'Teach me to fish' Querying Semantic Data Lakes

Mohamed Nadjib Mami
Fraunhofer Institute for Intelligent Analysis and Information Systems IAIS
19 PUBLICATIONS **135** CITATIONS

Hajira Jabeen
University of Bonn
76 PUBLICATIONS **484** CITATIONS

Sören Auer
Leibniz Universität Hannover
549 PUBLICATIONS **17,576** CITATIONS

‘Teach me to fish’ Querying Semantic Data Lakes

Mohamed Nadjib Mami[1], Hajira Jabeen[2], and Sören Auer[3]

[1] Fraunhofer IAIS, Germany - mami@cs.uni-bonn.de
[2] Bonn University, Germany - jabeen@cs.uni-bonn.de
[3] TIB Leibniz Information Center Science and Technology & L3S Research Center, University of Hannover, Germany - auer@l3s.de

Abstract. We have recently made a huge leap in terms of data formats, data modalities, and storage capabilities. Dozens of data storage techniques have been created as a result. Today, we are able to store cluster-wide data, and to choose a storage technique that suits our application needs, rather than the opposite. If different data stores are interlinked and integrated, this data can generate valuable knowledge and insights. In this article, we present an approach that uses semantic technologies to query heterogeneous Big Data stored in a Data Lake in a unified manner. Our approach is based on equipping original data stored in the Data Lake with mappings and adding transformations to the SPARQL query syntax to make heterogeneous data joinable across the Data Lake. We devise an implementation, named Sparkall, that uses Apache Spark as the underlying query engine. Our evaluation demonstrates the feasibility and efficiency of Sparkall in querying five popular data sources.

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 of 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 [2], 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 came to existence.

This development also correlated with the beginning of the new Big Data era of data management. The three challenges[4] are: 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

[4] There are other dimensions, e.g., Veracity, and Value, but these three mentioned are the original and most common ones.

Data frameworks, such as Hadoop, Spark, Flink, Kafka, etc. efficiently handle storing and processing of voluminous and continuously changing data. The support for the third Big Data dimension though, i.e., facilitating the processing heterogeneous data, remains still less explored.

Semantic Web standards for data integration. For almost two decades, semantic technologies are 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 for a number of popular data representation techniques, such as relational data, JSON, CSV or XML. Data can then be queried using the SPARQL query language employing terms from the ontology. Such data access can either be *physical* by exhaustively transforming the whole input data into RDF, based on the mappings. In a *virtual* data access method, called Ontology-Based Data Access (OBDA) [8], the data remains in its original format and form. It is only after the user issues a query that relevant data is pulled, by mapping the terms from the query to the schemata of the data.

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. For example, translating queries to the document query syntax in MongoDB is very complex as reported by [7].
– *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 and NoSQL databases. Thus, we enable querying heterogeneous data using a single query language: SPARQL.
– We extend the SPARQL syntax to enable declaring *transformations*, using which users can alter join values and make data *join-able*, when it is not in its original representation. 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 implement an instance of the proposed architecture using Apache Spark, 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 details the implementation of an instance of the architecture, named Sparkall. Section 4 is where we report on our evaluations of the implementation. Section 5 gives an overview of related work. Finally, Section 6 concludes and discusses future directions of the work.

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 store 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 tabular data structure that can be operated on in parallel, since it is 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 disks or main memories of a physical or virtual cluster.
– *Semantic Data Lake* is the heterogeneous pool of data consisting of data lying in its original raw format, without a binding high-level schema, is often referred to as a *Data Lake*. As such, adding an OBDA-like access lifts the lake into a *Semantic* Data Lake.

2.2 Architecture

Data warehouses follow the Extract-Transform-Load (ETL) paradigm, a process whereby data is first extracted, then undergoes a series of transformations that brings it to a desired form and structure, then loads this in the warehouse. A Data Lake, on the other hand, stores data in its original format on a scalable file or block storage infrastructure (e.g. using the HDFS cluster file system) often using commodity hardware instead of expensive clusters. A problem here, however, is that data in different representations can not be jointly queried. A semantic layer on on top of a Data Lake can help to map different data representations to the unified RDF data model and suitable vocabularies and ontologies. Once 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 the requires intelligently joining data. To do so, we envision a Semantic Data Lake architecture comprising five components: (1) Query Catalyst (2) Data Mapper, (3) Data Connector, (4) Distributed Query Processor, and (5) Query Designer, all depicted in Figure 1.

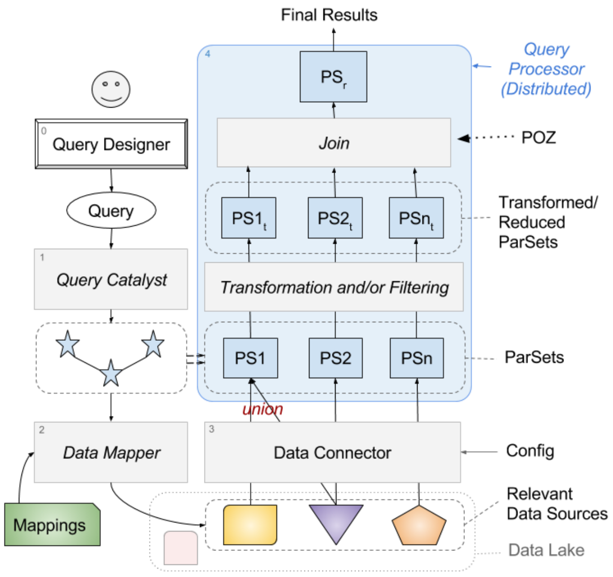


Fig. 1. Semantic Data Lake Architecture.

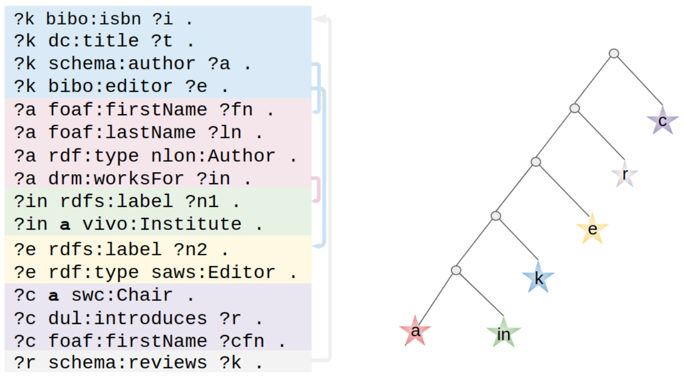
Query Catalyst. Once the user issues a SPARQL query, the BGP part of the query is decomposed into a set of star-shaped sub-BGPs, or *stars* for short. A star is then a set of triples that share the same subject. Figure 2 (a) shows an example of a BGP of a SPARQL query having six stars, each identified by its subject variable. A star can be typed or untyped. It is typed if there is a triple with the typing predicate: `rdf:type` (or a). In figure 2, stars k, a, e and r are untyped, while stars in and c, are typed.

Next, the stars will be linked together in the following way. Whenever a variable is detected in the object position of any of the triples of the star, it is checked if (1) it represents another star in the query, or (2) it is present in the object of another star. Note the arrows on the right of the colored boxes on figure 2 (a).

Data Mapper. One key concept of OBDA systems is the *mappings*, which are means of associating raw data with its equivalent semantic description. We use mappings to (1) abstract from the differences found across data schemata, and (2) provide a uniform query interface above heterogeneous data.

Entity Mapping. We use the mapping language to map entities contained in the Data Lake, or at least the ones that we want to be queried and found. The mappings are provided by the user as an input[5]. Three types of mappings are to

[5] Whether they are created manually, semi-automatically or automatically is out of the scope of this work.



```
?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.

be provided by the user: (1) attributes mapping, (2) entity mapping – optional, and (3) entity ID (primary key). 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:

- Attribute mappings: (first_name, foaf:firstName), (last_name, foaf:lastName)
- Entity mapping: (Author, nlon:Author)
- Entity ID: AID.

Where `foaf:firstName` and `foaf:lastName` are predicates from the ontology `foaf` and `nlon:Author` is a class from the ontology `nlon`.

Mappings usage. Once all query stars are extracted, each star is looked at separately. The Data Mapper checks the mappings for the existence of entities that have attribute mappings to *each of* the predicates of the star. The type of star (`rdf:type X`), if present, reduces the entities space to only those having the type of the star. Reciprocally, if a star is untyped, all found entities, regardless of which data source they come from, are regarded as the same. This can have a negative effect when the star is to be joined with other stars. For example, suppose a query with two joint stars. A star (supposedly) about `authors`, with the two predicates `foaf:firstName` and `foaf:lastName`; and a star about `books`. As both `authors` and `speakers` entities have those two attributes, they will be identified as relevant data, and, thus, be joined with `books`. This clearly yields wrong results as `speakers` have no relationship with books. Moreover, the availability of type information in the entities can enable advanced use-cases of Semantic Web, like creating hierarchies between classes. For example, if all *authors* and *professors* are *researchers*, then querying *researchers* would return results from *authors* and *professors*. This is left for the future, though.

Data Connector. The Data Connector is the component responsible for connecting data from its storage inside the Data Lake to the POZ. In order for

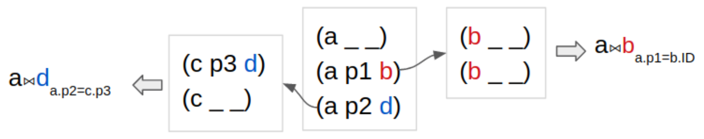


Fig. 3. ParSets join links.

relevant data entities to be loaded as a ParSet inside the POZ (in response to a query), Data Connector requires that each data, or at least needed ones, to be accompanied with metadata containing minimum information needed to access it, e.g., credentials and host of the containing database. The Data Connector serves as a middle-ware between the Data Lake and the Query Processor explained next.

Distributed Query Processor

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

$$\boxed{On(Join(s1, s2), pred) = \{\exists s1, s2 \in \mathcal{S} \wedge s1 \bowtie_{s1.pred1=s2.pred2} s2\}}$$

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 $s2$ (case 1 from 2.2), or a predicate from $s2$ (case 2 from 2.2). See figure 3.

Once all joins are identified, we proceed to compute the actual joins between all the ParSets, producing at the end the *results ParSet*. The joins we compute are nested-loop joins, hence, the most suitable execution order is a deep-left [4], see algorithm 1. Initially, we keep all the identified joins in a hash map, containing the two ParSets of each join along with their join variables, ((parset1,join_variable1) -> (parset2,join_variable2)), called *joins map*. This will act as the input to the algorithm. We start by joining the first pair of the *joins map*, results of which will be the first elements of *results ParSet* (line 3). This latter will be the base relation for all the remaining joins. The two ParSets of the join just computed are added to a list called *joined ParSets* (lines 4). Then, we iterate through each join pair of the remaining *joins map*, and check which ParSet of the two has not been added to *joined ParSets* (lines 8 to 15), i.e., joined before with *results ParSet*. If only one of the two exists, then join the non-existing ParSet with *results ParSet* using the join variable of the existing ParSet and the join variable of the non-existing one (line 9 and 12). Then add the two ParSets to *joined ParSets* (lines 10 and 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 (lines 8 to 15), with one addition: after each successful join, de-queue the added pair (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 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` (author star) and `r_firstName_foaf` (reviewer star).

Transformations. As data can be generated by different applications, the attributes to join on might e.g., have values formatted differently, lying in different value ranges, or have invalid values to skip or replace. Hence the join would yield no results, or yields undesirable ones. Transformations are then means to fix the data and bring it to the *joinable* state. Query transformations are executed on the ParSets, i.e., in parallel and distributed. If the storage support of the POZ is main memory, these transformations can be applied very efficiently.

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. Also, when we mention data silos, we talk about companies[6] and institutions that might have the need for a Semantic Data Lake, but neither do they have the knowledge about SPARQL, nor they want to invest in it. Therefore, a query designer is a necessity. We identified the following requirements:

– R1. The query should be built using the same principle as the Query Catalyst, i.e., the concept of stars, and links between stars.
– R2. Stars should contain only predicates that exist in the mappings, predicates that exist together in an entity.
– R3. Users incorporate transformations in a declarative way, i.e., they specify which transformations they want, not how they implement them.

R2 is suggestive only, users can build stars otherwise, but they will obtain no results. Therefore, Query Designer not only supports building a query without knowledge of SPARQL, but also allows to build queries that potentially return results. In order to meet R3, we extend the SPARQL syntax to allow declaring transformations applied to join variables. We suggest a new clause: `TRANSFORM()`, which is used inside the scope of graph pattern in the following way:

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

For example:

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

This reads as follows. Refer to the needed join by placing its two operands (stars) 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 separated with dots: `replc("id","1").toInt.skp(12)`. As the role of transformations is to make the two stars joinable, these transformations have to be executed on the ParSet of the concerned star *before* the join. When there is no pattern detected between the join values of the two tables to join, or the values are radically different, like values are in one side numeral auto-increments while in the other are random auto-generated textual codes. In that case, data is declared *unjoinable* and has to be transformed and regenerated by its provider.

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

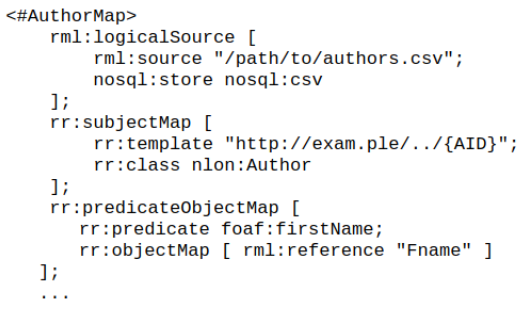
3 Sparkall: Semantic Data Lake in action

We present a realization of the Semantic Data Lake architecture, that we name Sparkall. It is one instance of the architecture, where components are implemented using a selection of technologies. Sparkall is based on *Apache Spark*[7], a general-purpose processing engine for Big Data, and the SPARQL query language used as an interface to the outside.

3.1 Data Mapper: RML

Our mapping language of choice is RML[8]. With minimal settings, RML enables us to annotate entities and attributes, exactly the way we need it. 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 (as explained in 2.2), and use these mappings to query relevant data given a SPARQL query.

Figure 4 shows an example of mapping the entity `Author` using RML. An entity mapping needs three building blocks: (1) `rml:logicalsource` used to specify the entity source and type. (2) `rr:subjectMap`, used only to extract the entity ID[9]. (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 type of the entity, e.g., Parquet, Cassandra, MongoDB, etc.



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

Fig. 4. Example of RML mappings.

We provide a user interface that guides and helps users in adding suitable mappings to their data. The interface has a placeholder for each of the mapping elements described in 2.2: attributes, entity and ID. Attributes and entity placeholders are pre-filled with terms from the LOV catalog[10]. User can validate, choose from LOV or create new terms. Among the attributes, they choose which will be the ID/primary key of the entity.

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 is `http://purl.org/db/nosql#` (prefix `nosql`). It contains a hierarchy of

[7] `https://spark.apache.org`
[8] `http://rml.io`
[9] The variable part, which is in brackets, is used as the primary key of the entity.
[10] `http://lov.okfn.org/dataset/lov/`

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 databses, e.g., CQL, HQL, AQL and Cypher.

3.2 Data Connector

We leverage here Spark's concept of Connector, which is a wrapper to load data from an external source into Spark data structures. The advantage is that, as Spark has grown in popularity, dozens of data sources developed their own connectors, offered either by the data source provider itself or by the open-source community. Spark Packages[11], the portal where Connectors are published, counts around 50 data source connectors.

Spark makes interfacing with a connector and building parallel data structures as a result very convenient. In most cases, it only requires giving values to a pre-defined list of *options*, passed to a connection template (see listing 1.1). We take advantage of this and create a graphical interface where users obtain effortlessly the list of available options, they only provide values for keys.

Listing 1.1. Spark connection template

```
val dataframe = spark.read.format(format).options(options).load
// format is to specify the data source type
```

On the downside, loading data into parallel data structures in memory is an extra step that adds up to the overall query response time. However, Spark offers a number of ways to alleviate this, e.g., by enabling pushing-down operations to the data source whenever possible, and by caching data so it is loaded only once at the first time and reused in subsequent queries. We deem this an unavoidable trade-off between performance and support for variety in Big Data.

3.3 Query Processor: Apache Spark

Spark bases its computation primarily on memory, so it improves upon disk-based processing engines, like *Hadoop*. We use, in particular, Spark's API for querying structured data: Spark SQL. The parallel data structures in Spark SQL are called DataFrames and can be queried using SQL, similarly to tables in databases. We use Spark for implementing the Query Processor. Thus, the POZ is where Spark stores its data during the computation, i.e., mainly memory, then disk. DataFrames are the implementation of our ParSets.

Figure 5 shows how stars are translated into SQL queries posed against DataFrames. Sparkall iterates through the predicates of each star. It visits the mappings to look for entities having attributes mapping to each of the predicates. In the figure, the top box (star + mappings) finds one entity about `authors`.

[11] `https://spark-packages.org`

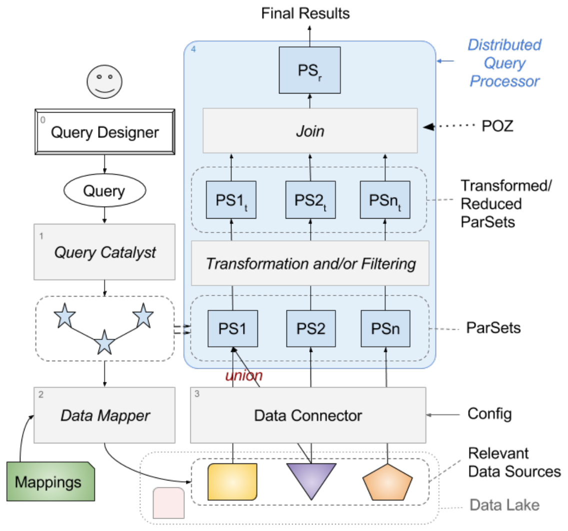
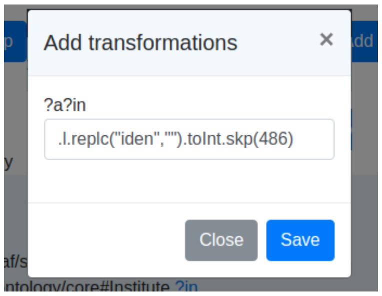


Fig. 5. Generating SQL queries on DataFrames starting from star joins.

The entity is loaded into a DataFrame called DF_A, and an SQL query is generated out-projecting the attributes found via the mappings. This reduces the DataFrame and generates a new one called A (we use the syntax of SQL views for brevity). Similarly with the star in the middle box, it detects the entity `Institute` that is loaded into a DataFrame DF_{IN} and queried generating a new DataFrame IN. This star is typed (`?in a vivo:Institute`), hence only entities typed with this class are taken (`?sm rr:class vivo:Institute`). If SPARQL query contains a filter on variables of the star, it is translated into SQL filters on the DataFrames; this generally is straightforward. The last box is part of the top one (we separated it for clarity), it shows the join point between the two stars. It triggers a join between the two DataFrames A and IN, following the scheme explained in figure 3. The left join variable is the attribute mapping to the predicate `drm:worksFor`, and the right join variable is the primary key of the entity/DataFrame (from `rr:template`).

3.4 Query Designer

We offer users a graphical interface which allows to create queries using *query blocks*, e.g., a block to create stars (requirement R1 in Query Designer architecture), `WHERE` clause, `TRANSFORM` clause (R3), etc. We provide for each query block a *widget* where users are guided to only fill values, not construct query blocks (see figure 6 for transformation widget). In order to build queries that (potentially) return results, we auto-suggest predicates and classes from the underlying metadata, predicates that appear together in entities (RML mappings) (R2). Users progressively add stars to the query (R1), by specifying their de-



Add transformations

?a?in

.l.replc("iden","").toInt.skp(486)

Fig. 6. Widget of transformation creation.

```
SELECT ?l ?c ?vf ?lg ?c1
WHERE {
  ?p   rdfs:label          ?l .
  ?p   xmpl:propertyText1 ?pt .
  ?p   mo:producer         ?pcr .
  ?pcr edm:country         ?c .
  ?pcr foaf:homepage       ?h .
  ?o   gr:validFrom        ?vf .
  ?o   bbc:product         ?p .
  ?r   bbc:product         ?p .
  ?r   dbpedia-owl:person ?pr .
  ?r   dcterms:language    ?lg .
  ?pr  edm:country         ?c1 .
  ?pr  a                foaf:Person .
  FILTER (?c = "DE") .
  TRANSFORM (?r?pr.r.replc("ID","").
    toInt && ?p?pcr.r.scl(-109))
}
```

Fig. 7. Q4' with transformations.

nominators (subject), a list of attributes and their values (predicate, object), and optionally a type (class). By visualizing the concept of stars guided by the interface, users can easily and progressively build an image of the data they want, and build the query accordingly, e.g., a query with a star of `authors`, then one of `books`, then `publishers`, etc.

Note that the usage of SPARQL in our approach is not an aim in itself, but a means to solving the heterogeneity problem. Therefore, we do not seek to provide a full-fledged SPARQL interface, but rather a subset of its functions that allow us to join and filter data coming from different sources. For example, we are not concerned with RDF-specific functions, like `isLiteral` or `isBlank`, etc.

4 Evaluation

We evaluate Sparkall[12] effectiveness in fulfilling its purpose: querying Data Lakes using Semantic Web standards.

Datasets: In order to have full control of the scale and nature of data, we opted in this first evaluation series for synthetic data. We use the BSBM benchmark[13] with e-commerce data in three scales: 500k, 1.5m and 5m (number of products). 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: Cassandra, MySQL, MongoDB, Parquet, and CSV, respectively. In order to evaluate the effectiveness of the transformations, we intentionally introduce variations to the data so it becomes unjoinable. In the table `Person`, we converted the column *pr* to text and

[12] code: `https://github.com/mnmami/sparkall` / screencast: `https://goo.gl/YFVNup`
[13] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`

Query	500k			1.5m			5m
	Sparkall	**MySQL**	**Accuracy**	**Sparkall**	**MySQL**	**Accuracy**	**Sparkall**
Q1	175.33	1	100%	130.67	1	100%	57.33
Q2	270.33	540	100%	152.67	4200	100%	85.67
Q3	153	1440	100%	207	-	-	83.67
Q4	303.33	1380	100%	608	-	-	2100
Q4'	288.67	-	100%	523	-	-	1839.33

Table 1. Query performance results (in seconds).

prefixed all its vales with "ID". In table `Producer`, we incremented all the values of the column *pr* by 109.

Queries: We created four SPARQL queries: Q1 has one join between two data sources, Q2 has two joins between three data sources, Q4 queries all the five data sources. All queries have one filter condition. Q4 comes in two variations: Q4 is used to query the data as it was generated, and Q4' is used to query the data after it has been altered to break the joinability. Figure 7 show Q4', it contains transformations that omit the extra prefix "ID" from the values of the join variable of ?pr, then convert them to integer type. Then values of the join variable of `?pcr` are subtracted by 109 to reach the values range of the other join table. The threshold is set to 1800s (30min).

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 Sparkall, we use Unix `time` function on `./spark-submit` script execution, so it includes all data preparation, loading to memory and execution, by Spark. In MySQL, we measured the time when the query is finished. We run each query three times and compute the average.

Environment setup: We ran our experiments in a small-sized 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, MongoDB 3.6, Cassandra 3.11 and MySQL 5.7. All queries are run on a cold cache with default settings, no performance tuning has been applied.

Results and discussion: Table 1 summarizes the results.

Accuracy. We run queries against Sparkall and MySQL. The number of results returned by Sparkall was always identical to MySQL, hence the 100% accuracy in all cases. For Q3-Q4 on the 1.5m scale, we were not able to complete queries against MySQL.

Performance. The results suggest that in medium size data, increasing the number of data sources and joins does not significantly affect the performance. Note also that, comparing Q4 and Q4', the transformations, i.e., transforming the data to enable the join, did not have an effect on query performance. That confirms our assumption that, when using main memory as storage during computation, the transformations are executed very efficiently. The numbers indicate that the query time of Q4' is even shorter. Throughout our experiments with Spark and other distributed systems, we noticed that the performance of identical or similar computations might differ significantly. This is probably due to the fact that heavy workloads require shuffling data, which involves the bandwidth factor, in addition to other factors like the garbage collector performance. The same is noticed between Q3 and Q2: even though Q3 involves one more join with one more data source, it was faster than Q2 on average. In the largest scale 5m, Q4 was around the threshold. This is because the query is unselective, and it joins the results of already unselective query, of 3-joins chain, Q3, with 100m records of the `Offers` entity.

5 Related Work

Several approaches for mapping relational databases to RDF exist, of which R2RML became a W3C standard. There exists a compact Sparqlification Mapping Language (SML) mapping language [9] with equal expressiveness to R2RML. MASTRO[1] provides a proprietary API for ontology-based data access (OBDA), where the specified ontology is connected to external JDBC enabled data management systems through semantic mappings. A similar combined approach [6] incorporates the information given by the ontology into the data and employs query rewriting to eliminate spurious answers. However, most of these techniques are tailored towards SPARQL access for a singe RDB. The Optique Platform [5] represents an example of OBDA adapted for Big Data. The Optique Platform starts from a conventional OBDA, and hence the general architecture is similar to Sparkall. However, we both adapt it for different end-goals. Sparkall emphasizes the variety aspect of Big Data by supporting as many data sources as possible, while Optique focuses more on the velocity aspect and supports ontology discovery. We, on the other hand, currently have no explicit ontology, but an implicit one encoded in the mappings, and reserve ontology discovery to future work. Authors in [3] deal with data heterogeneity. They suggest that performance of computations can be improved if we switch data, during the computation, between multiple databases, each used for what it is best for. They show that the overall performance, including the planning and data movement, is better than using only one database. However, there is no mention of the scale of the data, data movement and I/O might dominate the execution time if the data is very large.

6 Conclusion and Future Work

We presented an architecture for a Semantic Data Lake, and a realization using Apache Spark able to query up to five different data source types (and even more leveraging Spark connectors). Our primary goal was enabling users to accurately query heterogeneous data with only two steps: mapping the data, and querying the data using a single query language; both further supported with graphical interfaces. Currently, large datasets and unselective queries result in performance decrease. In the current version of Sparkall, we did not yet perform extensive performance tuning and plan this for the next iteration. Further experiments will shed more light on to what extend can we support the variety dimension of Big Data while dealing with volume and velocity at the same time. We used Spark to implement the Query Processor because of its significant number of ready-to-use connectors but the overall approach is not tied to Spark. In a next version, for example, we aim to extend support for Flink. The current work can be seen as foundation for realizing the Semantic Data Lake concept. We plan to expand on a couple of directions, mainly: design other types of joins between the ParSets and choose the most suitable one based on a cost model, support more complex queries with possibly multi-typed stars, explore approaches for learning and suggesting more suitable mappings, and enable provenance for both data and query results.

References

1. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.
2. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, pages 205–218, 2006.
3. Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron J. Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The bigdawg polystore system and architecture. *CoRR*, abs/1609.07548, 2016.
4. Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems – The Complete Book*. Prentice Hall, 2002.
5. Martin Giese, Ahmet Soylu, Guillermo Vega-Gorgojo, Arild Waaler, Peter Haase, Ernesto Jiménez-Ruiz, Davide Lanti, Martin Rezk, Guohui Xiao, Özgür Özçep, et al. Optique: Zooming in on big data. *Computer*, 48(3):60–67, 2015.
6. Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to ontology-based data access. In *IJCAI*, volume 11, pages 2656–2661, 2011.
7. Franck Michel, Catherine Faron-Zucker, and Johan Montagnat. A mapping-based method to query mongodb documents with sparql. In *DEXA (2)*, volume 9828 of *LNCS*, pages 52–67. Springer, 2016.
8. Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *Journal on Data Semantics*, 10:133–173, 2008.
9. Claus Stadler, Jörg Unbehauen, Patrick Westphal, Mohamed Ahmed Sherif, and Jens Lehmann. Simplified rdb2rdf mapping. In *LDOW@ WWW*, 2015.