Squerall: Virtual Ontology-Based Access to Heterogeneous and Large Data Sources

Mohamed Nadjib Mami [a,*], Damien Graux [a], Simon Scerri [b], Hajira Jabeen [b] and Sören Auer [c]
[a] *IAIS, Fraunhofer Gesellschaft, Germany*
E-mails: mami@cs.uni-bonn.de, damien.graux@iais.fraunhofer.de
[b] *Bonn University, Germany*
E-mails: scerri@cs.uni-bonn.de, jabeen@cs.uni-bonn.de
[c] *TIB Leibniz Information Center Science and Technology & L3S Research Center, Hannover University, Germany*
E-mail: auer@l3s.de

Abstract. During the last two decades, a huge leap in terms of data formats, data modalities, and storage capabilities has been made. As a consequence, dozens of storage techniques have been studied and developed. Today, it is possible to store cluster-wide data easily while choosing a storage technique that suits our application needs, rather than the opposite. If different data stores are interlinked and queried together, their data can generate valuable knowledge and insights. In this study, we present a unified architecture, which uses Semantic Web standards to query heterogeneous Big Data stored in a Data Lake in a unified manner. In a nutshell, our approach consists of equipping original heterogeneous data with mappings and offering a middleware able to aggregate the intermediate results in a distributed manner. Additionally, we devise an implementation, named Squerall, that uses both Apache Spark and Presto as an underlying query engines. Finally, we conduct experiments to demonstrate the feasibility, efficiency and solubility of Squerall in querying five popular data sources.

Keywords: Distributed Query Processing, Data Lake, OBDA, Heterogeneous Data Access, NoSQL

1. Introduction

For over four decades, relational data management was the dominant paradigm for storing and managing structured data. Use-cases such as storing vast amounts of indexed Web pages and user activities revealed the relational data management's weakness at dynamically scaling the storage and querying to massive amounts of data. This initiated a paradigm shift, calling for a new breed of databases capable of storing terabytes of data without deteriorating querying performance. Google BigTable [1], for example, a high-performing, fault-tolerant and scalable database appearing in 2006, was among those first databases to dis-adhere to the relational model. Since then, a variety of *non-relational* databases has come to existence, e.g., Cassandra, MongoDB, Couchbase, Neo4j, etc.

This development also correlated with the beginning of the new Big Data era of data management. The latter is characterized with three challenges: storing large *volumes* of data, processing fast paced flux of data – *velocity*, and embracing the ever increasing types and structures of data – *variety*. NoSQL databases, collectively with Big Data frameworks, such as Hadoop[2], Spark[3], Flink[4], Kafka[5], etc. efficiently handle storing and processing voluminous and continuously changing data. The support for the third Big Data dimension though, i.e., facilitating the processing of heterogeneous data, remains relatively unexplored.

Semantic Web standards for heterogeneous data integration. For almost two decades, semantic technologies have been developed to facilitate the integration of heterogeneous data coming from multiple sources following the local-as-view paradigm. Local data schemata are *mapped* to global ontology terms, using *mapping languages* that have been standardized

*Corresponding author. E-mail: mami@cs.uni-bonn.de.

for a number of popular data representations, such as relational data, JSON, CSV or XML. Data of *multiple* sources and forms can then be accessed in a *uniform* manner by means of queries using a unique query language: SPARQL, employing terms from the ontology. Such data access, commonly referred to as Ontology-Based Data Access (OBDA) [6], can either be *physical* or *virtual*. In a physical data access, the whole data is exhaustively transformed into RDF [7], based on the mappings. In a *virtual* data access, data remains in its original format and form; it is only after the user issues a query that relevant data is retrieved, by mapping the terms from the query to the schemata of the data.

The pool of heterogeneous data residing in its original format and form is commonly referred to as Data Lake [8]. It can contain databases (e.g. NoSQL stores) or scale-out file/block storage infrastructure (e.g. HDFS distributed file system). Our goal is to facilitate accessing large amounts of original data stored in a Data Lake, i.e., without pre-processing or physical data transformation. A problem here, however, is that data in different representations can not be linked and joined. We, therefore, build a virtual OBDA on top of the Data Lake, and call the resulting concept a SEMANTIC DATA LAKE. Although the term has been used before in [9], we have previously introduced the concept in a scientific publication in [10].

Challenges of OBDA for large-scale data. Implementing an OBDA on top of Big Data poses two major challenges:

- *Query translation.* SPARQL queries must be translated to the query dialect of each of the relevant data sources. Depending on the data type, the generic and dynamic translation between data models can be challenging [11].
- *Federated Query Execution.* In Big Data scenarios, it is common to have non-selective queries, so that joining or possibly union can not be performed on a single node in the cluster, but have to be executed in a distributed manner.

Contributions. In this article, we target the previous challenges and make the following contributions:

- We propose an architecture of an OBDA that lays on top of Big Data sources. Thus, we enable querying large heterogeneous data using a single query language: SPARQL[12].
- We extend the SPARQL syntax to enable declaring *transformations*, using which users can alter join keys and make data *join-able*, when it is not originally so. As we are targeting data that is possibly generated using different applications, data might not be readily join-able. Hence, allowing users to *declaratively* transform their data is of paramount importance.
- We introduce an approach for the distributed query execution, with focus on the multisource join operation.
- We implement an instance of the proposed architecture using state-of-the-art query engines (Apache Spark and Presto), which provides wrappers for several databases with SQL access. We therefore create a tailored SPARQL-to-SQL converter. We call this *Mediated OBDA*, as SQL is used as a mediator middle-ware between the data and the SPARQL interface.
- We build a NoSQL ontology, to classify NoSQL databases and related concepts and use it as part of the data mappings.

The remainder of the article is structured as follows: Section 2 suggests an architecture for the SEMANTIC DATA LAKE. Section 3 introduces our proposal of extending SPARQL syntax to enable data joinability. Section 4 details our approach of querying heterogeneous data with a special focus on the join. Section 5 we describe our implementation of the SEMANTIC DATA LAKE architecture. Section 6 is where we report on our evaluations of the implementation. Section 7 gives an overview of related work. Finally, Section 8 concludes and discusses future directions.

2. A Semantic Data Lake Architecture

2.1. Preliminaries

We define the following terms:

- **Data Attribute:** comprises all concepts used by data sources to characterize a particular stored datum. It can be a table column in a relational database (e.g. Cassandra) or a field in a document database (e.g. MongoDB).
- **Data Entity:** comprises all concepts that are used by different data sources to group similar data together. It can be a table in a relational or a collection in a document database. A data entity has one or multiple data attributes.
- **ParSet:** refers to a data structure that can be partitioned and distributed among cluster nodes.
- **Parallel Operational Zone (POZ):** is the parallel distributed environment where the ParSets live and evolve. In practice, it can be represented by

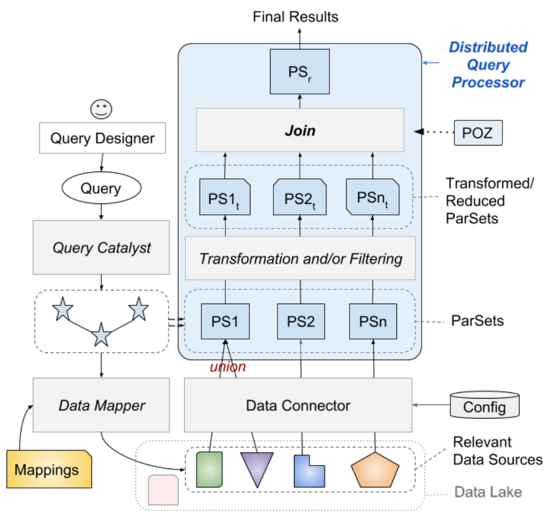


Fig. 1. SEMANTIC DATA LAKE Architecture.

disks or main memory of a physical or a virtual cluster.

- **Mapping:** a relation linking data with its equivalent semantic description. We are only interested in mappings at the schema level, e.g. (*first name*, *foaf:firstName*).
- **Data Source:** any source of data, be it a database e.g. Cassandra or MongoDB, a raw file e.g. CSV, or a structured file format e.g. Parquet. In the present study, we do not consider unstructured and streaming data.

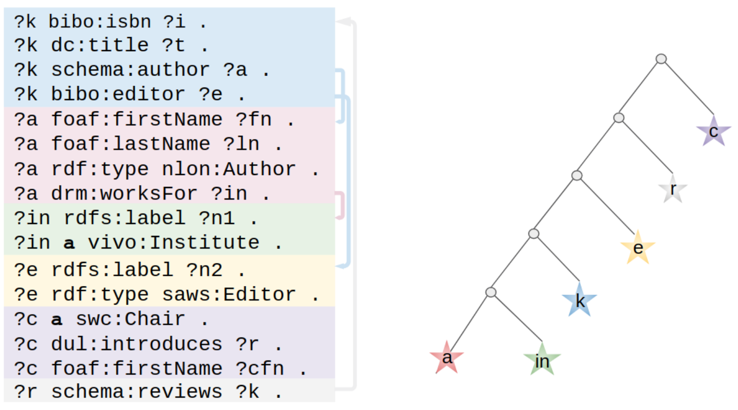
2.2. *Architecture*

A SEMANTIC DATA LAKE adds a semantic layer on top of a Data Lake, by mapping different data representations to the unified RDF data model and suitable vocabulary and ontology terms. Once the original data representations are mapped, SPARQL queries against the mapping ontologies can be translated and executed on the original data. However, different parts of a query can be satisfied by different data sources in the Data Lake, which requires the careful joining/reconciliation of retrieved data. To do so, we envision a SEMANTIC DATA LAKE architecture comprising five components (depicted in Figure 1):

- **Query Catalyst:** As commonly found in many OBDA and query federation systems, this catalyst decomposes the BGP part of a SPARQL query into a set of star-shaped sub-BGPs, or *stars* for short. A star is then a set of triples that share the same subject. Figure 2 (a) shows an example of a BGP of a SPARQL query having six stars, each identified by its subject variable.
- **Data Mapper:** One key concept of OBDA systems is the *mappings*. We use mappings to (1) abstract from the differences found across data schemata, and (2) provide a uniform query interface above heterogeneous data. The Data Mapper takes as input a set of mappings and the list of stars generated by Data Catalyst, and identifies a set of matching data sources. The method will be further explained in section 3.
- **Data Connector:** This component is responsible for connecting data from its storage inside the Data Lake to the POZ. In order for relevant data entities to be loaded as a ParSet inside the POZ (in response to a query), Data Connector requires that each data source be accompanied with metadata containing minimal information needed to access it, e.g., credentials and host of the containing database.
- **Distributed Query Processor:** The intermediate results of (multi-)join or potentially large unselective queries on top of heterogeneous big data sources can overflow the capacity of the single machine. As this is our case, queries have to be executed in a parallel and distributed manner. We explain our approach for a distributed query processing in section 3.
- **Query Designer:** In such a disperse environment with high schema variety and richness, providing an interface for plain-text SPARQL query creation would elevate the barrier of entry to SEMANTIC DATA LAKES. Many companies and institutions[1] have the need for a SEMANTIC DATA LAKES, but neither have the knowledge about SPARQL, nor want to invest in it. Therefore, a guided query designer is a necessity.

We here have presented the general architecture of SEMANTIC DATA LAKE. However, the current study concentrates on the aspect of enabling intra-source distributed query processing with special focus on *joining* those data sources. This involves multiple components of the architecture, mainly the Data Mapper and Distributed Query Processor.

[1]http://newvantage.com/wp-content/uploads/2016/01/Big-Data-Executive-Survey-2016-Findings-FINAL.pdf



```
?k bibo:isbn ?i .
?k dc:title ?t .
?k schema:author ?a .
?k bibo:editor ?e .
?a foaf:firstName ?fn .
?a foaf:lastName ?ln .
?a rdf:type nlon:Author .
?a drm:worksFor ?in .
?in rdfs:label ?n1 .
?in a vivo:Institute .
?e rdfs:label ?n2 .
?e rdf:type saws:Editor .
?c a swc:Chair .
?c dul:introduces ?r .
?c foaf:firstName ?cfn .
?r schema:reviews ?k .
```

Fig. 2. (a) BGP of a query, six stars and four joins (b) Left-deep join plan.

Query	:=	Prefix* `SELECT` Distinguish `WHERE`{ Clauses } Modifiers?
Prefix	:=	`PREFIX` “string:” `IRI`
Distinguish	:=	`DISTINCT`? (“*” | (Var|Aggregate)+)
Aggregate	:=	(AggOpe (Var) `AS` Var)
AggOpe	:=	`SUM`|`MIN`|`MAX`|`AVG`|`COUNT`
Clauses	:=	TP* Filter?
Filter	:=	`FILTER` (Var FiltOpe `Litteral`)
	|	`FILTER regex` (Var , “%string%”)
FiltOpe	:=	= | != | < | <= | > | >=
TP	:=	Var `IRI` Var .
	|	Var `rdf:type` `IRI` .
Var	:=	“?string”
Modifiers	:=	(`LIMIT` k)? (`ORDER BY` (`ASC`|`DESC`)? Var)? (`GROUP BY` Var+)?

Fig. 3. Grammar of the supported SPARQL fragment.

3. Extended SPARQL Syntax to Enable Data Joinability

In this section, we extend the current SPARQL syntax so we enable joining data coming from various data sources. First, in Figure 3, we describe the supported SPARQL fragment of the current study. It mainly consists of the popular conjunctive fragment –the so-called BGP– plus various aggregating operators and solution modifiers.

Transformations. As data can be generated by different applications, which is typical in Data Lakes, the attributes to join on might e.g., have values formatted differently, lying in different value ranges, have invalid values to skip or replace, etc. Hence the join would yield no results, or yields undesirable ones. In order to *fix* the data and bring it to the *joinable* state, we incorporate *transformations* into the SPARQL query; similar in spirit to ETL transformations [13]. We suggest a new clause: `TRANSFORM()`, which is located at the very end of the query, used in the following way:

```
TRANSFORM([leftJoinVar][rightJoinVar]
        .[l|r].[transformation]+))
```

Example:

```
?book  schema:author  ?author .
...
TRANSFORM(?book?author.l.replc("id","1")
                     .toInt.skp(12)
```

This reads as follows. Refer to the needed join by placing its two operands (star IDs) together: `?bk?a`; we call it *join reference*. Next, to instruct that we need to make changes on the variable of the left operand, which is `schema:author`, use the denominator [.l] on the join reference, i.e., `?bk?a.l`. Then, we list the needed transformations dot-separated: `replc("id","1").toInt.skp(12)`. When there is no pattern detected between the join keys of the two tables, or the keys are radically different, like they are in one side numeral auto-increments while in the other random auto-generated textual codes. Consequently, data is declared *unjoinable* and has to be transformed and regenerated by its provider.

4. Distributed Query Processing

Here we describe our approach of enabling and performing distributed query processing in the SEMANTIC DATA LAKE. Three steps are involved:

4.0.1. I. Data Mapping.

In order to enable finding and querying data in the Data Lake, data entities have to be mapped to ontology terms. Three mapping elements need to be provided as input:

- **Predicate mappings:** associates an attribute to an ontology predicate.
- **Class mapping:** associates an entity to an ontology class - optional.
- **Entity ID:** specifies an attribute to be used as an identifier of the entity.

For example, a Cassandra database has an entity (table) `Author` containing the attributes: `first_name`, `last_name` and its primary key `AID`. In order to enable finding this table, the user has to provide the three mapping elements. For example, (1) Predicate mappings: (first_name, foaf:firstName), (last_name, foaf:lastName) (2) Class mapping: (Author, nlon:Author), and (3) Entity ID: AID. `foaf:firstName` and `foaf:lastName` are predicates from the ontology `foaf` and `nlon:Author` is a class from the ontology `nlon`.

4.0.2. II. Relevant Entity Extraction.

Once all SPARQL query stars are extracted (by the Query Catalyst), each star is looked at separately. A star can be typed or untyped. It is typed if it contains a triple with the typing predicate: `rdf:type`. In Figure 2, stars k, a, e and r are untyped, while stars in and c, are typed. The Data Mapper checks the mappings for the existence of entities that have attribute mappings to *each of* the predicates of the star. If a star is untyped, all found entities, regardless of which data source they come from, are regarded as the same. Reciprocally, if the star is typed, the entity space is reduced to only those having the type of the star. For example, suppose a query with two joint stars (`?x _:hasAuthor ?y`) (`?y foaf:firstName ?fn`) (`?y foaf:lastName ?ln`). There exist, in the data, two entities about `authors` and `speakers`, with the two attributes both, mapping to the predicates `foaf:firstName` and `foaf:lastName`; and an entity about `books`. As star y is untyped, and as both `authors` and `speakers` entities have attributes mapping to the predicates of star y, both will be identified as relevant data, and, thus, be joined with star `x` about books. Semantically, this yields wrong results as `speakers` have no relationship with books. Hence the role of the type in a star. Additionally, the availability of type information in the entities can enable incorporating more semantics, like the hierarchies between classes. For example, if all *authors* and *professors* are *researchers*, then querying *researchers* would return results from *authors* and *professors*.

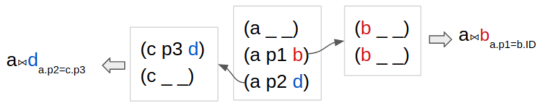


Fig. 4. ParSets join links. Star *a* joins stars *b* and *c*.

4.0.3. III. Distributed Join.

As we are querying multiple data sources, the emphasis in query processing is put on the join operation.

Star-to-Parset. Each star identified by the Query Catalyst will generate one ParSet at the end. The Query Processor takes the entity relevant to each star and loads it into a ParSet. When there are multiple relevant entities for a star, their respective ParSets will be union-ed into one ParSet.

Joining ParSets. The links between the stars identified by Query Catalyst will be translated into joins between the ParSets. Joins can be formulated as follows:

$$On(Join(s_1, s_2), pred) = \left\{ \exists (s_1, s_2) \in \mathcal{S}^2, s_1 \underset{s_1.pred_1 = s_2.pred_2}{\bowtie} s_2 \right\}$$

Where $\mathcal{S}$ is the set of all stars, e.g., in Figure 2 (a), $S = \{k, a, in, e, c, r\}$, $pred1$ is a predicate from $s1$, and $pred2$ can either be the entity ID of $s2$ (right side of Figure 4), or a predicate from $s2$ (left side of Figure 4).

Next, we proceed to join the ParSets, producing at the end the *results ParSet* – see algorithm 1. The join algorithm we currently use is nested-loop join, hence, the most suitable execution order is a deep-left [14]. The input to our algorithm is a hashmap, called *joins map*, containing the ParSet pairs of each identified join, with their join variables: *((parset1,join_variable1) -> (parset2,join_variable2))*. Initially, we start by joining the first pair of the *joins map*, the results of which will constitute the first elements of *results ParSet* (line 3). The two ParSets of the join just computed are added to a list called *joined ParSets* (line 4). Then, we iterate through each join pair from the remaining *joins map* (loop of line 5 starts from 1), and check which ParSet of the two has not yet been added to *joined ParSets* (line 8 to line 15), i.e., joined before with *results ParSet*. If one of the two exists, then join the non-existing ParSet with *results ParSet* using the join variable of the non-existing ParSet and the join variable of the existing one (line 9 and line 12). Then add the two ParSets to *joined ParSets* (line 10 and line 13). If none of the two ParSets has been found in *joined Parsets*, then it is impossible to join, and the pair at hand is added to a queue (line 15). We iterate till the end of the *joins map*. Now, we visit the queue and do exactly as we did with the pairs in the above for-loop (line 8 to line 15), with one addition: after each successful join, de-queue the added pair, till the queue is empty (line 21).

Algorithm 1: Building final results ParSet.

```
Input  : joins_map
Output: results_ps
1  $ps1 \longleftarrow joins\_map[0].key$;
2  $ps2 \longleftarrow joins\_map[0].value$;
3  $results\_ps \longleftarrow ps1.join(ps2).on(ps1.var = ps2.var)$;
4  $joined\_parsets.add(ps1).add(ps2)$;
5  for $i \leftarrow 1$ to $joins\_map.length$ do
6      $ps1 \longleftarrow joins\_map[i].key$;
7      $ps2 \longleftarrow joins\_map[i].value$;
8      if $joined\_parset.contains(ps1)$ and $\neg joined\_parset.contains(ps2)$ then
9          $results\_ps \longleftarrow results\_ps.join(ps2).on(results\_ps.(ps1.var) = ps2.var)$;
10         $joined\_parsets.add(ps1).add(ps2)$;
11     else if $\neg joined\_parset.contains(ps1)$ and $joined\_parset.contains(ps2)$ then
12         $results\_ps \longleftarrow results\_ps.join(ps1).on(results\_ps.(ps2.var) = ps1.var)$;
13         $joined\_parsets.add(ps1).add(ps2)$;
14     else if $\neg joined\_parset.contains(ps1)$ and $\neg joined\_parset.contains(ps2)$ then
15         $pending\_joins.enqueue((ps1, ps2))$;
16 end
17 while $pending\_joins.notEmpty$ do
18     $join\_pair \longleftarrow pending\_join.head$;
19     $ps1 \longleftarrow join\_pair.key$;
20     $ps2 \longleftarrow join\_pair.value$;
       /* Check and join like in lines 8
       to 15                              */
21     $join\_pair \longleftarrow pending\_join.tail$;
22 end
```

This will leave us with one big ParSet joining all the ParSets, as depicted in Figure 2 (b). All the joins are computed in parallel, inside the POZ. Note that in order to solve attribute naming conflicts between stars, like two stars having `foaf:firstName` predicates, we encode the names in the ParSet using the following template: $\{star\_pred\_namespace\}$, e.g., `a_firstName_foaf` for the *author* star and `r_firstName_foaf` for the *reviewer* star.

5. Realizing the Semantic Data Lake

We present in this section an instance of the SEMANTIC DATA LAKE architecture, called Squerall (from semantically query all), built using the following technologies: *RML*[2] to express the mappings, *Apache Spark*[3] and *Presto*[3] to implement the Data Connector and Distributed Data Processor, and *SPARQL* query language used as an interface to the outside. Our implementation is openly available under the terms of *Apache-2.0* from[4].

5.1. Data Mapping

Our mapping language of choice is RML. With minimal settings, RML enables us to annotate entities and attributes following the model explained in section section 3. Although we use the exact terms proposed in RML, our end-goal, at least for this implementation, is different. We do not intend to generate RDF triples, neither physically nor virtually. We rather use them to map entities, and use these mappings to query relevant data given a SPARQL query.

Figure 5 shows how we use RML to map the entity `Author`. (1) `rml:logicalsource` used to specify the entity source and type. (2) `rr:subjectMap` used only to extract the entity ID (in brackets).
(3) `rr:predicateObjectMap`, used as many attributes as the entity has; it maps an attribute using `rml:reference` to an ontology term using `rr:predicate`. Note the presence of the property `nosql:store` from the NoSQL ontology (see next section), it is used to specify the type of the entity, e.g., Parquet, Cassandra, MongoDB, etc.

NoSQL Ontology. The ontology was built to fill a gap we found in RML, that is the need to specify information about NoSQL databases. The ontology namespace

[2]http://rml.io
[3]http://prestodb.io
[4]https://github.com/EIS-Bonn/Squerall

```
<#AuthorMap>
  rml:logicalSource [
    rml:source: "../authors.parquet";
    nosql:store nosql:parquet
  ];
  rr:subjectMap [
    rr:template "http://exam.pl/../{AID}";
    rr:class nlon:Author
  ];
  rr:predicateObjectMap [
    rr:predicate foaf:firstName;
    rr:objectMap [ rml:reference "Fname" ]
  ];
  ...
```

Fig. 5. Mapping an entity using RML.

is http://purl.org/db/nosql# (prefix `nosql`). It contains a hierarchy of NoSQL databases, and some related properties. The hierarchy includes classes for NoSQL databases, KeyValue, Document, Columnar, Graph and Multimodal. Each has several databases in sub-classes, e.g., Redis, MongoDB, Cassandra, Neo4J and ArangoDB, for each class respectively. It also groups the query languages for several NoSQL databases, e.g., CQL, HQL, AQL and Cypher. Miscellaneous categories about how each database calls a specific concept, e.g., indexes, primary keys, viewers; and what is an entity in each database, .e.g., table, collection, data bucket, etc.

5.2. Connecting and Querying Data

The Semantic Data Lake concepts are general and thus not tied to any specific implementation. Any query engine with wrappers allowing to transform data from external data sources to its internal data structure on-the-fly is a valid candidate. For the current study, we implemented and experimented with two popular engines: Apache Spark and Presto. The former is a general-purpose processing engine, while the latter is a distributed SQL query engine for interactive analytic querying, both dealing with heterogeneous large data sources. Both engines base their computations primarily on memory, so they improve upon disk-based processing engines, like *Hadoop*. We chose Spark and Presto because they offer the best compromise between: the number of connectors available and ease of connecting, flexibility and possibility to implement Squerall techniques (efficient intra-source joins and transformations–particularly true for Spark), and performance [15, 16].

```
val dataframe =
    spark.read.format(source_type)
           .options(options).load
// format: data source type, e.g.
    CSV, MongoDB, Cassandra, etc.
```

Listing 1 Spark connection template

I. SQL Query Engine: The internal data format that the selected query engines adopt is tabular and, thus, the adopted query language is SQL. For Spark, prior to querying, data from external data sources has to be first (possibly partially) explicitly loaded, by users, into in-memory tabular structures called *DataFrames*. Presto also loads data into its internal native data structures in memory, however, it does it transparently, so users does not interact with those data structures, but directly issues SQL queries. In either case, we generate SQL queries after analyzing and decomposing the SPARQL query. In SparK, we run portions of SQL queries on DataFrames, then incrementally build a final DataFrame by joining the sub-DataFrames and, if needed, sorting or grouping using Spark methods (transformations). Presto however accepts directly one self-contained SQL query, we construct and run a single query containing all the references to the (relevant) data sources, column projection, selection and grouping.

II. Data Source Wrappers: We leverage Spark's and Presto's concept of Connectors, which are wrappers able to load data from an external source into the internal respective data structures. There are dozens connectors already available for many data sources see, *e.g.*, Spark[5] or Presto[6]. Spark and Presto make interfacing and using connectors very convenient. They only require providing values for a pre-defined list of *options*. Spark requires building DataFrames, however this is simply by passing those options to a connection template (see Listing 1).

6. Evaluation

We evaluate Squerall's effectiveness in fulfilling its purpose: querying Data Lakes using Semantic Web standards.

[5]Spark: https://spark-packages.org
[6]Presto: https://prestodb.io/docs/current/connector

	Product	Offer	Review	Person	Producer
Generated Data (BSBM)	Cassandra	MongoDB	Parquet	CSV	MySQL
# of tuples Scale 0.5M	0.5M	10M	5M	26K	10K
# of tuples Scale 1.5M	1.5M	30M	15M	77K	30K
# of tuples Scale 5M	5M	100M	50M	2.6M	100K

Table 1
Data sources and corresponding number of triples loaded.

6.1. Setup

Datasets: In order to have full control over the scale and nature of data, we opted in this evaluation for synthetic data. Our survey of the literature revealed that there was no suitable benchmark for our exact needs. Namely a benchmark that has (1) a data generator able to produce data under a format that can easily be loaded into the (supported) data sources (e.g., comma-separated values), (2) a set of queries in SPARQL format. For example, [17][18] provide SPARQL queries, but generate data only in RDF; [19][20] generate data in supported format but does not provide SPARQL queries. Therefore, we opt for BSBM benchmark [21], and generate three scales: 500k, 1.5M and 5M (number of products) [7]. In addition to RDF data, BSBM also generates structured data as SQL dumps. We pick five table dumps: Product, Producer, Offer, Review, and Person tables, pre-process them to extract tuples and save them in five different data sources, as shown in table 1.
Queries: Since we only populated a subset of the generated BSBM tables, we had to alter the initial queries as to discard joining with tables we did not consider, e.g., Vendors. Instead, we replace those tables with other populated tables [8]. The queries have various number of joins, from 1 (between two tables) to 4 (between 5 tables). Queries with yet unsupported syntax, e.g. DESCRIBE, CONSTRUCT, are omitted for the time being.
Metrics: We evaluate results accuracy as well as query performance. In the former we store the same data in a relational MySQL database, create equivalent SQL queries, run them and compare the number of results. We used a fully ACID-compliant centralized relational database as a reference for accuracy, because it represents data at its highest level of consistency. In the latter we measure the query execution time. For Squerall, we use Unix `time` function to measure the entire process from the query submission to showing the results, which includes data preparation (for Spark), loading to memory and query execution. In MySQL, we record the time returned when the query is finished. We run each query three times and retain the fastest execution time. The timeout threshold is set to 3600 sec.
Environment: We ran our experiments in a cluster of three machines each having DELL PowerEdge R815, 2x AMD Opteron 6376 (16 cores) CPU, 256GB RAM, and 3TB SATA RAID-5 disk. We used Spark 2.1, Presto 0.204 MongoDB 3.6, Cassandra 3.11 and MySQL 5.7. All queries are run on a cold cache with the default settings of the query engines. Spark 2.1 is used instead of the latest version (currently 2.3), because not all data source connectors support the newer version.

6.2. User Interfaces

In order to facilitate the use of Squerall, we have built three interfaces[9]:

- **Query Designer:** In order to build queries that (potentially) return results, this interface auto-suggests predicates and classes from the underlying mappings. By visualizing the concept of stars, guided by the interface, users can easily and progressively build an image of the data that they want, and build the query accordingly.
- **Data Connector:** It lists Spark and Presto options needed to load and manipulate data using Spark and Presto wrappers (connectors), e.g. credentials, host, ports, etc.
- **Mapping Builder:** It facilitates providing the necessary mapping elements (predicates, class and ID). It extracts attributes from the data sources, by connecting to the data using the options collected by the Data Connector. I provides ontology terms and classes from the Web.

6.3. Results and Discussion

In absence of a contender allowing to query all the five data sources we support, we resort to comparing the performance of the two underlying query engines of our same work: Spark and Presto.

[7]To give a sense of the size of data, the 1,5M scale factor would generate 500M RDF triples, and the 5M 1,75b triples. As we took 5/10 tables, the actual number is smaller. However, the taken tables contain most of the data.

[8]To enable reproducibility, used queries are made available at: https://github.com/EIS-Bonn/Squerall. In addition, we also offer a DockerFile to provide a running implementation of the complete benchmark.

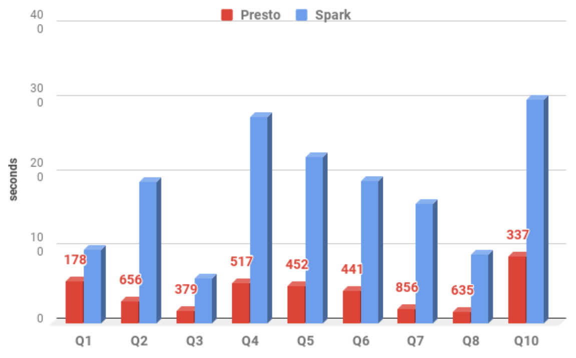
[9]Screencasts available at: https://goo.gl/YFVNup

Accuracy. We run queries against Squerall and MySQL. The number of results returned by Squerall was always identical to MySQL, i.e. 100% accuracy in all cases. MySQL timed out with data of scale factor 1.5M, so then we compared the returned results from the evaluation of both engines, Spark and Presto, and results were identical.

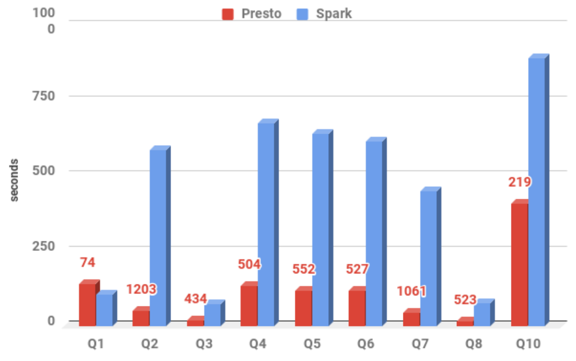
Performance. The results (cf. Figure 6) suggest that Squerall overall exhibits reasonable performance throughout the various queries, i.e., different number of joins, with and without filtering, ordering and grouping. Presto-based Squerall exhibited significantly better performance that Spark-based, up to an order of magnitude. In data scale 0.5M, query performance is superior across all the queries, with an increase of up to 800%. In data scale 1.5M and 5M, Presto-based is superior in all queries besides Q1, with and increase of up to 1300%. This is attributed to the fact that *ad hoc* querying is a core contribution in Presto, while it is a partial contribution in Spark. Presto's data flow and data structure are built, following MPP principles, specifically to accomplish and accelerate analytical interactive querying. Spark, in the other hand, is a general-purpose system, basing its SQL library on its native tabular data structure, called RDDs, which were not originally designed for efficient We note that we have not applied any optimization techniques in both Presto and Spark cases, both engines offer a number of configurations that may improve the query performance. *ad hoc* querying. Increasing data sizes did not deteriorate query performance, which demonstrates Squerall's ability to scale up.

In order to evaluate the effectiveness of the *TRANSFORM* method, we intentionally introduce variations to the data so it becomes unjoinable. In table `Product`, we decrease the column *pr* values by 71, in table `Producer`, we append the string "-A" to all values of column *pr*, and in table *Review* we prefix the values of column *person* with the character "P". We add the necessary transformations to the query, for example `TRANSFORM(?rev?per.l.replc("P",""))` to enable joinability between *Review* and and *Person*. The results show that there is a negligible cost in the majority of the queries in the three scale factors of the data. This is attributed to the fact that both Spark and Presto base computations on main memory. In addition, those transformations to records are executed linearly and locally, without transferring results across the cluster. Only a few queries in scale 5M in Presto-based Squerall exhibited a noticeable but not major cost, like Q1 and Q10. Due to the insignificant differences and also to improve readability, we only add results of the *TRANSFORM* cost in the scale 5M Figure 6c.

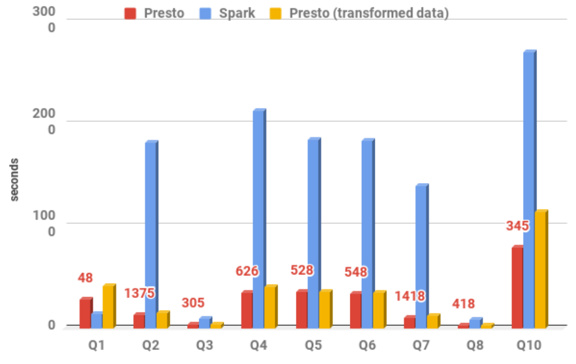
As a side benefit, the results of this experimental evaluation can also be taken as a useful recent benchmarking of the two engines, Spark and Presto, which has little been addressed in the literature [15, 16].



(a) Scale 0.5M.



(b) Scale 1.5M.



(c) Scale 5M.

Fig. 6. Query execution Time (in seconds). The labels on top of Presto's columns show the ratio between Presto's and Spark's execution times, e.g. in (a) the ratio is 178 on Q2, which means that Presto-based Squerall is 175% faster than Spark-based.

7. Related Work

There is a large body of research about mapping relational databases to RDF [22]. Although we share the concepts of ontology-based access and mapping language, our focus goes to the heterogeneous non-relational and distributed and scalable databases. On the non-relational side, there has been a number of works, which we can classify into ontology-based and non-ontology-based.

For non-ontology-based access, [23] defines a mapping language to express access links to data in NoSQL databases. It proposes an intermediate query language to transform SQL to Java methods accessing NoSQL databases. However, query processing is neither elaborated nor evaluated; for example, no mention of how data across databases is joined. [24] suggests that performance of computations can be improved if data is shifted between multiple databases; the suitable database is chosen based on the use-case. Although it demonstrates that the overall performance–including the planning and data movement–is higher when using one database, this is not proven to be true with large volumes of data. In real large-scale settings, data movement and I/O can dominate the query time. [25] allows to run CRUD operations over NoSQL databases. Beyond, the same authors in [26] enable joins as follows. If the join involves entities in the same database, it is performed locally. If not or if the database lacks join capability, data is moved to another capable database. This implies that no intra-source distributed join is possible; and similarly to [24], moving data can become a bottleneck in large scales. [27] proposes a unifying *programming* model to interface with different NoSQL databases. It allows direct access to individual databases using the primitives: get, put and delete. Join between databases is not addressed. Authors in [28] propose a SQL-like language containing invocations to the native query interface of relational and NoSQL databases. The learning curve of this query language is higher than other works suggesting to query solely using plain (or minimally adapted) SQL or JSONPath or SPARQL. Although their architecture is distributed, it is not explicitly stated whether the intra-source join is also distributed. Although interesting, the code-source is unfortunately not available. A number of works, e.g. [29–31], aim at bridging the gap between relational and NoSQL databases, but they demonstrate their approaches with only one database. Given the high semantic and structural heterogeneity found across NoSQL databases, a single database can not be representative of all the family. Among those, [31] adopts JSON as both conceptual and physical data model. This requires that query intermediate results are physically transformed, which has the engine to afford the transformation cost. This limitation is found with few other works as well. Further, the prototype is evaluated with only small data on a single machine. [32] presents an ambitious exhaustive study proposing a general query language, called SQL++. It is based on SQL and JSON and covers a vast portion of the capabilities of query languages found across the various NoSQL databases. However, the focus is all put on the query language, and the prototype is only minimally validated using a single database: MongoDB. [33] considers the case of same data being stored in multiple heterogeneous data sources. It studies the best data source to send a given query to. However, no join is supposed between the sources.

For ontology-based access, the Optique Platform [34] emphasizes the velocity aspect of Big Data by supporting streams in addition to dynamic data. We, on the other hand, emphasis the variety aspect of Big Data by supporting as many data sources as possible. In addition, the source is not publicly available. [35] considers simple query examples, where joins are only minimally addressed. The distributed and parallel implementation of the paper features are left to the future. We note here that although we classify our work as ontology-based, we do not have a separate global schema to maintain, but rather an implicit one encoded in the mappings.

In all the previous work, the support for the variety of data sources is limited or faces bottlenecks. In each observance, few data sources (1-3) are supported, and the wrappers are manually created and/or hard-coded. Squerall, on the other hand, does not reinvent the wheel and makes use of the many wrappers of existing engines. This makes Squerall the work with the widest support of the Big Data Variety dimension in terms of supported data sources. Also in contrast to previous work, from our own experience, Squerall is the most open and easiest to get started with.

8. Conclusion and Future Work

We presented an architecture for a SEMANTIC DATA LAKE, and a realization using two popular engine able to query up to five different data source types (and even more leveraging the engines connectors). Our primary goal was enabling users to accurately query heterogeneous data on-the-fly, i.e. without having to physically transform or move the data, using a single query language. We, thereby, set the foundations for the SEMANTIC DATA LAKE concept, then we plan to expand on a couple of directions. Firstly, design custom similarity joins to enable joinability automatically on query-time, by exploiting both syntactic and semantic similarities. Secondly, we would like to add support for RDF data sources as well streaming data sources. Also, in the current implementation, we adopted a tabular representation during the process of query processing, so only SQL is used given a SPARQL query. However, in the general context of querying heterogeneous data, we are interested in exploring how do the various semantics of each query language understood by the various data sources impose restrictions on the covered SPARQL fragment. Finally, in such a heterogeneous environment, there is a natural need for involving provenance retention both in data and query results levels.

In the long run, we are interested in exploring to what extent can we support the variety dimension of Big Data while still efficiently dealing with volume and velocity at the same time.

References

[1] 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, in: *OSDI*, 2006, pp. 205–218. http://labs.google.com/papers/bigtable-osdi06.pdf.

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

[3] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, Spark: Cluster computing with working sets., *HotCloud* **10**(10–10) (2010), 95.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi and K. Tzoumas, Apache flink: Stream and batch processing in a single engine, *IEEE Computer Society Technical Committee on Data Engineering* **36**(4) (2015).

[5] N. Garg, *Apache Kafka*, Packt Publishing Ltd, 2013.

[6] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini and R. Rosati, Linking data to ontologies, in: *Journal on Data Semantics X*, Springer, 2008, pp. 133–173.

[7] P. Hayes and B. McBride, RDF Semantics, *W3C Rec.* (2004).

[8] D.E. O'Leary, Embedding AI and Crowdsourcing in the Big Data Lake., *IEEE Intelligent Systems* **29**(5) (2014), 70–73. http://dblp.uni-trier.de/db/journals/expert/expert29.html#OLeary14.

[9] B. Reichert, Franz's CEO, Jans Aasman to Present at the Smart Data Conference in San Jose, 2015, [Online; accessed 20-July-2018].

[10] S. Auer, S. Scerri, A. Versteden, E. Pauwels, A. Charalambidis, S. Konstantopoulos, J. Lehmann, H. Jabeen, I. Ermilov, G. Sejdiu, A. Ikonomopoulos, S. Andronopoulos, M. Vlachogiannis, C. Pappas, A. Davettas, I.A. Klampanos, E. Grigoropoulos, V. Karkaletsis, V. de Boer, R. Siebes, M.N. Mami, S. Albani, M. Lazzarini, P. Nunes, E. Angiuli, N. Pittaras, G. Giannakopoulos, G. Argyriou, G. Stamoulis, G. Papadakis, M. Koubarakis, P. Karampiperis, A.-C.N. Ngomo and M.-E. Vidal, The BigDataEurope Platform - Supporting the Variety Dimension of Big Data, in: *17th International Conference on Web Engineering (ICWE2017)*, 2017. http://jens-lehmann.org/files/2017/icwe_bde.pdf.

[11] F. Michel, C. Faron-Zucker and J. Montagnat, A Mapping-Based Method to Query MongoDB Documents with SPARQL., in: *DEXA (2)*, LNCS, Vol. 9828, Springer, 2016, pp. 52–67. ISBN 978-3-319-44405-5. http://dblp.uni-trier.de/db/conf/dexa/dexa2016-2.html#MichelFM16.

[12] S. Harris, A. Seaborne and E. PrudâĂŹhommeaux, SPARQL 1.1 query language, *W3C recommendation* **21**(10) (2013).

[13] S. Chaudhuri and U. Dayal, An Overview of Data Warehousing and OLAP Technology, *ACM Sigmod Record* **26**(1) (1997), 65–74.

[14] H. Garcia-Molina, J. Ullman and J. Widom, *Database Systems – The Complete Book*, Prentice Hall, 2002.

[15] M.S. WiewiÃşrka, D.P. Wysakowicz, M.J. Okoniewski and T. Gambin, Benchmarking distributed data warehouse solutions for storing genomic variant information., *Database* **2017** (2017), 049. http://dblp.uni-trier.de/db/journals/biodb/biodb2017.html#WiewiorkaWOG17.

[16] A. KOLYCHEV and K. ZAYTSEV, RESEARCH OF THE EFFECTIVENESS OF SQL ENGINES WORKING IN HDFS., *Journal of Theoretical & Applied Information Technology* **95**(20) (2017).

[17] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel, SP^{2}Bench: A SPARQL Performance Benchmark, in: *IEEE ICDE*, Shanghai, 2009.

[18] Y. Guo, Z. Pan and J. Heflin, LUBM: A benchmark for OWL knowledge base systems, *Web Semantics* **3**(2) (2005), 158–182.

[19] M. Chevalier, M.E. Malki, A. Kopliku, O. Teste and R. Tournier, Benchmark for OLAP on NoSQL technologies comparing NoSQL multidimensional data warehousing solutions., in: *RCIS*, IEEE, 2015, pp. 480–485. ISBN 978-1-4673-6630-4. http://dblp.uni-trier.de/db/conf/rcis/rcis2015.html#ChevalierMKTT15.

[20] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, Benchmarking cloud serving systems with YCSB., in: *SoCC*, J.M. Hellerstein and et al., eds, ACM, 2010, pp. 143–154. ISBN 978-1-4503-0036-0. http://dblp.uni-trier.de/db/conf/cloud/socc2010.html#CooperSTRS10.

[21] C. Bizer and A. Schultz, The berlin sparql benchmark, 2009.

[22] D.E. Spanos, P. Stavrou and N. Mitrou, Bringing relational databases into the semantic web: A survey, *Semantic Web* (2010), 1–41.

[23] O. Curé, R. Hecht, C. Le Duc and M. Lamolle, Data integration over nosql stores using access path based mappings, in: *International Conference on Database and Expert Systems Applications*, Springer, 2011, pp. 481–495.

[24] V. Gadepally, P. Chen, J. Duggan, A.J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson and M. Stonebraker, The BigDAWG Polystore System and Architecture., *CoRR*

abs/1609.07548 (2016). http://dblp.uni-trier.de/db/journals/corr/corr1609.html#GadepallyCDEHKM16.

[25] R. Sellami, S. Bhiri and B. Defude, Supporting Multi Data Stores Applications in Cloud Environments., *IEEE Trans. Services Computing* **9**(1) (2016), 59–71. http://dblp.uni-trier.de/db/journals/tsc/tsc9.html#SellamiBD16.

[26] R. Sellami and B. Defude, Complex Queries Optimization and Evaluation over Relational and NoSQL Data Stores in Cloud Environments., *IEEE Trans. Big Data* **4**(2) (2018), 217–230. http://dblp.uni-trier.de/db/journals/tbd/tbd4.html#SellamiD18.

[27] P. Atzeni, F. Bugiotti and L. Rossi, Uniform Access to Non-relational Database Systems: The SOS Platform., in: *CAiSE*, J. RalytÃľ, X. Franch, S. Brinkkemper and S. Wrycza, eds, Lecture Notes in Computer Science, Vol. 7328, Springer, 2012, pp. 160–174. ISBN 978-3-642-31094-2. http://dblp.uni-trier.de/db/conf/caise/caise2012.html#AtzeniBR12.

[28] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau and J. Pereira, CloudMdsQL: querying heterogeneous cloud data stores with a common language., *Distributed and Parallel Databases* **34**(4) (2016), 463–503. http://dblp.uni-trier.de/db/journals/dpd/dpd34.html#KolevVBJPP16.

[29] J. Unbehauen and M. Martin, Executing SPARQL queries over Mapped Document Stores with SparqlMap-M, in: *12th International Conference on Semantic Systems Proceedings (SEMANTiCS 2016)*, SEMANTiCS '16, Leipzig, Germany, 2016.

[30] J. Roijackers and G.H.L. Fletcher, On Bridging Relational and Document-Centric Data Stores., in: *BNCOD*, G. Gottlob, G. Grasso, D. Olteanu and C. Schallhart, eds, Lecture Notes in Computer Science, Vol. 7968, Springer, 2013, pp. 135–148. ISBN 978-3-642-39466-9. http://dblp.uni-trier.de/db/conf/bncod/bncod2013.html#RoijackersF13.

[31] A. Vathy-Fogarassy and T. Hugyák, Uniform data access platform for SQL and NoSQL database systems., *Inf. Syst.* **69** (2017), 93–105. http://dblp.uni-trier.de/db/journals/is/is69.html#Vathy-Fogarassy17.

[32] K.W. Ong, Y. Papakonstantinou and R. Vernoux, The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases., *CoRR* **abs/1405.3631** (2014). http://dblp.uni-trier.de/db/journals/corr/corr1405.html#OngPV14.

[33] M. Vogt, A. Stiemer and H. Schuldt, Icarus: Towards a multistore database system., in: *BigData*, J.-Y. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R.A. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang and M. Toyoda, eds, IEEE, 2017, pp. 2490–2499. ISBN 978-1-5386-2715-0. http://dblp.uni-trier.de/db/conf/bigdataconf/bigdataconf2017.html#VogtSS17.

[34] M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jiménez-Ruiz, D. Lanti, M. Rezk, G. Xiao, Ö. Özçep et al., Optique: Zooming in on big data, *Computer* **48**(3) (2015), 60–67.

[35] O. Curé, F. Kerdjoudj, D. Faye, C.L. Duc and M. Lamolle, On The Potential Integration of an Ontology-Based Data Access Approach in NoSQL Stores., *IJDST* **4**(3) (2013), 17–30. http://dblp.uni-trier.de/db/journals/ijdst/ijdst4.html#CureKFDL13.

Appendix A. Benchmark Queries

We list here the 9 SPARQL queries used during our experiments. Please, note that, they are also available from the Github repository of the project and only added in this Appendix for the review process.

Used Prefixes:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/spec/>
PREFIX schema: <http://schema.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX edm: <http://www.europeana.eu/schemas/edm/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX bsbm:
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
```

Query1:

```
SELECT DISTINCT ?label ?value
WHERE {
    ?product rdfs:label ?label .
    ?product bsbm:productPropertyNumeric1 ?value .
    ?product rdf:type bsbm:Product .
    ?product bsbm:producer ?producer .
    ?producer rdf:type bsbm:Producer .
    ?producer foaf:homepage ?hp .
    ?review bsbm:reviewFor ?product .
    ?review rdf:type schema:Review .
    ?review rev:reviewer ?pers .
    ?pers foaf:name ?fn .
    ?pers edm:country ?cn .
    ?offer bsbm:product ?product .
    ?offer rdf:type schema:Offer .
    FILTER (?value > 102)
}
ORDER BY ?label
LIMIT 10
TRANSFORM(?product?producer.r.replc("-A","")
        && ?review?product.r.scl(+71))
```

Query2:

```
SELECT ?label ?comment ?producer ?price
    ?propertyTextual1 ?propertyTextual2 ?propertyTextual3
    ?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4
    ?propertyTextual5 ?propertyNumeric4
WHERE {
    ?p rdfs:label ?label .
    ?p rdfs:comment ?comment .
    ?p bsbm:producer ?producer .
    ?p dc:publisher ?ps .
    ?offer bsbm:product ?p .
    ?offer bsbm:price ?price .
    ?p bsbm:productPropertyTextual1 ?propertyTextual1 .
    ?p bsbm:productPropertyTextual2 ?propertyTextual2 .
    ?p bsbm:productPropertyTextual3 ?propertyTextual3 .
    ?p bsbm:productPropertyNumeric1 ?propertyNumeric1 .
    ?p bsbm:productPropertyNumeric2 ?propertyNumeric2 .
    ?p bsbm:productPropertyTextual4 ?propertyTextual4 .
    ?p bsbm:productPropertyTextual5 ?propertyTextual5 .
    ?p bsbm:productPropertyNumeric4 ?propertyNumeric4 .
}
TRANSFORM(?offer?p.r.scl(+71))
```

Query3:

```
SELECT ?product ?label ?p1 ?p3
WHERE {
    ?product rdfs:label ?label .
    ?product bsbm:producer ?producer .
    ?review bsbm:reviewFor ?product .
    ?review rdf:type schema:Review .
    ?product bsbm:productPropertyNumeric1 ?p1 .
    ?product bsbm:productPropertyNumeric3 ?p3 .
    FILTER (?p1 > 1800)
    FILTER (?p3 < 5 )
}
ORDER BY ?label
LIMIT 10
TRANSFORM(?review?product.r.scl(+71))
```

Query4:

```
SELECT DISTINCT ?label ?c ?propertyTextual ?p1
WHERE {
    ?product rdfs:label ?label .
    ?product bsbm:producer ?pcr .
    ?pcr edm:country ?c .
    ?pcr foaf:homepage ?h .
    ?offer gr:validFrom ?vf .
    ?offer bsbm:product ?product .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?pers .
    ?pers foaf:name ?fn .
    ?pers edm:country ?cn .
    ?product bsbm:productPropertyTextual1 ?propertyTextual
    ?product bsbm:productPropertyNumeric1 ?p1 .
    FILTER (?p1 > 630)
}
ORDER BY ?label
LIMIT 10
TRANSFORM(?product?pcr.r.replc("-A","")
          && ?offer?product.r.scl(+71)
          && ?review?pers.l.replc("P",""))
```

Query5:

```
SELECT DISTINCT ?productLabel ?simProperty1 ?simProperty2
WHERE {
    ?product rdfs:label ?productLabel .
    ?product bsbm:productPropertyNumeric1 ?simProperty1 .
    ?product bsbm:productPropertyNumeric2 ?simProperty2 .
    ?product bsbm:producer ?producer .
    ?review bsbm:reviewFor ?product .
    ?review rdf:type schema:Review .
    ?offer bsbm:product ?product .
    ?offer rdf:type schema:Offer .
    FILTER (?simProperty1 < 120)
    FILTER (?productLabel != "wineskins␣banded␣crc")
    FILTER (?simProperty2 < 170)
}
ORDER BY ?productLabel
LIMIT 5
TRANSFORM(?review?product.r.scl(+71))
```

Query6:

```
SELECT ?label
WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type bsbm:Product .
    ?product bsbm:producer ?producer .
    ?review bsbm:reviewFor ?product .
    ?review rdf:type schema:Review .
    ?offer bsbm:product ?product .
    ?offer rdf:type schema:Offer .
    FILTER regex(?label, "%prelate%")
}
TRANSFORM(?review?product.r.scl(+71))
```

Query7:

```
SELECT ?productLabel ?price ?vendor ?revTitle
?reviewer ?rating1 ?rating2 ?product ?revName
WHERE {
    ?product rdfs:label ?productLabel .
    ?product rdf:type bsbm:Product .
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
    ?offer bsbm:validTo ?date .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?review dc:title ?revTitle .
    ?review bsbm:rating1 ?rating1 .
    ?review bsbm:rating2 ?rating2 .
    ?reviewer foaf:name ?revName .
    ?reviewer a foaf:Person .
    FILTER (?price > 5000)
    FILTER (?product = 9)
}
TRANSFORM(?offer?product.r.scl(+71)
          && ?review?reviewer.l.replc("P",""))
```

Query8:

```
SELECT DISTINCT ?title ?text ?reviewDate ?reviewer
                ?reviewerName ?rating1 ?rating2
                ?rating3 ?rating4 ?product
WHERE {
    ?product rdfs:label  ?label .
    ?product bsbm:productPropertyTextual1 ?pt .
    ?product bsbm:producer ?producer .
    ?producer edm:country ?c .
    ?producer foaf:homepage ?h .
    ?review bsbm:reviewFor ?product .
    ?review dc:title ?title .
    ?review rev:text ?text .
    ?review bsbm:reviewDate ?reviewDate .
    ?review rev:reviewer ?reviewer .
    ?review bsbm:rating1 ?rating1 .
    ?review bsbm:rating2 ?rating2 .
    ?review bsbm:rating3 ?rating3 .
    ?review bsbm:rating4 ?rating4 .
    ?reviewer foaf:name ?reviewerName .
    ?reviewer a foaf:Person .
    FILTER (?product = 9)
}
ORDER BY DESC(?reviewDate)
LIMIT 9
TRANSFORM(?product?producer.r.replc("-A","")
          && ?review?reviewer.l.replc("P",""))
```

Query10:

```
SELECT DISTINCT ?price ?deliveryDays ?date ?c
WHERE {
    ?offer bsbm:product ?product .
    ?offer bsbm:vendor ?vendor .
    ?offer dc:publisher ?publisher .
    ?offer bsbm:deliveryDays ?deliveryDays .
    ?offer bsbm:price ?price .
    ?offer bsbm:validTo ?date .
    ?offer bsbm:producer ?producer .
    ?product rdfs:label  ?label .
    ?product bsbm:productPropertyTextual5 ?pt .
    ?producer edm:country ?c .
    ?producer foaf:homepage ?h .
    FILTER (?product > 9)
    FILTER (?deliveryDays <= 7)
    FILTER (?c = "DE" )
    #FILTER (?date > "2008-07-02 00:00:00" )
}
ORDER BY ?price
LIMIT 10
TRANSFORM(?offer?product.r.scl(+71)
          && ?offer?producer.r.replc("-A",""))
```