BGP1_0.z = BGP1_1.z )
                  JOIN (SELECT DISTINCT id AS "Z"
                        FROM   property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Course') BGP1_2
                    ON( BGP1_1.z = BGP1_2.z )) BGP1)
```

Listing B.23: LUBM Q9 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.x AS "X"
   FROM   (SELECT DISTINCT id AS "X"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'lubm:Student'
                  AND lubm_takescourse IS NOT NULL
                  AND lubm_takescourse =
                      '<http://www.Department0.University0.edu/GraduateCourse0>'
          )
          BGP1_0)
```

Listing B.24: LUBM Q10 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.x AS "X"
   FROM   (SELECT DISTINCT id AS "X"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'lubm:ResearchGroup'
                  AND lubm_suborganizationof IS NOT NULL
                  AND lubm_suborganizationof = '<http://www.University0.edu>')
          BGP1_0)
```

Listing B.25: LUBM Q11 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1.y AS "Y",
                   BGP1.x AS "X"
   FROM   (SELECT DISTINCT BGP1_0.y AS "Y",
                           BGP1_0.x AS "X"
           FROM   (SELECT DISTINCT lubm_worksfor AS "Y",
                                   id            AS "X"
                   FROM   property_table
                   WHERE  id IS NOT NULL
                          AND rdf_type IS NOT NULL
                          AND rdf_type = 'lubm:Chair'
                          AND lubm_worksfor IS NOT NULL) BGP1_0
                  JOIN (SELECT DISTINCT id AS "Y"
                        FROM   property_table
                        WHERE  id IS NOT NULL
                               AND rdf_type IS NOT NULL
                               AND rdf_type = 'lubm:Department'
                               AND lubm_suborganizationof IS NOT NULL
                               AND lubm_suborganizationof =
                                   '<http://www.University0.edu>'
                       ) BGP1_1
                    ON( BGP1_0.y = BGP1_1.y )) BGP1)
```

Listing B.26: LUBM Q12 (SQL)

```
Create Table result as (SELECT  DISTINCT BGP1.X AS "X"
 FROM
(SELECT  DISTINCT BGP1_0.X AS "X"
 FROM
(SELECT  DISTINCT ID AS "X"
 FROM
property_table
 WHERE
ID is not null  AND rdf_type is not null AND rdf_type = 'lubm:Person') BGP1_0 JOIN (SELECT  DISTINCT lubm_hasAlumnus
     AS "X"
 FROM
property_table
 WHERE
ID = '<http://www.University0.edu>' AND lubm_hasAlumnus is not null) BGP1_1 ON(BGP1_0.X=BGP1_1.X)) BGP1)
```

Listing B.27: LUBM Q13 (SQL)

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.x AS "X"
   FROM   (SELECT DISTINCT id AS "X"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'lubm:UndergraduateStudent') BGP1_0)
```

Listing B.28: LUBM Q14 (SQL)

C. BSBM queries

C.1. BSBM SPARQL queries

```
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#>
SELECT DISTINCT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product a %ProductType% .
    ?product bsbm:productFeature %ProductFeature1% .
    ?product bsbm:productFeature %ProductFeature2% .
    ?product bsbm:productPropertyNumeric1 ?value1 .
        FILTER (?value1 > %x%)
        }
ORDER BY ?label
LIMIT 10
```

Listing C.1: BSBM Q1 SPARQL query template

```
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 dc: <http://purl.org/dc/elements/1.1/>

SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1 ?propertyTextual2 ?propertyTextual3
 ?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4 ?propertyTextual5 ?propertyNumeric4
WHERE {
    %ProductXYZ% rdfs:label ?label .
    %ProductXYZ% rdfs:comment ?comment .
    %ProductXYZ% bsbm:producer ?p .
    ?p rdfs:label ?producer .
    %ProductXYZ% dc:publisher ?p .
    %ProductXYZ% bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
    %ProductXYZ% bsbm:productPropertyTextual1 ?propertyTextual1 .
    %ProductXYZ% bsbm:productPropertyTextual2 ?propertyTextual2 .
    %ProductXYZ% bsbm:productPropertyTextual3 ?propertyTextual3 .
    %ProductXYZ% bsbm:productPropertyNumeric1 ?propertyNumeric1 .
    %ProductXYZ% bsbm:productPropertyNumeric2 ?propertyNumeric2 .
    OPTIONAL { %ProductXYZ% bsbm:productPropertyTextual4 ?propertyTextual4 }
    OPTIONAL { %ProductXYZ% bsbm:productPropertyTextual5 ?propertyTextual5 }
    OPTIONAL { %ProductXYZ% bsbm:productPropertyNumeric4 ?propertyNumeric4 }
}
```

Listing C.2: BSBM Q2 SPARQL query template

```
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#>

SELECT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product a %ProductType% .
        ?product bsbm:productFeature %ProductFeature1% .
        ?product bsbm:productPropertyNumeric1 ?p1 .
        FILTER ( ?p1 > %x% )
        ?product bsbm:productPropertyNumeric3 ?p3 .
        FILTER (?p3 < %y% )
    OPTIONAL {
        ?product bsbm:productFeature %ProductFeature2% .
        ?product rdfs:label ?testVar }
    FILTER (!bound(?testVar))
}
ORDER BY ?label
LIMIT 10
```

Listing C.3: BSBM Q3 SPARQL query template

```
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#>

SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
    {
       ?product rdfs:label ?label .
       ?product rdf:type %ProductType% .
       ?product bsbm:productFeature %ProductFeature1% .
           ?product bsbm:productFeature %ProductFeature2% .
       ?product bsbm:productPropertyTextual1 ?propertyTextual .
           ?product bsbm:productPropertyNumeric1 ?p1 .
           FILTER ( ?p1 > %x% )
    } UNION {
       ?product rdfs:label ?label .
       ?product rdf:type %ProductType% .
       ?product bsbm:productFeature %ProductFeature1% .
           ?product bsbm:productFeature %ProductFeature3% .
       ?product bsbm:productPropertyTextual1 ?propertyTextual .
           ?product bsbm:productPropertyNumeric2 ?p2 .
           FILTER ( ?p2> %y% )
    }
}
ORDER BY ?label
OFFSET 5
LIMIT 10
```

Listing C.4: BSBM Q4 SPARQL query template

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>

SELECT DISTINCT ?product ?productLabel
WHERE {
        ?product rdfs:label ?productLabel .
    FILTER (%ProductXYZ% != ?product)
        %ProductXYZ% bsbm:productFeature ?prodFeature .
        ?product bsbm:productFeature ?prodFeature .
        %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
        ?product bsbm:productPropertyNumeric1 ?simProperty1 .
        FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 > (?origProperty1 - 120))
        %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
        ?product bsbm:productPropertyNumeric2 ?simProperty2 .
        FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

Listing C.5: BSBM Q5 SPARQL query template

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>

SELECT ?product ?label
WHERE {
        ?product rdfs:label ?label .
    ?product rdf:type bsbm:Product .
        FILTER regex(?label, %word1%)
}
```

Listing C.6: BSBM Q6 SPARQL query template

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review ?revTitle
       ?reviewer ?revName ?rating1 ?rating2
WHERE {
        %ProductXYZ% rdfs:label ?productLabel .
    OPTIONAL {
        ?offer bsbm:product %ProductXYZ% .
                ?offer bsbm:price ?price .
                ?offer bsbm:vendor ?vendor .
                ?vendor rdfs:label ?vendorTitle .
        ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
        ?offer dc:publisher ?vendor .
        ?offer bsbm:validTo ?date .
        FILTER (?date > %currentDate% )
    }
    OPTIONAL {
        ?review bsbm:reviewFor %ProductXYZ% .
        ?review rev:reviewer ?reviewer .
        ?reviewer foaf:name ?revName .
        ?review dc:title ?revTitle .
    OPTIONAL { ?review bsbm:rating1 ?rating1 . }
    OPTIONAL { ?review bsbm:rating2 ?rating2 . }
    }
}
```

Listing C.7: BSBM Q7 SPARQL query template

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1 ?rating2 ?rating3 ?rating4
WHERE {
        ?review bsbm:reviewFor %ProductXYZ% .
        ?review dc:title ?title .
        ?review rev:text ?text .
        FILTER langMatches( lang(?text), EN )
        ?review bsbm:reviewDate ?reviewDate .
        ?review rev:reviewer ?reviewer .
        ?reviewer foaf:name ?reviewerName .
        OPTIONAL { ?review bsbm:rating1 ?rating1 . }
        OPTIONAL { ?review bsbm:rating2 ?rating2 . }
        OPTIONAL { ?review bsbm:rating3 ?rating3 . }
        OPTIONAL { ?review bsbm:rating4 ?rating4 . }
}
ORDER BY DESC(?reviewDate)
LIMIT 20
```

Listing C.8: BSBM Q8 SPARQL query template

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?offer ?price
WHERE {
        ?offer bsbm:product %ProductXYZ% .
        ?offer bsbm:vendor ?vendor .
    ?offer dc:publisher ?vendor .
        ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
        ?offer bsbm:deliveryDays ?deliveryDays .
        FILTER (?deliveryDays <= 3)
        ?offer bsbm:price ?price .
    ?offer bsbm:validTo ?date .
    FILTER (?date > %currentDate% )
}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Listing C.9: BSBM Q10 SPARQL query template

```
SELECT ?property ?hasValue ?isValueOf
WHERE {
  { %OfferXYZ% ?property ?hasValue }
  UNION
  { ?isValueOf ?property %OfferXYZ% }
}
```

Listing C.10: BSBM Q11 SPARQL query template

C.2. BSBM SQL queries

In the following, the template variables have been replaced by actual values from the dataset by the test driver.

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1_0.product AS product , BGP1_0.label AS label
    FROM
     (SELECT DISTINCT BGP1_0_0.product AS product , BGP1_0_0.label AS label ,
                      BGP1_0_0.value1 AS value1
      FROM
        (SELECT DISTINCT ID AS product , _http___www_w3_org_2000_01_rdf_schema_label_ AS label ,
   _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ AS value1
         FROM property_table
         WHERE ID IS NOT NULL
           AND _http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ IS NOT NULL
           AND _http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
                instances/ProductType1848>'
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.fu-
                berlin.de/bizer/bsbm/v01/instances/ProductFeature43883>'
           AND _http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ IS NOT NULL)
                BGP1_0_0
      JOIN
        (SELECT DISTINCT ID AS product
         FROM property_table
         WHERE ID IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.fu-
                berlin.de/bizer/bsbm/v01/instances/ProductFeature7746>') BGP1_0_1 ON(BGP1_0_0.product=BGP1_0_1.
                product)
      WHERE (BGP1_0_0.value1 > 136)
      ORDER BY label LIMIT 100000000) BGP1_0 LIMIT 10) ;
```

Listing C.11: BSBM Q1 as SQL

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1.propertytextual4 AS "propertyTextual4", BGP1.propertytextual5 AS "propertyTextual5",
                   BGP1.propertytextual2 AS "propertyTextual2", BGP1.propertytextual3 AS "propertyTextual3",
                   BGP1.propertytextual1 AS "propertyTextual1", BGP1.productfeature   AS "productFeature",
                   BGP1.producer         AS "producer", BGP1.propertynumeric4 AS "propertyNumeric4",
                   BGP1.label            AS "label", BGP1.propertynumeric2 AS "propertyNumeric2",
                   BGP1.comme            AS "comme", BGP1.propertynumeric1 AS "propertyNumeric1"
   FROM   (SELECT DISTINCT BGP1_0.f               AS "f",
                           BGP1_0.propertytextual4 AS "propertyTextual4",
                           BGP1_0.propertytextual5 AS "propertyTextual5",
                           BGP1_0.propertytextual2 AS "propertyTextual2",
                           BGP1_0.propertytextual3 AS "propertyTextual3",
                           BGP1_0.propertytextual1 AS "propertyTextual1",
                           BGP1_2.productfeature   AS "productFeature",
                           BGP1_1.p                AS "p",
                           BGP1_1.producer         AS "producer",
                           BGP1_0.label            AS "label",
                           BGP1_0.propertynumeric4 AS "propertyNumeric4",
                           BGP1_0.comme            AS "comme",
                           BGP1_0.propertynumeric2 AS "propertyNumeric2",
                           BGP1_0.propertynumeric1 AS "propertyNumeric1"
           FROM   (SELECT DISTINCT id
                                   AS
                                   "p",
                                   http___www_w3_org_2000_01_rdf_schema_label_
                                   AS
                                   "producer"
                   FROM   property_table
                   WHERE  id IS NOT NULL
                          AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT
                              NULL)
                  BGP1_1
                  JOIN (SELECT DISTINCT
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual4_ AS "propertyTextual4",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productfeature_ AS "f",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual5_ AS "propertyTextual5",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual2_ AS "propertyTextual2",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual3_ AS "propertyTextual3",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual1_ AS "propertyTextual1",
http___purl_org_dc_elements_1_1_publisher_                                          AS "p",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertynumeric4_
                                                                                    AS "propertyNumeric4",
http___www_w3_org_2000_01_rdf_schema_label_                                         AS "label",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertynumeric2_ AS "propertyNumeric2",
http___www_w3_org_2000_01_rdf_schema_comment_ AS "comme",
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertynumeric1_ AS "propertyNumeric1"
 FROM   property_table
 WHERE  id = '$prod'
        AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
        AND http___www_w3_org_2000_01_rdf_schema_comment_ IS NOT NULL
        AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_producer_
            IS NOT
            NULL
        AND http___purl_org_dc_elements_1_1_publisher_ IS NOT NULL
        AND
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productfeature_ IS NOT NULL
        AND
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual1 IS NOT NULL
AND
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual2_ IS NOT NULL
AND
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertytextual3_ IS NOT NULL
AND
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertynumeric1_ IS NOT NULL
AND
http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productpropertynumeric2_ IS NOT NULL) BGP1_0
  ON( BGP1_1.p = BGP1_0.p )
JOIN (SELECT DISTINCT id                                          AS "f",
                      http___www_w3_org_2000_01_rdf_schema_label_ AS "productFeature"
      FROM   property_table
      WHERE  id IS NOT NULL
             AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL) BGP1_2
  ON( BGP1_0.f = BGP1_2.f )) BGP1);
```

Listing C.12: BSBM Q2 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT OPTIONAL0.product AS product ,
                   OPTIONAL0.label AS label
   FROM
     (SELECT DISTINCT BGP1_0.product AS product ,
                      BGP1_0.p3 AS p3 ,
                      BGP1_0.p1 AS p1 ,
                      BGP2_0.testVar AS testVar ,
                      BGP1_0.label AS label
      FROM
        (SELECT DISTINCT ID AS product ,
             _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric3_ AS p3 ,
             _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ AS p1 ,
             _http___www_w3_org_2000_01_rdf_schema_label_ AS label
         FROM property_table
         WHERE ID IS NOT NULL
           AND _http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ IS NOT NULL
           AND _http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
                instances/ProductType3082>'
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.fu-
                berlin.de/bizer/bsbm/v01/instances/ProductFeature95>'
           AND _http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ IS NOT NULL
           AND _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric3_ IS NOT NULL) BGP1_0
      LEFT JOIN
        (SELECT DISTINCT ID AS product , _http___www_w3_org_2000_01_rdf_schema_label_ AS testVar
         FROM property_table
         WHERE _http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.fu-
              berlin.de/bizer/bsbm/v01/instances/ProductFeature96>') BGP2_0 ON(BGP1_0.product=BGP2_0.product)
      WHERE (BGP1_0.p1 > 132)
        AND (BGP1_0.p3 < 159)
        AND BGP2_0.testVar IS NULL
      ORDER BY label LIMIT 10) OPTIONAL0) ;
```

Listing C.13: BSBM Q3 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT UNION1.product AS "product" ,
                   UNION1.label AS "label" ,
                   UNION1.propertyTextual AS "propertyTextual"
   FROM (
           (SELECT DISTINCT BGP1_0_0.product AS "product" ,
                            NULL AS "p2" ,
                            BGP1_0_0.p1 AS "p1" ,
                            BGP1_0_0.label AS "label" ,
                            BGP1_0_0.propertyTextual AS "propertyTextual"
            FROM
              (SELECT DISTINCT ID AS "product" ,
                               http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ AS "
                                    p1" ,
                               http___www_w3_org_2000_01_rdf_schema_label_ AS "label" ,
                               http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyTextual1_ AS "
                                    propertyTextual"
               FROM bigtable_parquet
               WHERE ID IS NOT NULL
                 AND http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ IS NOT NULL
                 AND http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
                      v01/instances/ProductType1665>'
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.
                      fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature545>'
                 AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyTextual1_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ IS NOT NULL)
                      BGP1_0_0
            JOIN
              (SELECT DISTINCT ID AS "product"
               FROM bigtable_parquet
               WHERE ID IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.
                      fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature39586>') BGP1_0_1 ON(BGP1_0_0.product=
                      BGP1_0_1.product)
            WHERE (BGP1_0_0.p1 > 278)
            ORDER BY label LIMIT 100000000)
         UNION
           (SELECT DISTINCT BGP2_0_0.product AS "product" ,
                            BGP2_0_0.p2 AS "p2" ,
                            NULL AS "p1" ,
                            BGP2_0_0.label AS "label" ,
                            BGP2_0_0.propertyTextual AS "propertyTextual"
            FROM
              (SELECT DISTINCT ID AS "product" ,
                               http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric2_ AS "
                                    p2" ,
                               http___www_w3_org_2000_01_rdf_schema_label_ AS "label" ,
                               http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyTextual1_ AS "
                                    propertyTextual"
               FROM bigtable_parquet
               WHERE ID IS NOT NULL
                 AND http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ IS NOT NULL
                 AND http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
                      v01/instances/ProductType1665>'
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.
                      fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature545>'
                 AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyTextual1_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric2_ IS NOT NULL)
                      BGP2_0_0
            JOIN
              (SELECT DISTINCT ID AS "product"
               FROM bigtable_parquet
               WHERE ID IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ = '<http://www4.wiwiss.
                      fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature39573>') BGP2_0_1 ON(BGP2_0_0.product=
                      BGP2_0_1.product)
            WHERE (BGP2_0_0.p2 > 53)
            ORDER BY label LIMIT 100000000)) UNION1 LIMIT 10) ;
```

Listing C.14: BSBM Q4 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1.product AS "product" ,
                   BGP1.productLabel AS "productLabel"
   FROM
     (SELECT DISTINCT BGP1_0.product AS "product" ,
                      BGP1_0.simProperty2 AS "simProperty2" ,
                      BGP1_1.origProperty1 AS "origProperty1" ,
                      BGP1_0.simProperty1 AS "simProperty1" ,
                      BGP1_1.origProperty2 AS "origProperty2" ,
                      BGP1_0.productLabel AS "productLabel" ,
                      BGP1_1.prodFeature AS "prodFeature"
      FROM
        (SELECT DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ AS "
             origProperty1" ,
                         http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric2_ AS "
                              origProperty2" ,
                         http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ AS "prodFeature"
         FROM bigtable_parquet
         WHERE ID = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer17999/Product909534>'
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric2_ IS NOT NULL) BGP1_1
      JOIN
        (SELECT DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric2_ AS "
             simProperty2" ,
                         ID AS "product" ,
                         http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ AS "
                              simProperty1" ,
                         http___www_w3_org_2000_01_rdf_schema_label_ AS "productLabel" ,
                         http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ AS "prodFeature"
         FROM bigtable_parquet
         WHERE ID IS NOT NULL
           AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productFeature_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric1_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_productPropertyNumeric2_ IS NOT NULL) BGP1_0
               ON(BGP1_1.prodFeature=BGP1_0.prodFeature)
      WHERE ('<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer17999/Product909534>' !=
           BGP1_0.product)
        AND ((BGP1_0.simProperty1 < (BGP1_1.origProperty1+120))
             AND (BGP1_0.simProperty1 > (BGP1_1.origProperty1-120)))
        AND ((BGP1_0.simProperty2 < (BGP1_1.origProperty2+170))
             AND (BGP1_0.simProperty2 > (BGP1_1.origProperty2-170)))
      ORDER BY productLabel LIMIT 100000000) BGP1 LIMIT 5) ;
```

Listing C.15: BSBM Q5 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT SEQUENCE_JOIN1.product AS "product" ,
                   SEQUENCE_JOIN1.label AS "label"
   FROM
     (SELECT DISTINCT BGP1_0.product AS "product" ,
                      BGP1_0.label AS "label"
      FROM
        (SELECT DISTINCT ID AS "product" ,
                         http___www_w3_org_2000_01_rdf_schema_label_ AS "label"
         FROM bigtable_parquet
         WHERE ID IS NOT NULL
           AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL
           AND (http___www_w3_org_2000_01_rdf_schema_label_ LIKE '"wisher"')) BGP1_0
      JOIN
        (SELECT DISTINCT ID AS "product"
         FROM bigtable_parquet
         WHERE ID IS NOT NULL
           AND http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ IS NOT NULL
           AND http___www_w3_org_1999_02_22_rdf_syntax_ns_type_ = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
                vocabulary/Product>') BGP2_0 ON(BGP1_0.product=BGP2_0.product)) SEQUENCE_JOIN1) ;
```

Listing C.16: BSBM Q6 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT OPTIONAL3.revName AS "revName", OPTIONAL3.price AS "price" ,
                   OPTIONAL3.vendor AS "vendor", OPTIONAL3.rating1 AS "rating1" ,
                   OPTIONAL3.rating2 AS "rating2", OPTIONAL3.reviewer AS "reviewer" ,
                   OPTIONAL3.vendorTitle AS "vendorTitle", OPTIONAL3.productLabel AS "productLabel" ,
                   OPTIONAL3.offer AS "offer", OPTIONAL3.revTitle AS "revTitle", OPTIONAL3.review AS "review"
   FROM
     (SELECT DISTINCT OPTIONAL2.revName AS "revName", OPTIONAL0.price AS "price", OPTIONAL0.vendor AS "vendor",
                      OPTIONAL2.rating1 AS "rating1", OPTIONAL2.rating2 AS "rating2",
                      OPTIONAL2.reviewer AS "reviewer", OPTIONAL0.productLabel AS "productLabel",
                      OPTIONAL0.vendorTitle AS "vendorTitle", OPTIONAL0.offer AS "offer" ,
                      OPTIONAL0.dat AS "dat", OPTIONAL2.revTitle AS "revTitle", OPTIONAL2.review AS "review"
      FROM
        (SELECT DISTINCT BGP2.price AS "price", BGP2.vendor AS "vendor", BGP2.vendorTitle AS "vendorTitle" ,
                         BGP1_0.productLabel AS "productLabel", BGP2.offer AS "offer", BGP2.dat AS "dat"
         FROM
           (SELECT DISTINCT http___www_w3_org_2000_01_rdf_schema_label_ AS "productLabel"
            FROM bigtable_parquet
            WHERE ID = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer4671/Product236408>'
              AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL) BGP1_0
         CROSS JOIN
           (SELECT DISTINCT BGP2_0.price AS "price", BGP2_1.vendor AS "vendor", BGP2_1.vendorTitle AS "vendorTitle" ,
                            BGP2_0.offer AS "offer", BGP2_0.dat AS "dat"
            FROM
              (SELECT DISTINCT ID AS "vendor" ,
                               http___www_w3_org_2000_01_rdf_schema_label_ AS "vendorTitle"
               FROM bigtable_parquet
               WHERE ID IS NOT NULL AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_country_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_country_ = '<http://downlode.org/rdf/
                      iso-3166/countries#DE>'
                 AND http___www_w3_org_2000_01_rdf_schema_label_ IS NOT NULL) BGP2_1
            JOIN
              (SELECT DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_price_ AS "price" ,
                               http___purl_org_dc_elements_1_1_publisher_ AS "vendor" ,
                               ID AS "offer" ,
                               http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_validTo_ AS "dat"
               FROM bigtable_parquet
               WHERE ID IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_product_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_product_ = '<http://www4.wiwiss.fu-
                      berlin.de/bizer/bsbm/v01/instances/dataFromProducer4671/Product236408>'
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_price_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_vendor_ IS NOT NULL
                 AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_validTo_ IS NOT NULL
                 AND http___purl_org_dc_elements_1_1_publisher_ IS NOT NULL) BGP2_0 ON(BGP2_1.vendor=BGP2_0.vendor))
                      BGP2
         WHERE (BGP2.dat > '"2008-06-20T00:00:00"^^<http://www.w3.org/2001/XMLSchema#datTime>')) OPTIONAL0
      CROSS JOIN
```

```
(SELECT DISTINCT OPTIONAL1.revName AS "revName", OPTIONAL1.rating1 AS "rating1", BGP5_0.rating2 AS "rating2" ,
                OPTIONAL1.reviewer AS "reviewer", OPTIONAL1.revTitle AS "revTitle", OPTIONAL1.review AS "
                     review"
 FROM
   (SELECT DISTINCT BGP3.revName AS "revName", BGP4_0.rating1 AS "rating1", BGP3.reviewer AS "reviewer" ,
                   BGP3.revTitle AS "revTitle", BGP3.review AS "review"
    FROM
      (SELECT DISTINCT BGP3_1.revName AS "revName", BGP3_1.reviewer AS "reviewer", BGP3_0.revTitle AS "
           revTitle" ,
                    BGP3_0.review AS "review"
       FROM
         (SELECT DISTINCT http___xmlns_com_foaf_0_1_name_ AS "revName" ,
                         ID AS "reviewer"
          FROM bigtable_parquet
          WHERE ID IS NOT NULL
            AND http___xmlns_com_foaf_0_1_name_ IS NOT NULL) BGP3_1
       JOIN
         (SELECT DISTINCT http___purl_org_stuff_rev_reviewer_ AS "reviewer",
              http___purl_org_dc_elements_1_1_title_ AS "revTitle",
                         ID AS "review"
          FROM bigtable_parquet
          WHERE ID IS NOT NULL
            AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_reviewFor_ IS NOT NULL
            AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_reviewFor_ =
                       '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer4671/
                            Product236408>'
            AND http___purl_org_stuff_rev_reviewer_ IS NOT NULL
            AND http___purl_org_dc_elements_1_1_title_ IS NOT NULL) BGP3_0 ON(BGP3_1.reviewer=BGP3_0.reviewer)
                 ) BGP3
    LEFT JOIN
      (SELECT DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_rating1_ AS "rating1" ,
                      ID AS "review"
       FROM bigtable_parquet) BGP4_0 ON(BGP3.review=BGP4_0.review)) OPTIONAL1
 LEFT JOIN
   (SELECT DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_rating2_ AS "rating2" ,
                   ID AS "review"
    FROM bigtable_parquet) BGP5_0 ON(OPTIONAL1.review=BGP5_0.review)) OPTIONAL2) OPTIONAL3) ;
```

Listing C.17: BSBM Q7 as SQL

```
 Create Table result as (SELECT  DISTINCT OPTIONAL3.text AS "text" , OPTIONAL3.title AS "title" , OPTIONAL3.rating1 AS
       "rating1" , OPTIONAL3.rating2 AS "rating2" , OPTIONAL3.reviewerName AS "reviewerName" , OPTIONAL3.reviewDate
      AS "reviewDate" , OPTIONAL3.rating3 AS "rating3" , OPTIONAL3.reviewer AS "reviewer" , OPTIONAL3.rating4 AS "
      rating4"
 FROM
(SELECT  DISTINCT OPTIONAL2.title AS "title" , OPTIONAL2.text AS "text" , OPTIONAL2.rating1 AS "rating1" , OPTIONAL2.
     rating2 AS "rating2" , OPTIONAL2.reviewDate AS "reviewDate" , OPTIONAL2.reviewerName AS "reviewerName" ,
     OPTIONAL2.reviewer AS "reviewer" , OPTIONAL2.rating3 AS "rating3" , BGP5_0.rating4 AS "rating4" , OPTIONAL2.
     review AS "review"
 FROM
(SELECT  DISTINCT OPTIONAL1.text AS "text" , OPTIONAL1.title AS "title" , OPTIONAL1.rating1 AS "rating1" , OPTIONAL1.
     reviewerName AS "reviewerName" , OPTIONAL1.reviewDate AS "reviewDate" , OPTIONAL1.rating2 AS "rating2" , BGP4_0.
     rating3 AS "rating3" , OPTIONAL1.reviewer AS "reviewer" , OPTIONAL1.review AS "review"
 FROM
(SELECT  DISTINCT OPTIONAL0.title AS "title" , OPTIONAL0.text AS "text" , OPTIONAL0.rating1 AS "rating1" , BGP3_0.
     rating2 AS "rating2" , OPTIONAL0.reviewDate AS "reviewDate" , OPTIONAL0.reviewerName AS "reviewerName" ,
     OPTIONAL0.reviewer AS "reviewer" , OPTIONAL0.review AS "review"
 FROM
(SELECT  DISTINCT BGP1.text AS "text" , BGP1.title AS "title" , BGP2_0.rating1 AS "rating1" , BGP1.reviewerName AS "
     reviewerName" , BGP1.reviewDate AS "reviewDate" , BGP1.reviewer AS "reviewer" , BGP1.review AS "review"
 FROM
(SELECT  DISTINCT BGP1_0.title AS "title" , BGP1_0.text AS "text" , BGP1_0.reviewDate AS "reviewDate" , BGP1_1.
     reviewerName AS "reviewerName" , BGP1_1.reviewer AS "reviewer" , BGP1_0.review AS "review"
 FROM
(SELECT  DISTINCT http___xmlns_com_foaf_0_1_name_ AS "reviewerName" , ID AS "reviewer"
 FROM
bigtable_parquet
 WHERE
ID is not null  AND http___xmlns_com_foaf_0_1_name_ is not null) BGP1_1 JOIN (SELECT  DISTINCT
     http___purl_org_stuff_rev_text_ AS "text" , http___purl_org_dc_elements_1_1_title_ AS "title" ,
     http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_reviewDate_ AS "reviewDate" ,
     http___purl_org_stuff_rev_reviewer_ AS "reviewer" , ID AS "review"
 FROM
bigtable_parquet
 WHERE
ID is not null  AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_reviewFor_ is not null AND
     http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_reviewFor_ = '<http://www4.wiwiss.fu-berlin.de/bizer/
     bsbm/v01/instances/dataFromProducer6375/Product323755>' AND http___purl_org_dc_elements_1_1_title_ is not null
     AND http___purl_org_stuff_rev_text_ is not null AND
     http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_reviewDate_ is not null AND
     http___purl_org_stuff_rev_reviewer_ is not null) BGP1_0 ON(BGP1_1.reviewer=BGP1_0.reviewer)) BGP1 LEFT JOIN (
     SELECT  DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_rating1_ AS "rating1" , ID AS "review
     "
 FROM
bigtable_parquet) BGP2_0 ON(BGP1.review=BGP2_0.review)) OPTIONAL0 LEFT JOIN (SELECT  DISTINCT
     http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_rating2_ AS "rating2" , ID AS "review"
 FROM
bigtable_parquet) BGP3_0 ON(OPTIONAL0.review=BGP3_0.review)) OPTIONAL1 LEFT JOIN (SELECT  DISTINCT
     http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_rating3_ AS "rating3" , ID AS "review"
 FROM
bigtable_parquet) BGP4_0 ON(OPTIONAL1.review=BGP4_0.review)) OPTIONAL2 LEFT JOIN (SELECT  DISTINCT
     http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_rating4_ AS "rating4" , ID AS "review"
 FROM
bigtable_parquet) BGP5_0 ON(OPTIONAL2.review=BGP5_0.review)
 WHERE
(OPTIONAL2.text LIKE '%@EN')
 ORDER BY reviewDate DESC
 LIMIT 20) OPTIONAL3) ;
```

Listing C.18: BSBM Q8 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1.price AS "price" ,
                   BGP1.offer AS "offer"
   FROM
     (SELECT DISTINCT BGP1_0.price AS "price" ,
                      BGP1_1.vendor AS "vendor" ,
                      BGP1_0.offer AS "offer" ,
                      BGP1_0.dat AS "dat" ,
                      BGP1_0.deliveryDays AS "deliveryDays"
      FROM
        (SELECT DISTINCT ID AS "vendor"
         FROM property_table
         WHERE ID IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_country_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_country_ = '<http://downlode.org/rdf/iso
                -3166/countries#US>') BGP1_1
      JOIN
        (SELECT DISTINCT http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_price_ AS "price" ,
                         http___purl_org_dc_elements_1_1_publisher_ AS "vendor" ,
                         ID AS "offer" ,
                         http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_validTo_ AS "dat" ,
                         http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_deliveryDays_ AS "deliveryDays"
         FROM property_table
         WHERE ID IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_product_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_product_ = '<http://www4.wiwiss.fu-berlin.de/
                bizer/bsbm/v01/instances/dataFromProducer32217/Product1632131>'
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_vendor_ IS NOT NULL
           AND http___purl_org_dc_elements_1_1_publisher_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_deliveryDays_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_price_ IS NOT NULL
           AND http___www4_wiwiss_fu_berlin_de_bizer_bsbm_v01_vocabulary_validTo_ IS NOT NULL) BGP1_0 ON(BGP1_1.vendor
                =BGP1_0.vendor)
      WHERE (BGP1_0.deliveryDays <= 3)
        AND (BGP1_0.dat > '"2008-06-20T00:00:00"^^xsd:dateTime')
      ORDER BY BGP1_0.price LIMIT 100000000) BGP1 LIMIT 10) ;
```

Listing C.19: BSBM Q10 as SQL

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT UNION1.hasValue AS "hasValue" ,
                   UNION1.property AS "property" ,
                   UNION1.isValueOf AS "isValueOf"
   FROM (
           (SELECT DISTINCT OBJECT AS "hasValue" ,
                                    predicate AS "property" ,
                                    NULL AS "isValueOf"
            FROM triplestore_parquet
            WHERE ID = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromVendor8783/Offer17422713>')
         UNION
           (SELECT DISTINCT NULL AS "hasValue" ,
                            predicate AS "property" ,
                            ID AS "isValueOf"
            FROM triplestore_parquet
            WHERE ID IS NOT NULL
              AND OBJECT = '<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromVendor8783/Offer17422713
                   >')) UNION1) ;
```

Listing C.20: BSBM Q11 as SQL

D. SP^{2}Bench queries

D.1. SP^{2}Bench SPARQL queries

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX bench:   <http://localhost/vocabulary/bench/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title "Journal␣1␣(1940)"^^xsd:string .
  ?journal dcterms:issued ?yr
}
```

Listing D.1: SP^{2}Bench Q1

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc:    <http://swrc.ontoware.org/ontology#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX bench:   <http://localhost/vocabulary/bench/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL { ?inproc bench:abstract ?abstract } }
ORDER BY ?yr
```

Listing D.2: SP^{2}Bench Q2

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX swrc:  <http://swrc.ontoware.org/ontology#>

SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:pages) }
```

Listing D.3: SP²Bench Q3a (Q3b and Q3c similar)

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bench:   <http://localhost/vocabulary/bench/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX swrc:    <http://swrc.ontoware.org/ontology#>

SELECT DISTINCT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article .
  ?article2 rdf:type bench:Article .
  ?article1 dc:creator ?author1 .
  ?author1 foaf:name ?name1 .
  ?article2 dc:creator ?author2 .
  ?author2 foaf:name ?name2 .
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
 FILTER (?name1<?name2) }
```

Listing D.4: SP²Bench Q4

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX dc:    <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person2 .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name2
  FILTER (?name=?name2) }
```

Listing D.5: SP²Bench Q5a

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX dc:    <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name }
```

Listing D.6: SP²Bench Q5b

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?yr ?name ?document
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
  ?document dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 rdf:type ?class2 .
    ?document2 dcterms:issued ?yr2 .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2)) }
```

Listing D.7: SP²Bench Q6

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc rdf:type ?class .
  ?doc dc:title ?title .
  ?bag2 ?member2 ?doc .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?doc3 rdf:type ?class3 .
    ?doc3 dcterms:references ?bag3 .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?doc4 rdf:type ?class4 .
      ?doc4 dcterms:references ?bag4 .
      ?bag4 ?member4 ?doc3
    } FILTER (!bound(?doc4))
  } FILTER (!bound(?doc3)) }
```

Listing D.8: SP²Bench Q7

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?predicate
WHERE {
  {
    ?person rdf:type foaf:Person .
    ?subject ?predicate ?person
  } UNION {
    ?person rdf:type foaf:Person .
    ?person ?predicate ?object
  } }
```

Listing D.9: SP²Bench Q9

```
PREFIX person: <http://localhost/persons/>

SELECT ?subject ?predicate
WHERE {
  ?subject ?predicate <http://localhost/persons/Paul_Erdoes>
}
```

Listing D.10: SP²Bench Q10

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?ee
WHERE {
  ?publication rdfs:seeAlso ?ee
}
ORDER BY ?ee
LIMIT 10
OFFSET 50
```

Listing D.11: SP²Bench Q11

D.2. SP²Bench SQL queries

```
CREATE TABLE result AS
  (SELECT DISTINCT BGP1_0.yr AS "yr"
   FROM   (SELECT DISTINCT id             AS "journal",
                           dcterms_issued AS "yr"
           FROM   property_table
           WHERE  id IS NOT NULL
                  AND rdf_type IS NOT NULL
                  AND rdf_type = 'bench:Journal'
                  AND dc_title IS NOT NULL
                  AND dc_title = 'Journal␣1␣(1940)'
                  AND dcterms_issued IS NOT NULL) BGP1_0);
```

Listing D.12: SP²Bench Q1 (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT OPTIONAL0.abstract AS "abstract" ,
                   OPTIONAL0.author AS "author" ,
                   OPTIONAL0.title AS "title" ,
                   OPTIONAL0.ee AS "ee" ,
                   OPTIONAL0.page AS "page" ,
                   OPTIONAL0.yr AS "yr" ,
                   OPTIONAL0.booktitle AS "booktitle" ,
                   OPTIONAL0.inproc AS "inproc" ,
                   OPTIONAL0.url AS "url" ,
                   OPTIONAL0.proc AS "proc"
   FROM
     (SELECT DISTINCT BGP2_0.abstract AS "abstract" ,
                      BGP1_0.author AS "author" ,
                      BGP1_0.title AS "title" ,
                      BGP1_0.ee AS "ee" ,
                      BGP1_0.page AS "page" ,
                      BGP1_0.yr AS "yr" ,
                      BGP1_0.booktitle AS "booktitle" ,
                      BGP1_0.inproc AS "inproc" ,
                      BGP1_0.proc AS "proc" ,
                      BGP1_0.url AS "url"
      FROM
        (SELECT DISTINCT dc_creator AS "author" ,
                         dc_title AS "title" ,
                         rdfs_seeAlso AS "ee" ,
                         swrc_pages AS "page" ,
                         dcterms_issued AS "yr" ,
                         bench_booktitle AS "booktitle" ,
                         ID AS "inproc" ,
                         foaf_homepage AS "url" ,
                         dcterms_partOf AS "proc"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdf_type IS NOT NULL
           AND rdf_type = 'bench:Inproceedings'
           AND dc_creator IS NOT NULL
           AND bench_booktitle IS NOT NULL
           AND dc_title IS NOT NULL
           AND dcterms_partOf IS NOT NULL
           AND rdfs_seeAlso IS NOT NULL
           AND swrc_pages IS NOT NULL
           AND foaf_homepage IS NOT NULL
           AND dcterms_issued IS NOT NULL) BGP1_0
      LEFT JOIN
        (SELECT DISTINCT bench_abstract AS "abstract" ,
                         ID AS "inproc"
         FROM property_table) BGP2_0 ON(BGP1_0.inproc=BGP2_0.inproc)
      ORDER BY yr LIMIT 100000000) OPTIONAL0) ;
```

Listing D.13: SP²Bench Q2 (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1_0.article AS "article"
   FROM
     (SELECT DISTINCT ID AS "article" ,
                      swrc_pages AS "value"
      FROM property_table
      WHERE ID IS NOT NULL
        AND rdf_type IS NOT NULL
        AND rdf_type = 'bench:Article'
        AND swrc_pages IS NOT NULL) BGP1_0) ;
```

Listing D.14: SP²Bench Q3a (SQL)

```
Drop table result; Create Table result as (SELECT  DISTINCT BGP1.name1 AS "name1" , BGP1.name2 AS "name2"
 FROM
(SELECT  DISTINCT BGP1_0.journal AS "journal" , BGP1_2.article2 AS "article2" , BGP1_2.author2 AS "author2" , BGP1_1.
     author1 AS "author1" , BGP1_1.name1 AS "name1" , BGP1_3.name2 AS "name2" , BGP1_0.article1 AS "article1"
 FROM
(SELECT  DISTINCT ID AS "author1" , foaf_name AS "name1"
 FROM
property_table
 WHERE
ID is not null  AND foaf_name is not null) BGP1_1 JOIN (SELECT  DISTINCT swrc_journal AS "journal" , dc_creator AS "
     author1" , ID AS "article1"
 FROM
property_table
 WHERE
ID is not null  AND rdf_type is not null AND rdf_type = 'bench:Article' AND dc_creator is not null AND swrc_journal is
      not null) BGP1_0 ON(BGP1_1.author1=BGP1_0.author1) JOIN (SELECT  DISTINCT swrc_journal AS "journal" , ID AS "
     article2" , dc_creator AS "author2"
 FROM
property_table
 WHERE
ID is not null  AND swrc_journal is not null AND rdf_type is not null AND rdf_type = 'bench:Article' AND dc_creator is
      not null) BGP1_2 ON(BGP1_0.journal=BGP1_2.journal) JOIN (SELECT  DISTINCT ID AS "author2" , foaf_name AS "name2
     "
 FROM
property_table
 WHERE
ID is not null  AND foaf_name is not null) BGP1_3 ON(BGP1_2.author2=BGP1_3.author2)
 WHERE
(BGP1_1.name1 < BGP1_3.name2)) BGP1) ;
```

Listing D.15: SP²Bench Q4 (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1.person AS "person" ,
                   BGP1.name AS "name"
   FROM
     (SELECT DISTINCT BGP1_1.person AS "person" ,
                      BGP1_3.person2 AS "person2" ,
                      BGP1_0.article AS "article" ,
                      BGP1_2.name AS "name" ,
                      BGP1_3.inproc AS "inproc"
      FROM
        (SELECT DISTINCT dc_creator AS "person2" ,
                         ID AS "inproc"
         FROM property_table
         WHERE ID IS NOT NULL
           AND dc_creator IS NOT NULL
           AND rdf_type IS NOT NULL
           AND rdf_type = 'bench:Inproceedings') BGP1_3
      JOIN
        (SELECT DISTINCT ID AS "person2" ,
                         foaf_name AS "name"
         FROM property_table
         WHERE ID IS NOT NULL
           AND foaf_name IS NOT NULL) BGP1_2 ON(BGP1_3.person2=BGP1_2.person2)
      JOIN
        (SELECT DISTINCT ID AS "person" ,
                         foaf_name AS "name"
         FROM property_table
         WHERE ID IS NOT NULL
           AND foaf_name IS NOT NULL) BGP1_1 ON(BGP1_2.name=BGP1_1.name)
      JOIN
        (SELECT DISTINCT dc_creator AS "person" ,
                         ID AS "article"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdf_type IS NOT NULL
           AND rdf_type = 'bench:Article'
           AND dc_creator IS NOT NULL) BGP1_0 ON(BGP1_1.person=BGP1_0.person)) BGP1) ;
```

Listing D.16: SP²Bench Q5a (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1.person AS "person" ,
                   BGP1.name AS "name"
   FROM
     (SELECT DISTINCT BGP1_2.person AS "person" ,
                      BGP1_0.article AS "article" ,
                      BGP1_1.name AS "name" ,
                      BGP1_2.inproc AS "inproc"
      FROM
        (SELECT DISTINCT dc_creator AS "person" ,
                         ID AS "inproc"
         FROM property_table
         WHERE ID IS NOT NULL
           AND dc_creator IS NOT NULL
           AND rdf_type IS NOT NULL
           AND rdf_type = 'bench:Inproceedings') BGP1_2
      JOIN
        (SELECT DISTINCT ID AS "person" ,
                         foaf_name AS "name"
         FROM property_table
         WHERE ID IS NOT NULL
           AND foaf_name IS NOT NULL) BGP1_1 ON(BGP1_2.person=BGP1_1.person)
      JOIN
        (SELECT DISTINCT dc_creator AS "person" ,
                         ID AS "article"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdf_type IS NOT NULL
           AND rdf_type = 'bench:Article'
           AND dc_creator IS NOT NULL) BGP1_0 ON(BGP1_1.person=BGP1_0.person)) BGP1) ;
```

Listing D.17: SP²Bench Q5b (SQL)

```
CREATE TABLE a AS
  (SELECT DISTINCT BGP1.author AS "author" ,
                   BGP1.document AS "document" ,
                   BGP2.class2 AS "class2" ,
                   BGP1.yr AS "yr" ,
                   BGP2.document2 AS "document2" ,
                   BGP1.name AS "name" ,
                   BGP1.class AS "class" ,
                   BGP2.author2 AS "author2" ,
                   BGP2.yr2 AS "yr2"
   FROM
     (SELECT DISTINCT BGP1_1.author AS "author" ,
                      BGP1_1.document AS "document" ,
                      BGP1_1.yr AS "yr" ,
                      BGP1_2.name AS "name" ,
                      BGP1_1.class AS "class"
      FROM
        (SELECT DISTINCT dc_creator AS "author" ,
                         ID AS "document" ,
                         dcterms_issued AS "yr" ,
                         rdf_type AS "class"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdf_type IS NOT NULL
           AND dcterms_issued IS NOT NULL
           AND dc_creator IS NOT NULL) BGP1_1
      JOIN
        (SELECT DISTINCT ID AS "class"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdfs_subClassOf IS NOT NULL
           AND rdfs_subClassOf = 'foaf:Document') BGP1_0 ON(BGP1_1.class=BGP1_0.class)
      JOIN
        (SELECT DISTINCT ID AS "author" ,
                         foaf_name AS "name"
         FROM property_table
         WHERE ID IS NOT NULL
           AND foaf_name IS NOT NULL) BGP1_2 ON(BGP1_1.author=BGP1_2.author)) BGP1
   LEFT JOIN
     (SELECT DISTINCT BGP2_1.class2 AS "class2" ,
                      BGP2_1.document2 AS "document2" ,
                      BGP2_1.author2 AS "author2" ,
                      BGP2_1.yr2 AS "yr2"
      FROM
        (SELECT DISTINCT rdf_type AS "class2" ,
                         ID AS "document2" ,
                         dc_creator AS "author2" ,
                         dcterms_issued AS "yr2"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdf_type IS NOT NULL
           AND dcterms_issued IS NOT NULL
           AND dc_creator IS NOT NULL) BGP2_1
      JOIN
        (SELECT DISTINCT ID AS "class2"
         FROM property_table
         WHERE ID IS NOT NULL
           AND rdfs_subClassOf IS NOT NULL
           AND rdfs_subClassOf = 'foaf:Document') BGP2_0 ON(BGP2_1.class2=BGP2_0.class2)) BGP2 ON(((BGP1.author = BGP2
                .author2)
                                                                                                  AND (BGP2.yr2 <
                                                                                                       BGP1.yr))));

CREATE TABLE RESULT AS
  (SELECT DISTINCT a.document AS "document" ,
                   a.yr AS "yr" ,
                   a.name AS "name"
   FROM a
   WHERE a.author2 IS NULL);
```

Listing D.18: SP²Bench Q6 (SQL) - A bug in the Impala query engine was discovered, which lead to wrong results if the last filter operation was included in the first query. Ergo, the query is split into two subqueries.

```
CREATE TABLE a AS
  (SELECT DISTINCT BGP1.member2 AS "member2", OPTIONAL0.member3 AS "member3", OPTIONAL0.member4 AS "member4",
                                BGP1.class AS "class", OPTIONAL0.class4 AS "class4", BGP1.title AS "title",
                                OPTIONAL0.class3 AS "class3", BGP1.doc AS "doc", OPTIONAL0.doc4 AS "doc4",
                                OPTIONAL0.bag3 AS "bag3" ,
                  BGP1.bag2 AS "bag2", BGP1.doc2 AS "doc2", OPTIONAL0.doc3 AS "doc3", OPTIONAL0.bag4 AS "bag4"
   FROM
     (SELECT DISTINCT BGP1_2.member2 AS "member2", BGP1_1.title AS "title", BGP1_2.doc AS "doc" ,
                      BGP1_1.class AS "class", BGP1_3.bag2 AS "bag2", BGP1_3.doc2 AS "doc2"
      FROM
        (SELECT DISTINCT dcterms_references AS "bag2" ,
                         ID AS "doc2"
         FROM property_table
         WHERE ID IS NOT NULL AND dcterms_references IS NOT NULL) BGP1_3
      JOIN
        (SELECT DISTINCT predicate AS "member2" ,
                         OBJECT AS "doc" ,
                                   ID AS "bag2"
         FROM triplestore_parquet
         WHERE ID IS NOT NULL) BGP1_2 ON(BGP1_3.bag2=BGP1_2.bag2)
      JOIN
        (SELECT DISTINCT dc_title AS "title" ,
                         ID AS "doc" ,
                         rdf_type AS "class"
         FROM property_table
         WHERE ID IS NOT NULL AND rdf_type IS NOT NULL AND dc_title IS NOT NULL) BGP1_1 ON(BGP1_2.doc=BGP1_1.doc)
      JOIN
        (SELECT DISTINCT ID AS "class"
         FROM property_table
         WHERE ID IS NOT NULL AND rdfs_subClassOf IS NOT NULL AND rdfs_subClassOf = 'foaf:Document')
         BGP1_0 ON(BGP1_1.class=BGP1_0.class)) BGP1
   LEFT JOIN
     (SELECT DISTINCT BGP3.class4 AS "class4", BGP2.member3 AS "member3", BGP3.member4 AS "member4",
                                 BGP2.class3 AS "class3",BGP2.doc AS "doc", BGP3.doc4 AS "doc4" ,
                      BGP2.bag3 AS "bag3", BGP3.bag4 AS "bag4", BGP2.doc3 AS "doc3"
      FROM
        (SELECT DISTINCT BGP2_2.member3 AS "member3", BGP2_1.class3 AS "class3" ,
                         BGP2_2.doc AS "doc", BGP2_1.bag3 AS "bag3" ,
                         BGP2_1.doc3 AS "doc3"
         FROM
           (SELECT DISTINCT rdf_type AS "class3" ,
                            dcterms_references AS "bag3" ,
                            ID AS "doc3"
            FROM property_table
            WHERE ID IS NOT NULL AND rdf_type IS NOT NULL AND dcterms_references IS NOT NULL) BGP2_1
         JOIN
           (SELECT DISTINCT predicate AS "member3" ,
                            OBJECT AS "doc" ,
                                      ID AS "bag3"
            FROM triplestore_parquet
            WHERE ID IS NOT NULL) BGP2_2 ON(BGP2_1.bag3=BGP2_2.bag3)
         JOIN
           (SELECT DISTINCT ID AS "class3"
            FROM property_table
            WHERE ID IS NOT NULL AND rdfs_subClassOf IS NOT NULL
            AND rdfs_subClassOf = 'foaf:Document') BGP2_0 ON(BGP2_1.class3=BGP2_0.class3)) BGP2
      LEFT JOIN
        (SELECT DISTINCT BGP3_1.class4 AS "class4", BGP3_2.member4 AS "member4" ,
                         BGP3_1.doc4 AS "doc4", BGP3_2.bag4 AS "bag4", BGP3_2.doc3 AS "doc3"
         FROM
           (SELECT DISTINCT predicate AS "member4" ,
                            OBJECT AS "doc3" ,
                                      ID AS "bag4"
            FROM triplestore_parquet
            WHERE ID IS NOT NULL) BGP3_2
         JOIN
           (SELECT DISTINCT rdf_type AS "class4" ,
                            ID AS "doc4" ,
                            dcterms_references AS "bag4"
            FROM property_table
            WHERE ID IS NOT NULL AND rdf_type IS NOT NULL AND dcterms_references IS NOT NULL) BGP3_1 ON(BGP3_2.bag4=
                  BGP3_1.bag4)
         JOIN
           (SELECT DISTINCT ID AS "class4"
            FROM property_table
            WHERE ID IS NOT NULL
              AND rdfs_subClassOf IS NOT NULL
              AND rdfs_subClassOf = 'foaf:Document') BGP3_0 ON(BGP3_1.class4=BGP3_0.class4)) BGP3
         ON(BGP2.doc3=BGP3.doc3)) OPTIONAL0 ON(BGP1.doc=OPTIONAL0.doc AND OPTIONAL0.doc4 IS NULL)) ;

CREATE TABLE RESULT AS
  (SELECT DISTINCT a.title AS "title"
   FROM a
   WHERE a.doc3 IS NULL);
```

Listing D.19: SP²Bench Q7 (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT UNION1.predicate AS "predicate"
   FROM (
         (SELECT DISTINCT BGP1_0.person AS "person" ,
                          BGP1_1.subject AS "subject" ,
                          BGP1_1.predicate AS "predicate" ,
                          NULL AS "object"
          FROM
            (SELECT DISTINCT ID AS "person"
             FROM property_table
             WHERE ID IS NOT NULL
               AND rdf_type IS NOT NULL
               AND rdf_type = 'foaf:Person') BGP1_0
          JOIN
            (SELECT DISTINCT OBJECT AS "person" ,
                                       ID AS "subject" ,
                                       predicate AS "predicate"
             FROM triplestore_parquet
             WHERE ID IS NOT NULL) BGP1_1 ON(BGP1_0.person=BGP1_1.person))
        UNION
          (SELECT DISTINCT BGP2_0.person AS "person" ,
                          NULL AS "subject" ,
                          BGP2_1.predicate AS "predicate" ,
                          BGP2_1.OBJECT AS "object"
          FROM
            (SELECT DISTINCT ID AS "person"
             FROM property_table
             WHERE ID IS NOT NULL
               AND rdf_type IS NOT NULL
               AND rdf_type = 'foaf:Person') BGP2_0
          JOIN
            (SELECT DISTINCT ID AS "person" ,
                            predicate AS "predicate" ,
                            OBJECT AS "object"
             FROM triplestore_parquet
             WHERE ID IS NOT NULL) BGP2_1 ON(BGP2_0.person=BGP2_1.person))) UNION1) ;
```

Listing D.20: SP²Bench Q9 (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1_0.subject AS "subject" ,
                   BGP1_0.predicate AS "predicate"
   FROM
     (SELECT DISTINCT ID AS "subject" ,
                      predicate AS "predicate"
      FROM triplestore_parquet
      WHERE ID IS NOT NULL
        AND OBJECT = '<http://localhost/persons/Paul_Erdoes>') BGP1_0) ;
```

Listing D.21: SP²Bench Q10 (SQL)

```
CREATE TABLE RESULT AS
  (SELECT DISTINCT BGP1_0.ee AS "ee"
   FROM
     (SELECT DISTINCT rdfs_seeAlso AS "ee" ,
                      ID AS "publication"
      FROM property_table
      WHERE ID IS NOT NULL
        AND rdfs_seeAlso IS NOT NULL
      ORDER BY ee LIMIT 10
      OFFSET 50) BGP1_0) ;
```

Listing D.22: SP²Bench Q11 (SQL)

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift