Uniform Access to Multiform Data Lakes using Semantic Technologies

Mohamed Nadjib Mami
Fraunhofer IAIS & SDA Group, University of Bonn, Germany
mami@cs.uni-bonn.de

Damien Graux
Fraunhofer IAIS, Germany & ADAPT Centre, Trinity College of Dublin
damien.graux@adaptcentre.ie

Simon Scerri
Fraunhofer IAIS, Germany
simon.scerri@iais.fraunhofer.de

Hajira Jabeen
SDA Group, University of Bonn, Germany
jabeen@cs.uni-bonn.de

Sören Auer
TIB Leibniz Information Centre & University of Hannover, Germany
auer@l3s.de

Jens Lehmann
Fraunhofer IAIS & SDA Group, University of Bonn, Germany
jens.lehmann@cs.uni-bonn.de

ABSTRACT

Increasing data volumes have extensively increased application possibilities. However, accessing this data in an *ad hoc* manner remains an unsolved problem due to the diversity of data management approaches, formats and storage frameworks, resulting in the need to effectively access and process distributed heterogeneous data at scale. For years, Semantic Web techniques have addressed data integration challenges with practical knowledge representation models and ontology-based mappings. Leveraging these techniques, we provide a solution enabling uniform access to large, heterogeneous data sources, without enforcing centralisation; thus realizing the vision of a *Semantic Data Lake*. In this paper, we define the core concepts underlying this vision and the architectural requirements that systems implementing it need to fulfill. Squerall, an example of such a system, is an extensible framework built on top of state-of-the-art Big Data technologies. We focus on Squerall's distributed query execution techniques and strategies, empirically evaluating its performance throughout its various sub-phases.

CCS CONCEPTS

• **Information systems** → **Database query processing**; **Parallel and distributed DBMSs**; **Mediators and data integration**; • **Applied computing** → **Information integration and interoperability**; • **Computing methodologies** → *Knowledge representation and reasoning*.

KEYWORDS

Semantic Data Lake, Data Variety, Big Data, SPARQL, NoSQL

ACM Reference Format:
Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, Sören Auer, and Jens Lehmann. 2019. Uniform Access to Multiform Data Lakes using Semantic Technologies. In *The 21st International Conference on Information Integration and Web-based Applications & Services (iiWAS2019), December 2–4, 2019, Munich, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3366030.3366054

1 INTRODUCTION

The massive collection of data enabled by technological advancements achieved in the last decade introduced several data management challenges, hindering the effective data use and exploitation. For example, traditional highly-structured data management techniques and technologies face prohibitive performance bottlenecks when processing both large batches and fast streams of data. Furthermore, the multiplicity of data formats and data management systems make it very challenging to offer *ad hoc* uniform access to heterogeneous data. Both research and industry continue to investigate methods tackling those challenges along the Volume, Velocity and Variety Big Data dimensions [16]. Volume and Velocity can be addressed by distributing the heavy workloads and fast stream processing across commodity clusters. Variety is tackled by either enforcing centralization and converting the various formats to a unique representation that can be accessed in an *ad hoc* manner, or by creating a virtual middleware under which the heterogeneous formats are homogenized *on-the-fly* without data transformation or materialization.

Implementing the latter virtual approach, we propose Squerall [21] framework. Squerall allows the querying of large and heterogeneous sources in their original form, without prior transformation; an environment that is known as a Data Lake [7]. A Data Lake is a *schema-less* repository of original data stored *as is*, without requiring prior transformations or pre-processing. Originally coined within the Hadoop environment [7], Data Lake concept was quickly generalized to include other data sources, such as Amazon S3 file system or the various modern NoSQL stores (e.g., Cassandra, MongoDB, Neo4j, HBase, etc.). Squerall promotes the Data Lake abstraction to a *Semantic* Data Lake (SDL, briefly introduced in our previous work [2]), whereby distributed and heterogeneous data is glued together for interpretation using a common Semantic model, or ontology. This results in a virtual *(semantic) layer*, against which *ad hoc* queries can be posed.

The software implementation details behind Squerall were presented in [19, 21]. In this paper we add the following contributions:

- We provide a more formal and thorough description of the SDL's underlying concepts and components.
- We suggest six requirements that need to be fulfilled by a system implementing the SDL.
- We detail the distributed query execution aspects of the SDL.
- Present an empirical evaluation of Squerall, detailing the various involved stages of query processing, as well as the resource consumption along the memory, disk and network utilization.

The rest of the paper is structured as follows. Section 2 defines SDL's underlying concepts and requirements. Section 3 reminds about SDL architecture and our proposed implementation. In Section 4 we detail the query execution aspects of the SDL with projection on our implementation. We report in Section 5 on the experimental study conducted using our implementation. Finally, Section 7 concludes the paper and presents ideas for future work.

2 SEMANTIC DATA LAKE PRINCIPLES

Semantic Data Lake is an extension of the Data Lake supplying it with a semantic middleware, which allows the uniform access to original heterogeneous data sources. Squerall makes use of established Semantic Web techniques, such as ontologies, mappings and SPARQL queries. In the next, we define the concepts underlying the SDL, and suggest a set of requirements that SDL implementations must meet.

2.1 Running Example

In order to facilitate the understanding of subsequent definitions, we will use the SPARQL query in Listing 1 as a reference. `ns` denotes an example ontology where classes and properties are defined.

```
SELECT DISTINCT ?type ?price
WHERE {
    ?product    a               ns:Product .
    ?product    ns:hasType      ?type .
    ?product    ns:hasPrice     ?price .
    ?product    ns:hasProducer  ?producer .
    ?producer   a               ns:Producer .
    ?producer   ns:homepage     ?page .
    FILTER (?price > 1200)
} ORDER BY ?type LIMIT 10
```

Listing 1: Example SPARQL Query.

2.2 SDL Building Blocks

Definition 2.1 (Data Source). A data source refers to any data storage medium, e.g., plain file, structured file or a database. We denote a data source by d and the set of all data sources by $D = \{d_i\}$.

Definition 2.2 (Data Entities and Attributes). Entities are collections of data that share similar form and characteristics. These characteristics are encoded into *attributes*. For example, 'Product' is an entity of relational form, and is characterized by (Name, Type, Producer) attributes. We denote an entity by $e_x = \{a_i\}$, where x is the entity name and a_i are its attributes. A data source consists of one or more entities, $d = \{e_i\}$.

Definition 2.3 (Ontology). An ontology O is a set of terms that describe a common domain conceptualization. It principally defines classes C of concepts, and properties P about concepts, $O = C \cup P$. For example, `ns:Product` is the class of all products in an e-commerce system (`ns:` is ontology identifier, or *namespace*), of which `ns:hasType`, `ns:hasPrice` and `ns:hasProducer` are properties and `ns:Book` is a sub-class .

Definition 2.4 (Semantic Mapping). A semantic mapping is a relation linking two semantically-equivalent terms. We differentiate between two types of semantic mappings:

- *Entity mappings:* $m^{en} = (e, c)$ a relation mapping an entity e from d onto an ontology class c. For example, *(ProductTable, ns:Product)* is mapping the entity `ProductTable` from a Cassandra database to the class `ns:Product` of the ontology *ns*. M^{en} is the set of all entity mappings.
- *Attribute mappings:* $m^{at} = (a, p)$ a relation mapping an attribute a from an entity e onto an ontology property p. For example, *(price, ns:hasPrice)* is mapping attribute `price` of a Cassandra table to the ontology property `ns:hasPrice`. M^{at} is the set of all attribute mappings.

Definition 2.5 (Query). A query q is a statement in a query language used to extract entities by means of describing their attributes. We consider SPARQL as query language, which is used to query (RDF [17]) triple data (subject, property, object). In particular, we are concerned with the following fragment: the BGP section (Basic Graph Pattern), which is conjunctive set of triple patterns *(?subject, ?property, ?object)*, filtering, aggregation and solution modifiers (projection, limiting, ordering, and distinct).

Definition 2.6 (Query Star). A query star is a shorthand version of *subject-based star-shaped sub-BGP*, a set of triple patterns sharing the same subject. We denote a star by $st_x = \left\{t_i = (x, p_i, o_i) \mid t \in BGP_q\right\}$ where x is the shared subject and $BGP_q = \{(s_i, p_i, o_i) \mid p_i \in O\}$, i.e., triple patterns of a star (thus of the BGP) use properties from the ontology. We call *star variable* the subject shared by the star's triples. A star is *typed* if it has a triple with a typing property (e.g., `rdf:type` or `a`). For example, (`?product rdf:type ns:Product . ?product ns:hasPrice ?price . ?p ns:hasType ?type . ?product ns:hasProducer ?producer`) is a star of variable *product* and of type `ns:Product`.

Definition 2.7 (Query Star Connection). A star st_a is connected to another star st_b if st_a has a triple pattern with the object being the star variable of st_b, i.e., $connected(st_a, st_b) \rightarrow \exists t_i = (s_i, p_i, b) \in st_a$. For example, triple (`?product ns:hasProducer ?producer`) of $st_{product}$ connects $st_{product}$ with $st_{producer}$.

Definition 2.8 (Relevant Entities to Star). An entity e is relevant to a star st if it contains attributes a_i mapping to every triple property p_i of the star i.e., $relevant(e, st) \rightarrow \forall p_i \in prop(st) \exists a_j \in e \mid (p_i, a_j) \in M^{at}$, where *prop* is a relation returning the set of properties of a given star.

Definition 2.9 (Distributed Execution Environment). A Distributed Execution Environment, *DEE*, is the shared physical space where large data can be transformed, aggregated and joined together. It has an internal data structure that contained data comply with. For example, DEE can be a shared pool of memory in a cluster where data is organized in large distributed tables.

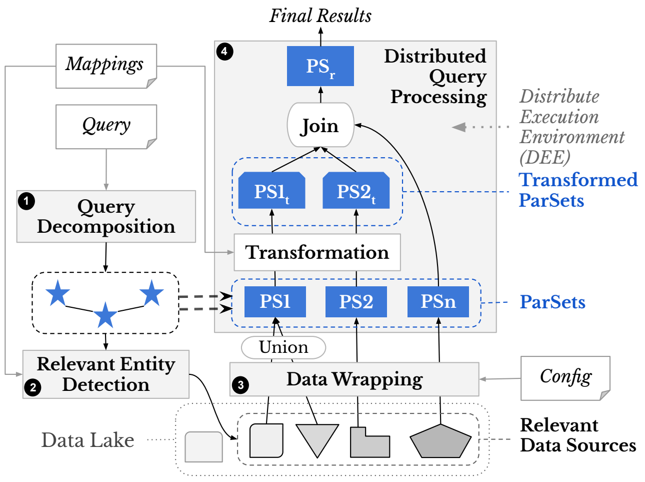


Figure 1: Squerall Architecture (Mappings, Query and Config are user inputs).

Definition 2.10 (ParSet (Parallel dataSet)). A ParSet is a data structure that is partitioned and distributed, and that is queried in parallel. ParSet has the following properties:

- It is created by loading star's relevant entities into the DEE.
- It has a well-defined data model, e.g., relational, graph, key-value.
- It is populated *on-the-fly* and not materialized, i.e., used only during query processing then cleared.

Its main purpose is to abstract away the structural differences between various varied data sources. We denote by PS_x the ParSet corresponding to the star st_x, $PS = \{st_x\}$ is the set of all ParSets.

Definition 2.11 (ParSet Schema). ParSet has a *schema* that is composed of the properties of its corresponding star, plus an *ID* that uniquely identifies ParSet's individual elements. For example, the schema of $PS_{product}$ is *{hasPrice, hasType, hasProducer, ID}*. To refer to a property p of a ParSet, the following notation is used $PS_{star}.p$, e.g., $PS_{product}.hasPrice$.

Definition 2.12 (Joinable ParSets). Joinable ParSets are ParSets that store inter-matching values. For example, if the ParSet has a tabular representation, joinable ParSets have the same meaning as joinable tables in relational algebra, i.e., tables sharing common attribute values. ParSets are incrementally joined in the course of a query, result of which is called *Results* ParSet, denoted $PS^{results}$

Definition 2.13 (Entity Wrapping). Entity wrapping is a function *wrap* that takes one or more relevant entities to a star and returns a ParSet. It loads entity elements (e.g., collection documents of a Document database) and organizes them according to ParSet's model and schema. $wrap : E^n \longrightarrow PS$.

2.3 SDL Requirements

Stemming from the Data Lake, Semantic Data Lake specializes in accessing large and heterogeneous data sources. Therefore, a *SDL-compliant* system must meet the following requirements:

- **R1. It should be able to access large-scale data sources**. Typical data sources inside a Data Lake rage from large plain files stored in a scale-out file/block storage infrastructure (e.g., Hadoop Distributed File System, Amazon S3) to scalable databases (e.g., NoSQL stores). Typical applications built to access a Data Lake are data- and compute-intensive. While a Data Lake may contain a centralized relational database, or small files fitting into a single-machine's memory, the primary focus of Data Lake is, by its definition [7], data and computations that grow beyond the capacity of single-machine deployments.
- **R2. It should be able to access heterogeneous sources**. The value of a Data Lake-accessing system increases with its ability to support as much data as possible. Thus, a SDL system should be able to access data of various forms, in plain-text files e.g., CSV, JSON, in structured file formats, e.g., Parquet, ORC, in databases, e.g., MongoDB, Cassandra, etc.
- **R3. Query execution should be performed in a distributed manner**. This is natural for multiple reasons. (1) Queries joining or aggregating only a sub-set of the large stored data may incur large intermediate results that can only be calculated and contained in multiple compute nodes. (2) As original data is already distributed across multiple nodes, e.g., in HDFS or high-availability MongoDB cluster, query intermediate results can only be stored distributedly and computed in parallel. (3) Many NoSQL stores, e.g., Cassandra and MongoDB, have dropped the support for certain query operations in favor of improving storage scalability and query performance [14, 22]. In order to join entities of even one same database, e.g., tables in Cassandra, they need to be loaded on-the-fly into an execution environment that supports join, e.g., using Apache Spark [32].
- **R4. It should be able to query heterogeneous data sources in a uniform manner**. One of the main purposes of adding a semantic layer on top of the various sources is to abstract away the structural differences found across Data Lake sources. This semantic middleware adds a schema to the originally schema-less repository of data, which can then be *uniformly* queried using a unique query language.
- **R5. It should query fresh original data without prior processing.** One of the ways Data Lake concept contrasts with the traditional Data Warehouse concept is that it does not enforce centralization by transforming the whole data into a new format and form; it rather queries directly the original version of the data. Querying transformed data compromises data freshness, i.e., a query returns an outdated response when data has changed or been added to the Data Lake after the transformation. Such pre-processing also includes indexing; once new data has been added to the Data Lake, queries will no longer access the original data, but excludes the new yet un-indexed data. Besides Data Lake requirements, index creation in the highly scalable environment of the Data Lake is an expensive operation that requires both storage space and time as it parses the full pool of data.
- **R6. It should have a mechanism to enable the join of originally disparate unrelated sources**. Data Lake is often created by dumping silos of data, i.e., data generated using separate applications but has the potential to be linked to derive new knowledge and drive new business decisions. As a result, Data Lake-contained data may not be readily joinable, so it is required to introduce changes at *query-time* to enable the join operation.

3 SDL ARCHITECTURE AND IMPLEMENTATION

We envision a SDL architecture to contain four core components (see Figure 1): Query Decomposition, Relevant Source Detection, Data Wrapping and Distributed Query Processing. We later on describe our implementation [21] of the architecture.

3.1 SDL Architecture

3.1.1 Query Decomposition. Once query is issued, it is analyzed to extract its contained query stars (Definition 2.6), as well as the connections between them (Definition 2.7).

3.1.2 Relevant Source Detection. For every star, mappings are visited to find relevant entities (i.e., having attribute mappings to every property of the star). If more than an entity is relevant, they are combined (union). If a star is typed with an ontology class, then only entities with entity mapping to that class are extracted.

3.1.3 Data Wrapping. Relevant entities are loaded as ParSets, one ParSet per query star. This component implements the Entity Wrapping function (Definition 2.13), namely loading entities under ParSet's model and schema, e.g., flattening of various data into tables.

3.1.4 Distributed Query Execution. Connections between stars detected in Query Decomposition step will be translated into joins between ParSets. Any operations on star properties (e.g., filtering) or results-wide operations (e.g., aggregation) are translated to operations on ParSets. Section 4 is reserved to detailing this component.

There are three inputs to the architecture: the query, semantic mappings and access information (e.g., username, password, cluster configurations, etc.)

3.2 SDL Implementation: Squerall

Squerall (from Semantically query all) is our implementation of the Semantic Data Lake architecture. Squerall makes use of state-of-the art Big Data technologies Apache Spark [32] and Presto [26] as query engines. Apache Spark is a general-purpose Big Data processing engine with modules for general batch processing, stream processing, Machine Leaning and structured data access using SQL. Presto is a SQL query engine allowing the joint querying of heterogeneous data sources using a single SQL query. Both engines primarily base their computations in-memory. In addition to their ability to query large-scale data, we chose Spark and Presto because they both have connectors (wrappers) allowing the access to a wide array of data sources. Using those connectors, we are able to avoid reinventing the wheel, and only resort to manually building a wrapper for a data source when no wrapper is available. We have explained and exemplified extending Squerall with an RDF connector in our another publication [20]. Dedicated graphical interfaces[1] guide Squerall users through the creation of the necessary inputs.

4 QUERY PROCESSING IN THE SDL

The pivotal property of the SDL distinguishing it from traditional centralized data integration and federated systems is its ability to query large-scale data sources in a distributed manner. This requires the incorporation of adapted or new approaches, which are able to overcome the data scale barrier. In this section we detail the various mechanisms and techniques underlying the distributed query execution in the SDL, and project it on our implementation.

[1]https://github.com/EIS-Bonn/Squerall-GUI

```
parset = wrapper(entity)
filter(parset)
aggregate(parset)
join(parset,otherParSet)
```

Listing 2: ParSets manipulation.

4.1 ParSets Formation

ParSets are the unit of computation in the SDL architecture. They are loaded from relevant data source entities and then filtered, transformed, joined and aggregated as necessary in the course of a query. They can be seen as *views* that are populated from underlying data entities and are available during the query execution time.

4.1.1 ParSets Language as 'Intermediate' Query Language. ParSets have a specific data model and, thus, can be queried using a specific query language, e.g., tabular model and SQL. The query language of the ParSets can then be used as an *intermediate* query language. Using the latter, it becomes possible to convert SDL's unique query language (SPARQL in our case) to ParSet's unique query language (SQL in our case), instead of the languages of every data source. Solving data integration problems using an intermediate or meta query language is an established approach throughout the literature. It has been used in recent works to access large and heterogeneous data sources (e.g., [4, 28]).

4.1.2 Data Source's Model to ParSet's Model. Wrappers, or as also known as connectors, are the components responsible for retrieving data from a data source and populating the ParSet, which includes model conversion. For example, if ParSet is of a tabular model and one source is of a graph model, the wrapper creates a tabular version of the graph data using the available data source access methods (e.g., API, HTTP requests, JDBC, query language, etc.) and necessary structural adaptations (e.g., flattening).

4.1.3 Interaction with ParSets. Once ParSets are created, they are used in two different ways, which we will next detail and relate to our implementation.

- **Manipulated ParSets.** This is the case of wrappers generating ParSets in a format that users can manipulate. For example, if the model of the ParSet is tabular, the returned ParSet would be a table that users can interact with by means of SQL-like functions. This approach is more flexible as users have direct access to and control over the ParSets using e.g., a programming language. However, knowledge about the latter is, thus, a requirement; Listing 2 illustrates this mechanism. For example, this mechanism is the one used in Spark-based Squerall. We use *Spark SQL API* to create DataFrames, which are the implementation of ParSets having a tabular model. These DataFrames can then be manipulated using various specialized Spark SQL-like functions, e.g., `filter`, `groupBy`, `orderBy`, `join`, etc.

```
SELECT C.type ... FROM cassandra.cdb.product C JOIN
       mongo.mdb.producer M ON C.producerID = M.ID
WHERE M.page="www.site.com" ...
```

Listing 3: A self-contained SQL query.

- **Self-Contained Query.** This is the case where wrappers do not create ParSets in a data structure that users can control and manipulate. Users can only write one universal self-contained *declarative* query (i.e., ask for what is needed and not how to obtain it) containing directly references to the needed data sources. An example of such a self-contained query is presented in Listing 3, where `product` and `producer` are two tables from Cassandra `cdb` and MongodB `mdb` databases, respectively. This mechanism is followed in our Presto-based Squerall, where we incrementally augment one self-contained SQL query starting from ParSets, details given in the next subsection.

This classification can categorize the various existing engines that can implement the SDL. For example, the first mechanism is applicable using Apache Flink[2], the second mechanism is applicable using Impala[3] or Drill[4].

4.2 ParSet Querying

4.2.1 From SPARQL to ParSet Operations. As previously explained (Definition 2.8), for every query star a ParSet is created from the entities relevant to that star. A ParSet has a schema that is composed of the properties of its corresponding star (Definition 2.11). In order to avoid property naming conflicts, i.e., multiple stars having an identical property, we form the components of ParSet's schema using the following template: *{star-variable}_{property}_{property-namespace}*, e.g., `product_hasType_ns`. Then, SPARQL query is translated into ParSets *operations* following the algorithm in Listing 5:

(1) For each star, relevant entities are extracted (lines 5 to 7).
(2) For each relevant entity, a ParSet (`oneParSet`) is loaded and changed in the following way. If SPARQL query contains conditions on a star property, equivalent ParSet filtering operations are executed (lines 10 and 11). If there are joinability transformations [21] (requirement 6), also equivalent transformation operations on ParSets are executed. Then, the loaded `oneParSet` is combined with the other entity ParSets of the same star (line 14). Finally, add the changed and combined `starParSet` to a list of all ParSets (line 15).
(3) Connections between query stars translate into joins between the respective ParSets, resulting in an array of join pairs e.g., *[(Product,Producer),(Product,Review)]* (line 17). As shown in the dedicated algorithm in Listing 6, results ParSet ($PS^{results}$) is created by iterating through the ParSet join pairs (operands) and incrementally apply the join between them (line 18).
(4) Finally, if SPARQL query contains aggregations or solution modifiers (project, limit, distinct or ordering) equivalent operations are executed on the results ParSet (lines 19 to 28).

Steps 1–3 are also illustrated in Table 1.

[2]https://flink.apache.org
[3]https://impala.apache.org
[4]https://drill.apache.org/

```
SELECT DISTINCT C.type, C.price
FROM cassandra.cdb.product C
JOIN mongo.mdb.producer M ON C.producerID = M.ID
WHERE C.price > 1200 ORDER BY C.type LIMIT 10
```

Listing 4: Generated self-contained SQL Query.

```
1   Input: SPARQL query q
2   Output: resultsParSet
3
4   allParSets = new ParSet()
5   foreach s in stars(q)
6     starParSet = new ParSet()
7     relevant-entities = extractRelevantEntities(s)
8     foreach e in relevant-entities // e one or more
9       oneParSet = loadParSet(q)
10      if containsConditions(s,q)
11        oneParSet = filter(oneParSet,conditions(s))
12      if containsTransformations(s,q)
13        oneParSet = transform(oneParSet,conditions(s)))
14      starParSet = combine(starParSet,oneParSet)
15    allParSets += starParSet
16
17  parSetJoinsArray = detectJoinPoints(allParSets,q)
18  resultsParSet = joinAll(parSetJoinsArray)
19  if containsGroupBy(b)
20    resultsParSet = aggregate(resultsParSet,q)
21  if containsOrderBy(q)
22    resultsParSet = order(resultsParSet,q)
23  if containsProjection(q)
24    resultsParSet = project(resultsParSet,q)
25  if containsDistinct(q)
26    resultsParSet = distinct(resultsParSet,q)
27  if containsLimit(q)
28    resultsParSet = limit(resultsParSet,q)
```

Listing 5: ParSet Querying Process (simplified).

4.2.2 ParSet Operations Execution. ParSet operations are marked with underline in the algorithm of Listing 5. The way those operations are executed depends on the interaction mechanism of the implementing sustem described in 4.1.3.

- **Manipulated ParSets.** This applies when ParSets are manually loaded and manipulated using execution engine functions. This is the case of DataFrames, the implementation of ParSets in Spark, which undergo *Spark transformations*, e.g., `map`, `filter`, `join`, `groupByKey`, `sortByKey`, etc. If we consider `oneParSet` of line 9 Listing 5 as a DataFrame, then every ParSet operation can be implemented using an equivalent Spark transformation.
- **Self-Contained Query.** As in this case a high-level declarative query is generated, there are no explicit operations to manually run in a sequence. Rather, ParSet operations of Listing 5 create and gradually *augment* a query, e.g., SQL query in our Presto-based implementation. For example, in the query of Listing 1, there is a condition filter `?price > 1200`, ParSet operation of line 11 augments the query with `WHERE price > 1200`. Similarly, the query parts `ORDER BY ?type` and `LIMIT` represented by ParSet operations `order` (line 22) and `limit` (line 28) augment the query by `ORDER BY C.type LIMIT 10`. Full generated SQL query of the query in Listing 1 is presented in Listing 4. Query augmentation is a known technique in query translation literature, e.g., [8].

```
Input: ParSetJoinsArray // Pairs [ParSet,ParSet]
Output: ResultsParSet // ParSet joining all ParSets

ResultsParSet = ParSetJoinsArray[0] // 1st pair
foreach currentPair in ParSetJoinsArray
  if joinableWith(currentPair,ResultsParSet)
    ResultsParSet = join(ResultsParSet,currentPair)
  else PendingJoinsQueue += currentPair
// Next, iterate through PendingJoinsQueue
// similarly to ParSetJoinsArray
```

Listing 6: JoinAll - Iterative ParSets Join.

$Star_{product}$: ?product a ns:Product . ?product ns:hasType ?type . ?product ns:hasPrice ?price .	**Mappings:** Product ⟶ ns:Product type ⟶ ns:hasType price ⟶ ns:hasPrice
↳ $PS_{Product}$: SELECT type AS product_hasType_ns, price AS product_hasPrice_ns	
$Star_{producer}$: ?producer a ns:Producer . ?producer ns:homepage ?page .	**Mappings:** Producer ⟶ ns:Producer website ⟶ ns:homepage
↳ $PS_{Producer}$: SELECT website AS producer_homepage_ns	
$Star_{product}$ (follow up) ?product ns:hasProducer ?producer	
$PS^{results}$: Product JOIN Producer ON Product.product_hasProducer_ns = Producer.ID	

Table 1: ParSets generation from SPARQL & mappings.

4.2.3 Optimization Strategies. In order to optimize query execution time, we have designed the query processing algorithm (Listing 5) in such a way that we reduce as much data as soon as possible, especially before the cross-ParSet join is executed. There are three locations where this is applied:

(1) We push the query operations that effect only the elements of a single ParSet to the ParSet itself, not until obtaining *results ParSet*. Concretely, we execute the `filter` and `transform` operations before the `join`. Aggregation and the other solution modifiers are left to the final results ParSet, as those operations have results-wide effect.
(2) `Filter` operation runs before `transform` (line 11 then 13). This is because the transform affects the attribute values (which will later participate in a join), so if they are reduced by the filter, less data will have to be transformed (then joined).
(3) We leverage filter push-down optimization offered by both Spark and Presto. Namely, we allow the engines to filter data, whenever possible (e.g., possible with Parquet and not with CSV), even before loading them into ParSets.

Optimization decisions 1 and 2 only apply to Spark implementation, where we can manipulate DataFrames. In Presto implementation, we do not have control over the internal data structures implementing the ParSets. In either case, both Spark and Presto apply internally built-in optimization strategies. For example, Spark analyses the query and decides on the DataFrames join order, e.g., join between a Parquet entity and a Cassandra entity, then results of which are joined with a CSV entity.

	Product	Offer	Review	Person	Producer
# of tuples	**Cassandra**	**MongoDB**	**Parquet**	**CSV**	**MySQL**
Scale 1 (0.5M)	0.5M	10M	5M	~26K	~10K
Scale 2 (1.5M)	1.5M	30M	15M	~77K	~30K
Scale 3 (5M)	5M	100M	50M	~2.6M	~100K

Table 2: Data loaded and corresponding number of tuples.

5 EVALUATION

In this section we will report on the empirical study that we have conducted in order to evaluate Squerall's performance with regard to various metrics. We set to answer the following questions:

- **RQ1:** What is the query performance when Spark and Presto are used as underlying Squerall query engine?
- **RQ2:** What is effect of query analysis and relevant source detection on the overall query execution time?
- **RQ3:** What is the performance of Squerall when increasing data sizes are queried?
- **RQ4:** Is there a *direct* impact of involving more data sources in a join query?
- **RQ5:** What is the resource consumption (CPU, memory, data transfer) of the run queries?

5.1 Data and Queries

As we have explored in [21], there is still no dedicated benchmark for evaluating Semantic Data Lake implementations, i.e., querying large and heterogeneous sources right on their original form using SPARQL. We resort to using and adapting BSBM Benchmark [3], which is originally designed to evaluate the performance of RDF triple stores with SPARQL-to-SQL rewriters. As Squerall currently supports five data sources: Cassandra, MongoDB, Parquet, CSV, and JDBC, we have chosen to load five BSBM-generated tables into the five supported data sources as shown in Table 2. We generate three scales: 500k, 1.5M and 5M (in terms of number of products), which we will refer to in the following as *Scale 1*, *Scale 2* and *Scale 3*, respectively.

We have adapted the original BSBM queries so they involve only the tables we have effectively used (e.g., *Vendor* table was not populated), and not include unsupported SPARQL constructs like, e.g., `DESCRIBE`, `CONSTRUCT`. This resulted in nine queries[5] joining different tables with various SPARQL query operations, see Table 3.

5.2 Metrics

Our main objective is to evaluate *Query Execution Time*. In particular, we observe Squerall's performance with (1) increasing data sizes, (2) increasing data sources. We evaluate the effect of *Query Analyses* and *Relevant Source Detection* on the overall query execution time. We leave time markers at the beginning and end of each phase. Considering the distributed nature of Squerall, we also include a set of system-related metrics, following the framework presented in [13], e.g., average CPU consumption and spikes, memory usage, data read from disk and transferred across the network. We run every query three times on a *cold cache*. As we report on the impact of every phase on the total query time, we cannot calculate the average of the recorded times. Rather, we calculate the sum of

[5]Available at: https://github.com/EIS-Bonn/Squerall/tree/master/evaluation/input_files/queries

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q10
Product	✓	✓	✓	✓	✓	✓	✓	✓	✓
Offer		✓		✓	✓	✓	✓		✓
Review	✓		✓	✓	✓	✓	✓	✓	
Person							✓	✓	
Producer	✓			✓				✓	✓
`FILTER`	✓1		✓2	✓1	✓3	✓1r	✓2	✓1	✓3
`ORDER BY`	✓		✓	✓	✓			✓	✓
`LIMIT`	✓		✓	✓	✓			✓	✓
`DISTINCT`	✓			✓	✓		✓	✓	✓

Table 3: Tables and query operations involved in the Queries. Numbers in FILTER are the number of conditions involved and r denotes the presence of a regex filter.

the overall query times of the nine queries of the three runs and take the run with the median sum value. Threshold is set to 3600s.

5.3 Environment

All queries are run on a cluster of three nodes having DELL PowerEdge R815, 2x AMD Opteron 6376 (16 cores) CPU and 256GB RAM. No caching or engine optimizations tuning were exploited.

5.4 Results & Discussions

Our extensive literature review reveals no single work that was openly available and that supported all the five sources and the SPARQL fragment that we support. Thus, we compare Squerall performance when Spark and Presto are used as query engines.

5.4.1 Performance: The columns in Figure 2 show the query execution time divided into three phases (each with a distinct color):

(1) **Query Analysis:** Time taken to extract the stars, detect joins between stars, and various query operations linked to every star e.g., filtering, ordering, aggregation, etc.

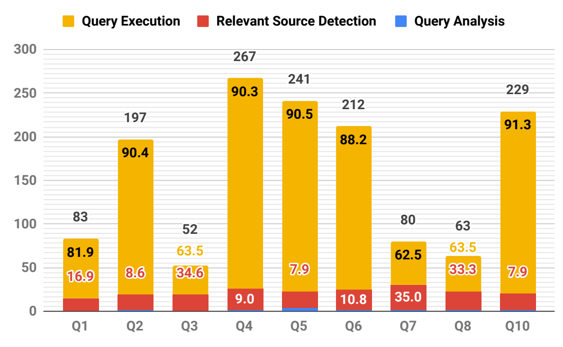
(2) **Relevant Source Detection:** Time taken to visit the mappings and find relevant entities by matching SPARQL star types and properties against data entities and attributes.

(3) **Query Execution:** Time taken by the underlying engine (Spark or Presto) to effectively query the (relevant) data sources and collect the results. This includes loading the relevant entities into ParSets (or sub-sets of them if filters are pushed down to the source), performing ParSet-related operations e.g., filtering, executing joins between ParSets, and finally performing results-wide operations e.g., ordering, aggregation, and de-duplication.

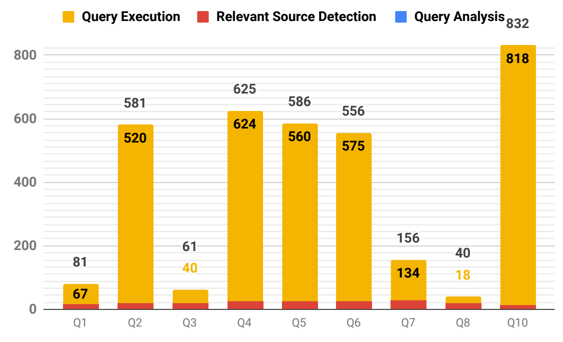
The results[6] presented in Figures 2 and 3 suggest the following:

- Presto-based Squarall is faster than Spark-based in most cases except for Q3 and Q10 at Scale 1; it has comparable to slightly lower performance in Q1 and Q8 at Scale 2. Presto is built and optimized for running *ad hoc* analytical SQL queries. Spark on the other hand is a general-purpose engine with a SQL layer, which builds on Spark's core in-memory structures that were not originally designed for ad hoc querying. Spark is optimized for fault tolerance and query recovery, in contrast to Presto which is optimized for speed in favor of weaker query resiliency. This

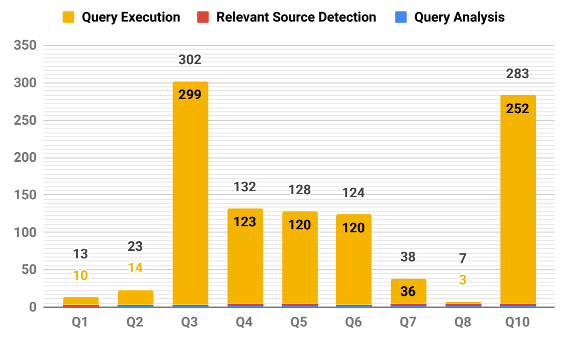
[6]Also available online: https://github.com/EIS-Bonn/Squerall/tree/master/evaluation



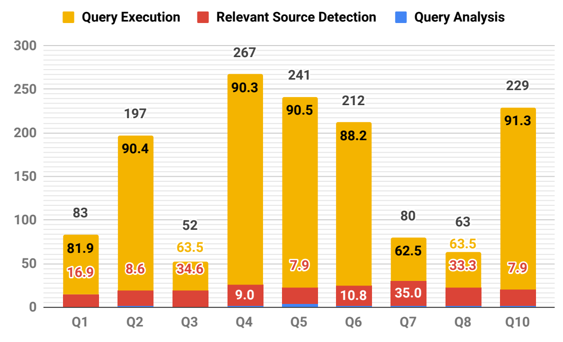
(a) Spark Scale 1.



(b) Spark Scale 2.

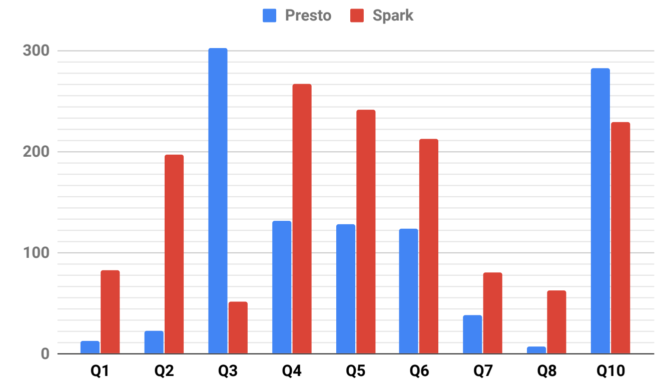


(c) Presto Scale 1.

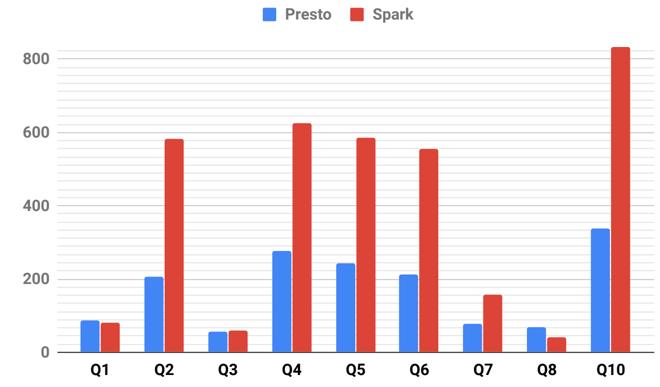


(d) Presto Scale 2.

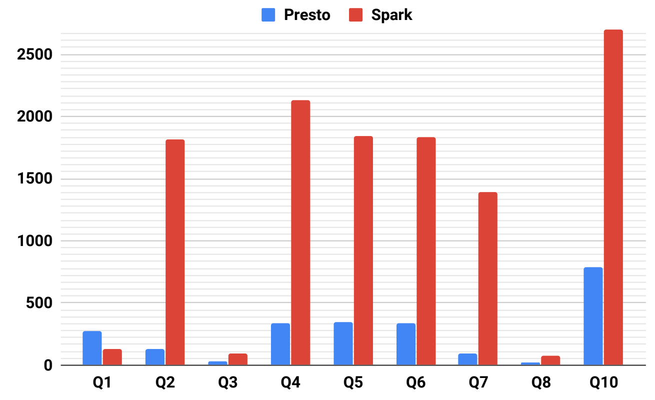
Figure 2: Stacked view of the times (seconds): (1) Query Analyses, (2) Relevant Source Detection, (3) Query Execution, sum of which is the total execution time (labels on top).



(a) Scale 1.



(b) Scale 2.



(c) Scale 3.

Figure 3: Query Execution Time (seconds): comparison Spark-based vs. Presto-based Squerall.

explains Presto's superiority. Spark, on the other hand, performs better on long-running complex queries, which is not the case for our benchmark queries. **(RQ1)**

- Query Analysis time is negligible, in all the cases it did not exceed 4 seconds, ranging from < 1% to 8% of the total execution time. Relevant Source Detection time varies with the queries and scales. It ranges from 0.3% (Q3 Presto Scale 1) to 38.6% (Q8 Spark Scale 2). It is however homogeneous across the queries of the same scale and query engine. Query Execution time is what dominates the total query execution time in all the cases. It ranges from %42.9 (Q8 Presto Scale 1) to 99% (Q3 Spark Scale 2), with most percentages being about or above 90%, regardless of the total execution time. Both Query Analysis and Relevant Source Detection depend on the query not the data, so their performance is not affected by the size of the data. That is why we did not include the numbers for the performance at Scale 3. **(RQ2)**
- Increasing the size of the queried data did not deteriorate query performance. Co-relating query time and data scale indicates that performance is proportional to data size. In addition, all the queries finished within the threshold. **(RQ3)**
- The number of joins did not have a decisive impact on query performance, it rather should be taken in consideration with other factors, e.g., size of involved data, presence of filters. For example, Q2 joins only two data sources but has comparable performance with Q5 and Q6, which join three. This may be due to the presence of filtering in Q5 and Q6. Q7 and Q8 involve four data sources, yet they are among the fastest queries. This is because they involve the small entities *Person* and *Producer*, which significantly reduce intermediate results to join. With four data sources to join, Q4 is among the most expensive. This can be attributed to the fact that the filter on *Product* is not selective (`?p1 > 630`), in contrast to Q7 and Q8 (`?product = 9`). Although, the three-source join Q10 involves the small entity *Producer*, it is the most expensive; this can be attributed to the very unselective product filter it has (`?product > 9`). **(RQ4)**

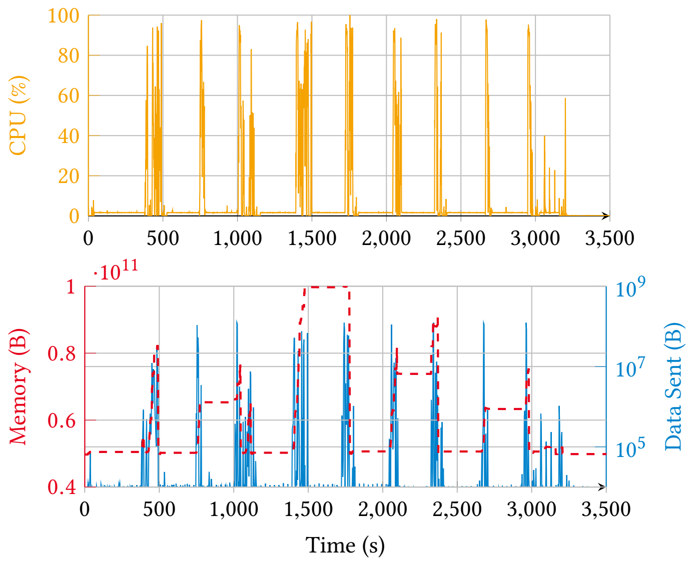


Figure 4: Resources of node2 with Spark Scale 2.

5.4.2 *Resource Consumption:* We record (1) CPU utilization by calculating its average percentage usage as well as the sum of times it reached 80% and 90%, (2) memory used in GB, (3) data sent across the network in GB, and (4) data read from disk in GB (see Table 4). We could make the following observations **(RQ5)** (see Table 4):

Metrics	Spark			Presto		
	Node 1	Node 2	Node 3	Node 1	Node 2	Node 3
CPU average (%)	4.327	7.141	4.327	2.283	2.858	2.283
Time above 90% CPU (s)	9	71	9	0	2	0
Time above 80% CPU (s)	19	119	19	0	5	0
Max memory (GB)	98.4	100	98.4	99.5	99.7	99.5
Data sent (GB)	4.5	6.3	4.5	5.4	8.5	5.4
Data received (GB)	3.5	3.0	3.5	8.4	4.6	8.4
Data read (GB)	9.6	5.6	9.6	1.9	0.2	1.9

Table 4: Resource Consumption by Spark and Presto across the three nodes on Scale 2.

- Although the average CPU is low (below 10%), monitoring the 90% and 80% spikes shows that there were lots of instants where the CPU was almost fully used. The latter applies to Spark only, as Presto had far less 80%, 90% and average CPU usage, making it a lot less CPU-greedy than Spark.
- From the CPU average, it still holds that the queries overall are not CPU-intensive, CPU is in most of the time idle, the query time is then divided between loading and transferring (shuffling) data between the nodes.
- The total memory reserved, 250GB per node, was not fully used; at most ≈100GB was used by both Spark and Presto.
- Presto reads less data from disk (Data read), this possibly reflects its effectiveness at filtering irrelevant data (through predicate push down) and already starting query processing with reduced intermediate results.

Moreover, in Figure 4, we represent in function of time the CPU utilization (in orange), the memory usage (in dashed red) and the data sent over the network[7] (in light blue) second by second during the complete run of the benchmark at Scale 2 with Spark[8]. The noticed curve stresses correspond to the nine evaluated queries. We observe the following:

- CPU utilization curves change simultaneously with Data Sent curve, which implies that there is a constant data movement throughout query processing. Data movement is mainly due to the join operation that exists in all queries.
- Changes in all the three represented metrics are almost simultaneous throughout all query executions, e.g., there is no apparent delay between data sent or memory usage and the beginning of the CPU computation.
- Memory activity is correlated with the data (received and sent) activities, all changes of memory usage levels are correlated with a high network activity.
- Unexpectedly, the memory usage seems to remain *stable* between two consecutive query executions, which is in contradiction with our experimental protocol that applies cache clearing between each query run. In practice, this can be attributed to the fact that even if some blocks of memory are *freed*, they remain shown as *used* as long as they are not used again by another process following the common UNIX strategy of memory management[9].

[7]For clarity reason we do not add here the *received data* traffic since it is completely synchronized with the *sent data.*
[8]Note: the case of node2 with Spark is representative of the other nodes and configurations.
[9]For more details, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=34e431b0ae398fc54ea69ff85ec700722c9da773

6 RELATED WORK

Providing a uniform access to multiform data sources is a promising research area that continues to attract a lot of attention. Our study's scope is accessing large and heterogeneous data sources including the famous NoSQL family, around which also plenty work have been published. Different efforts used different approaches to provide the unified access interface to the data.

- *Using SPARQL query language.* Optique [12] is a platform that allows to access both static and dynamic data sources (streams). It implements a large-scale application using the open-source Ontop, code-source of which is unfortunately not available. Ontario [9] is an implementation of the Semantic Data Lake; however, it was applied to small and few data sources. [6] also accesses heterogeneous data but using only simple queries with minimal and centralized join support.
- *Using SQL query language.* [5] suggests an intermediate query language that transforms SQL to Java methods accessing NoSQL databases. A dedicated mapping language to express access links to NoSQL databases was defined. The underlying approach, e.g., join processing, is not explained and the prototype is not evaluated. [4, 28, 29] are efforts aiming at bridging the gap between relational and NoSQL databases, but evaluating with only individual NoSQL stores and in cases only with small data sizes.
- *Using access methods:* [1] allows direct access to NoSQL databases using *get*, *put* and *delete* primitives based on a suggested unified *programming* model. Cross-database join was not addressed. [24] enables running CRUD operations over NoSQL stores. The authors extend their approach to support joins [25] but not in a scalable way; the approach performs joins locally if involved data is located in the same database and the latter supports join, otherwise, data is moved to another capable database.
- *Using a hybrid query language:* [23] suggests a generic query language based on SQL and JSON, called SQL++. It tries to cover the capabilities of various NoSQL query languages. However, here again, only one data source, MongoDB, was used to showcase the query capabilities. In [15], a SQL-like language is suggested invoking the native query interfaces of relational and NoSQL databases. Their general architecture is distributed; however, we were not able to verify whether intra-source join is also distributed in absence of the source-code. In both efforts, users are expected to learn syntaxes of other languages in addition to SQL's.

There are two other solution families that are similar to the Semantic Data Lake concept in that heterogeneous stores are to be accessed. However, they differ in the scope of data sources to access or the access mechanism. The first family is represented by the solutions mapping relational databases to RDF [27], and Ontology-Based Data Access over relational databases [31], e.g., Ontop, Morph, Ultrawrap, Mastro, Stardog. These solutions are not designed to query large-scale data sources, e.g., NoSQL stores or HDFS. The second family is represented by the so-called *polystore* solutions, which address large-scale sources, but involves the movement of data across the data sources, or the data itself is duplicated across the sources, e.g., [10, 11, 30]. The task is to find which store answers best a given query (analytical, transactional, etc.) or which store to move all/part of the data to.

All surveyed efforts support a few data sources (1-3) with limited query capabilities, e.g., not supporting (distributed) joins. More importantly, wrappers are manually created or hard-coded. Further, in order to support as many data sources as possible, we choose not reinvent the wheel and leverage the wrappers offered by or built for the variety of existing engines. This makes it the solution with the broadest support of Big Data Variety dimension in terms of data source types. Further, Squerall has among the richest query capabilities[10], e.g., join, aggregation, and solution modifiers.

7 CONCLUSION & FUTURE WORK

In this paper we provided a (semi)formal description of the terms and principles underlying the Semantic Data Lake concept, and suggested six requirements that must be met for an implemented system to be SDL-compliant. Further, we provided detailed description of the various query execution mechanisms underlying the SDL. Furthermore, we have conducted experiments to evaluate Squerall's query execution performance throughout various execution stages. In the future we plan to enrich Squerall's query capability by supporting `OPTIONAL` constructs and sub-queries. After having been integrated in the SANSA Stack [18], Squerall is already being internally used by a large industrial company. As a result, we also plan to evaluate Squerall's performance on real-world data in addition to the synthetic data used in this study. Finally, we intend to leverage more thoroughly the optimization techniques offered by the query engines, e.g., various types of join algorithms, partitioning, and statistics.

ACKNOWLEDGMENTS

This work is partly supported by the EU H2020 projects BETTER (GA 776280) and QualiChain (GA 822404); and by the ADAPT Centre for Digital Content Technology funded under the SFI Research Centres Programme (Grant 13/RC/2106) and co-funded under the European Regional Development Fund.

REFERENCES

[1] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. 2012. Uniform Access to Non-relational Database Systems: The SOS Platform.. In *In CAiSE*, Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza (Eds.), Vol. 7328. Springer, 160–174.

[2] Sören Auer, Simon Scerri, Aad Versteden, Erika Pauwels, Stasinos Konstantopoulos, Jens Lehmann, Hajira Jabeen, Ivan Ermilov, Gezim Sejdiu, Mohamed Nadjib Mami, et al. 2017. The BigDataEurope platform–supporting the variety dimension of big data. In *International Conference on Web Engineering*. Springer, 41–59.

[3] Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)* 5, 2 (2009), 1–24.

[4] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao. 2018. A Generalized Framework for Ontology-Based Data Access. In *International Conference of the Italian Association for Artificial Intelligence*. Springer, 166–180.

[5] Olivier Curé, Robin Hecht, Chan Le Duc, and Myriam Lamolle. 2011. Data integration over NoSQL stores using access path based mappings. In *International Conference on Database and Expert Systems Applications*. Springer, 481–495.

[6] Oliver Curé, Fadhela Kerdjoudj, David Faye, Chan Le Duc, and Myriam Lamolle. 2013. On the potential integration of an ontology-based data access approach in NoSQL stores. *International Journal of Distributed Systems and Technologies (IJDST)* 4, 3 (2013), 17–30.

[7] James Dixon. 2010. Pentaho, Hadoop, and Data Lakes. (2010). https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes Online; accessed 06-August-2019.

[10]Details about the full fragment supported can be found at https://github.com/EIS-Bonn/Squerall/tree/master/evaluation

[8] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z Meral Ozsoyoglu. 2009. A complete translation from SPARQL into efficient SQL. In *Proceedings of the International Database Engineering & Applications Symposium*. ACM, 31–42.

[9] Kemele M Endris, Philipp D Rohde, Maria-Esther Vidal, and Sören Auer. 2019. Ontario: Federated Query Processing Against a Semantic Data Lake. In *International Conference on Database and Expert Systems Applications*. Springer, 379–395.

[10] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. 2016. The bigdawg polystore system and architecture. In *High Performance Extreme Computing Conference*. IEEE, 1–6.

[11] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 452–461.

[12] Martin Giese, Ahmet Soylu, Guillermo Vega-Gorgojo, Arild Waaler, Peter Haase, Ernesto Jiménez-Ruiz, Davide Lanti, Martín Rezk, Guohui Xiao, Özgür Özçep, et al. 2015. Optique: Zooming in on big data. *Computer* 48, 3 (2015), 60–67.

[13] Damien Graux, Louis Jachiet, Pierre Geneves, and Nabil Layaïda. 2018. A Multi-Criteria Experimental Ranking of Distributed SPARQL Evaluators. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 693–702.

[14] Eben Hewitt. 2010. *Cassandra: the definitive guide.* " O'Reilly Media, Inc.".

[15] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. 2016. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases* 34, 4 (2016), 463–503.

[16] Doug Laney. 2012. Deja VVVu: others claiming Gartner's construct for big data. *Gartner Blog, Jan* 14 (2012).

[17] Ora Lassila, Ralph R Swick, et al. 1998. Resource description framework (RDF) model and syntax specification. (1998).

[18] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Nilesh Chakraborty, Muhammad Saleem, and Axel-Cyrille Ngonga Ngomo. 2017. Distributed Semantic Analytics using the SANSA Stack. In *ISWC*. Springer, 147–155.

[19] Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, and Sören Auer. 2019. Querying Data Lakes using Spark and Presto. In *The World Wide Web Conference*. ACM, 3574–3578.

[20] Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, Sören Auer, and Jens Lehman. 2019. How to feed the Squerall with RDF and other data nuts? *Proceedings of 18th International Semantic Web Conference (Poster & Demo Track)* (2019).

[21] Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, Sören Auer, and Jens Lehmann. 2019. Squerall: Virtual Ontology-Based Access to Heterogeneous and Large Data Sources. *Proceedings of 18th International Semantic Web Conference* (2019).

[22] Franck Michel, Catherine Faron-Zucker, and Johan Montagnat. 2016. A mapping-based method to query MongoDB documents with SPARQL. In *International Conference on Database and Expert Systems Applications*. Springer, 52–67.

[23] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases. *CoRR, abs/1405.3631* (2014).

[24] Rami Sellami, Sami Bhiri, and Bruno Defude. 2016. Supporting Multi Data Stores Applications in Cloud Environments. *IEEE Trans. Services Computing* 9, 1 (2016), 59–71.

[25] Rami Sellami and Bruno Defude. 2018. Complex Queries Optimization and Evaluation over Relational and NoSQL Data Stores in Cloud Environments. *IEEE Trans. Big Data* 4, 2 (2018), 217–230.

[26] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

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

[28] Jörg Unbehauen and Michael Martin. 2016. Executing SPARQL queries over Mapped Document Stores with SparqlMap-M. In *12th Int. Conf. on Semantic Systems*.

[29] Ágnes Vathy-Fogarassy and Tamás Hugyák. 2017. Uniform data access platform for SQL and NoSQL database systems. *Information Systems* 69 (2017), 93–105.

[30] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2017. Icarus: Towards a multistore database system. *2017 IEEE International Conference on Big Data (Big Data)* (2017), 2490–2499.

[31] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. 2018. Ontology-based data access: A survey. IJCAI.

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