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

Mohamed Nadjib Mami[1,2] (iD), Damien Graux[2,3], Simon Scerri[1,2], Hajira Jabeen[1], Sören Auer[4], and Jens Lehmann[1,2]

[1] Smart Data Analytics (SDA) Group, Bonn University, Germany
[2] Enterprise Information Systems, Fraunhofer IAIS, Germany
[3] ADAPT Centre, Trinity College of Dublin, Ireland
[4] TIB & Hannover University, Germany
{mami,scerri,jabeen,jens.lehmann}@cs.uni-bonn.de
damien.graux@iais.fraunhofer.de, auer@l3s.de

Abstract. The last two decades witnessed a remarkable evolution in terms of data formats, modalities, and storage capabilities. Instead of having to adapt one's application needs to the, earlier limited, available storage options, today there is a wide array of options to choose from to best meet an application's needs. This has resulted in vast amounts of data available in a variety of forms and formats which, if interlinked and jointly queried, can generate valuable knowledge and insights. In this article, we describe Squerall: a framework that builds on the principles of Ontology-Based Data Access (OBDA) to enable the querying of disparate heterogeneous sources using a unique query language, SPARQL. In Squerall, original data is queried on-the-fly without prior data materialization or transformation. In particular, Squerall allows the aggregation and joining of large data in a distributed manner. Squerall supports out-of-the-box five data sources and moreover, it can be programmatically extended to cover more sources and incorporate new query engines. The framework provides user interfaces for the creation of necessary inputs, as well as guiding non-SPARQL experts to write SPARQL queries. Squerall is integrated into the popular SANSA stack and available as open-source software via GitHub and as a Docker image.

Software Framework. https://eis-bonn.github.io/Squerall

1 Introduction

For over four decades, relational data management remained a dominant paradigm for storing and managing structured data. However, the advent of extremely large-scale applications revealed the weakness of relational data management at dynamically and horizontally scaling the storage and querying of massive amounts of data. This prompted a paradigm shift, calling for a new breed of databases capable of managing large data volumes without jeopardising query performance by reducing query expressivity and consistency requirements. Since 2008 to date, a wide array of so-called *non-relational* or *NoSQL* (Not only SQL)

databases emerged (e.g., Cassandra, MongoDB, Couchbase, Neo4j). This heterogeneity contributed to one of the main Big Data challenges: variety. The integration of heterogeneous data is the key rational for the development of semantic technologies over the past two decades. Local data schemata are *mapped* to global ontology terms, using *mapping languages* that have been standardized for a number of popular data representations, e.g., relational data, JSON, CSV or XML. Heterogeneous data can then be accessed in a *uniform* manner by means of queries in a standardized query language, SPARQL [15], employing terms from the ontology. Such data access is commonly referred to as Ontology-Based Data Access (OBDA) [23]. The term *Data Lake* [11] refers to the schema-less pool of heterogeneous and large data residing in its original formats on a horizontally-scalable cluster infrastructure. It comprises databases (e.g., NoSQL stores) or scale-out file/block storage infrastructure (e.g., Hadoop Distributed File System), and requires dealing with the original data without *prior physical transformation or pre-processing*. After emerging in industry, the concept has increasingly been discussed in the literature [24,21,32]. The integration of semantic technologies into Data Lakes led to the SEMANTIC DATA LAKE concept, briefly introduced in our earlier work [2]. By adopting the OBDA paradigm to the NoSQL and Data Lake technology space, we realize the Semantic Data Lake concept and present in this article a comprehensive implementation.

Implementing an OBDA architecture atop Big Data raises three challenges:

1. *Query translation.* SPARQL queries must be translated into the query language of each of the respective data sources. A generic and dynamic translation between data models is challenging (even impossible in some cases e.g., join operations are unsupported in Cassandra and MongoDB [20]).
2. *Federated Query Execution.* In Big Data scenarios it is common to have non-selective queries with large intermediate results, so joining or aggregation cannot be performed on a single node, but only distributed across a cluster.
3. *Data silos.* Data coming from various sources can be connected to generate new insights, but it may not be readily 'joinable' (cf. definition below).

To target the aforementioned challenges we build Squerall [19], an extensible framework for querying Data Lakes.

– It allows *ad hoc* querying of large and heterogeneous data sources *virtually* without any data transformation or materialization.
– It allows the *distributed* query execution, in particular the joining of disparate heterogeneous sources.
– It enables users to declare query-time *transformations* for altering join keys and thus making data *joinable*.
– Squerall integrates the state-of-the-art Big Data engines Apache Spark and Presto with the semantic technologies RML and FnO.

The article is structured as follows. Squerall architecture is presented in Section 2 and its implementation in Section 3. The performance is evaluated in Section 4 and its sustainability, availability and extensibility aspects are discussed in Section 5. Related Work is presented in Section 6 and Section 7 concludes with an outlook on possible future work.
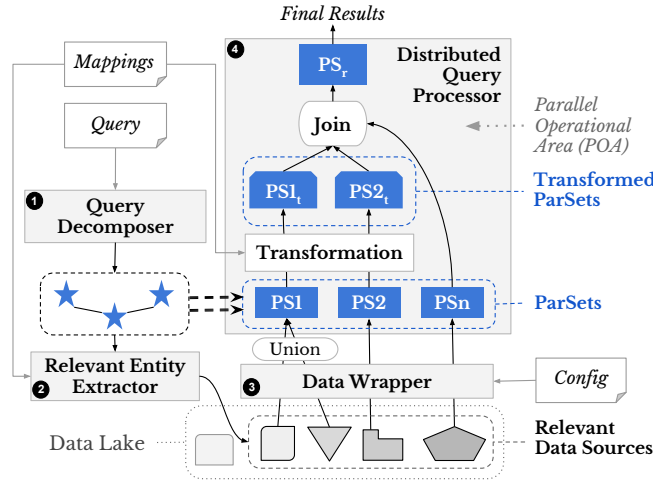
Fig. 1: Squerall Architecture (Mappings, Query and Config are user inputs).

2   Architecture

Squerall (Semantically query all) is built following the OBDA principles [23]. The latter were originally devised for accessing relational data but do not impose a restriction on the type or size of data it deals with. We project them to large and heterogeneous data sources contained in a Data Lake.

2.1   Preliminaries

In order to guide the subsequent discussions, we first define the following terms:
Data Attribute represents all concepts used by data sources to characterize a particular stored datum, e.g., a *column* in a tabular database like Cassandra, or a *field* in a document database like MongoDB.

Data Entity and Relevant Entity: an entity represents all concepts that are used by data sources to group together similar data, e.g., a *table* in a tabular database or a *collection* in a document database. An entity has one or multiple data attributes. An entity is relevant to a query if it contains information matching a part of the query (similarly found in federated systems, e.g., [25]).

ParSet and Joinable ParSets: from *Parallel dataSet*, ParSet refers to a data structure that is partitioned and distributed, and that is queried in parallel. ParSet is populated on-the-fly, and not materialized. Joinable ParSets are ParSets that store inter-matching values. For example, if the ParSet has a tabular representation, it has the same meaning as joinable tables in relational algebra, i.e., tables sharing common attribute values.

Parallel Operational Area (POA) is the parallel distributed environment where ParSets are loaded, joined and transformed, in response to a query. It has its internal data structure, which ParSets comply with.

Data Source refers to any storage medium, e.g., plain file storage or a database.
Data Lake is a repository of multiple data sources where data is stored and accessed directly in its original form and format, without prior transformation.
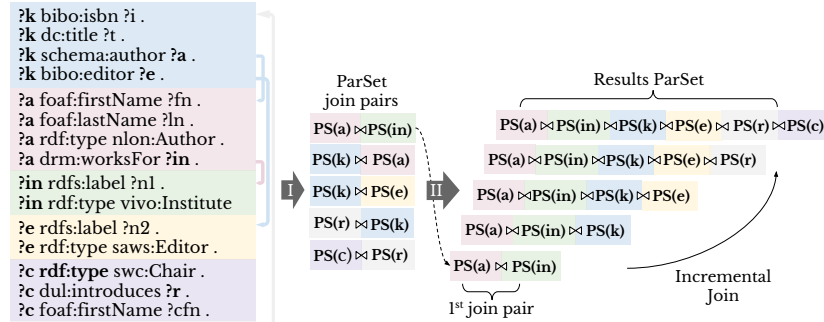
Fig. 2: ParSets extraction and Join (for clarity ParSet(x) is shortened to PS(x)).

2.2 OBDA Building Blocks

A typical OBDA system is composed of five main components:

Data. A Data Lake is a collection of multiple heterogeneous data sources, be it raw files (e.g., CSV), structured file formats (e.g., Parquet) or databases (e.g., MongoDB). Currently, Squerall does not support unstructured data but it can be part of the Data Lake.

Schema. A Data Lake is by definition a schema-less repository of data. Schemata exist at the level of the individual data sources.

Ontology. Ontologies are used to define a common domain conceptualization across Data Lake entities. At least class and properties definition is required.

Mappings. Mappings are association links between elements of the data schema and ontology terms (i.e., classes and properties). Three mapping elements need to be provided as input for an entity to be queried:

1. **Class mapping:** associates an entity to an ontology class.
2. **Property mappings:** associate entity attributes to ontology properties.
3. **Entity ID:** specifies an attribute to be used as identifier of the entity.

For example, `Author(AID,first_name,last_name)` is an entity in a table of a Cassandra database. In order to enable finding this entity, the user must provide the three mapping elements. As example, (1) *Class mapping:* (Author, nlon:Author) (2) *Property mappings:* (first_name, foaf:firstName), (last_name, foaf:lastName), and (3) *Entity ID:* AID. `firstName` and `lastName` are properties from the `foaf` ontology (`http://xmlns.com/foaf/spec`) and `Author` is a class from the `nlon` ontology (`http://lod.nl.go.kr/page/ontology`).

Query. The purpose of using a top query language, SPARQL in our case, is mainly to *join* data coming from multiple sources. Therefore, we assume that certain query forms and constructs are of less concern to Data Lake users, e.g., multi-level nested queries, queries with variable properties, `CONSTRUCT` queries.

2.3 Architecture Components

Squerall consists of four main components (cf. Figure 1). Because of Squerall extensible design, also for clarity, we hereafter use the generic ParSets and POA

```
- Input: ParSetJoinsArray // An Array of all join pairs [ParSet,ParSet]
- Output: ResultsParSet // A ParSet joining all ParSets
ResultsParSet = ParSetJoinsArray.head // First join pair
iterate ParSetJoinsArray : current-pair
    if current-pair joinable_with ResultsParSet
        ResultsParSet = ResultsParSet join current-pair
    else add current-pair to PendingJoinsQueue
// Next, iterate through PendingJoinsQueue like ParSetJoinsArray
```

Listing 1.1: ParSet Join.

concepts instead of Squerall's underlying equivalent concrete terms, which differ from engine to engine. The latter are presented in Section 3.

(1) Query Decomposor. This component is commonly found in OBDA and query federation systems (e.g., [12]). It here decomposes the query's Basic Graph Pattern (BGP, conjunctive set of triple patterns in the `where` clause) into a set of star-shaped sub-BGPs, where each sub-BGP contains all the triple patterns sharing the same *subject variable*. We refer to these sub-BGPs as *stars* for brevity; (see Figure 2 left, stars are shown in distinct colored boxes). Query decomposition is subject-based (variable subjects), because the focus of query execution is on bringing and joining *entities* from different sources, not to retrieve a specific known entity. Retrieving a specific entity, i.e., subject is constant, requires full-data parsing and creating an index in a pre-processing phase. This defies the Data Lake definition to access original data without a pre-processing phase. A specific entity can be obtained, nonetheless, by filtering on its attributes.

(2) Relevant Entity Extractor. For every extracted star, this component looks in the *Mappings* for entities that have attributes mappings to *each of* the properties of the star. Such entities are *relevant* to the star.

(3) Data Wrapper. In the classical OBDA, SPARQL query has to be translated to the query language of the relevant data sources. This is in practice hard to achieve in the highly heterogeneous Data Lake settings. Therefore, numerous recent publications (e.g., [4,29]) advocated for the use of an intermediate query language. In our case, the intermediate query language is POA's query language, dictated by its internal data structure. The Data Wrapper generates data in POA's data structure at query-time, which allows for the parallel execution of expensive operations, e.g., join. There must exist *wrappers* to convert data entities from the source to POA's data structure, either fully, or partially if parts of the data can be pushed down to the original source. Each identified star from step (1) will generate exactly one ParSet. If more than an entity are relevant, the ParSet is formed as a *union*. An auxiliary user input *Config* is used to guide the conversion process, e.g., authentication, or deployment specifications.

(4) Distributed Query Processor. Finally, ParSets are joined together forming the final results. ParSets in the POA can undergo any query operation, e.g., selection, aggregation, ordering, etc. However, since our focus is on querying multiple data sources, the emphasis is on the *join* operation. Joins between stars translate into joins between ParSets (Figure 2 phase I). Next, ParSet pairs are all iteratively joined to form the *Results ParSet* (Figure 2 phase II) using Listing 1.1

```
1   <#AuthorMap>
2    rml:logicalSource [
3     rml:source: "../authors.parquet" ; nosql:store nosql:parquet ] ;
4    rr:subjectMap [
5      rr:template "http://exam.pl/../{AID}" ; rr:class nlon:Author ] ;
6    rr:predicateObjectMap [
7      rr:predicate foaf:firstName ; rr:objectMap [rml:reference "Fname"] ] ;
8    rr:predicateObjectMap [ rr:predicate drm:worksFor ; rr:objectMap <#FunctionMap>] .
9   <#FunctionMap>
10   fnml:functionValue [ rml:logicalSource "../authors.parquet" ; # Same as above
11     rr:predicateObjectMap [ rr:predicate fno:executes ;
12      rr:objectMap [rr:constant grel:string_toUppercase] ];
13     rr:predicateObjectMap [
14       rr:predicate grel:inputString ; rr:objectMap [rr:reference "InID"]
15     ] ] . # Transform "InID" attribute using grel:string_toUppercase
```

Listing 1.2: Mapping an entity using RML and FnO.

algorithm. In short, extracted join pairs are initially stored in an array. After the first pair is joined, it iterates through each remaining pair to attempt further joins or, else, add to a queue[1]. Next, the queue is similarly iterated, when a pair is joined, it is unqueued. The algorithm completes when the queue is empty. As the *Results ParSet* is a ParSet, it can also undergo query operations. The join capability of ParSets in the POA replaces the lack of the join common in many NoSQL databases, e.g., Cassandra, MongoDB [20]. Sometimes ParSets cannot be readily joined due to a syntactic mismatch between attribute values. Squerall allows users to declare *Transformations*, which are atomic operations applied to textual or numeral values, details are given in subsection 3.2.

3   Implementation

Squerall[2] is written in Scala. It uses *RML* and *FnO* to declare data mappings and transformations, and *Spark* [35] and *Presto*[3] as query engines.

3.1   Data Mapping

Squerall accepts entity and attribute mappings declared in RML [10], a mapping language extending the W3C R2RML [7] to allow mapping heterogeneous sources. The following fragment is expected (e.g., `#AuthorMap` in Listing 1.2):

- `rml:logicalsource` used to specify the entity source and type.
- `rr:subjectMap` used (only) to extract the entity ID (in brackets).
- `rr:predicateObjectMap`, used for all entity attributes; maps an attribute using `rml:reference` to an ontology term using `rr:predicate`.

---

[1] We used queue data structure simply to be able to dynamically pull (unqueue) elements from it iteratively till it has no more elements.

[2] Available at `https://github.com/EIS-Bonn/Squerall` (Apache-2.0 license).

[3] `http://prestodb.io`

We complement RML with the property `nosql:store` (line 5) from our NoSQL ontology[4], to enable specifying the entity type, e.g., Cassandra, MongoDB, etc.

3.2 Data Transformation

To enable data joinability, Squerall allows users to declare transformations. Two requirements should be met: (1) transformation specification should be decoupled from the technical implementation, (2) transformations should be performed on-the-fly on *query-time*, complying with the Data Lake definition.

We incorporate the Function Ontology (FnO, [8]), which allows to declare machine-processable high-level functions, abstracting from the concrete technology used. We use FnO in conjunction with RML similarly to the approach in [9] applied to the DBpedia Extraction Framework. However, we do not physically generate RDF triples but only apply FnO transformations on-the-fly at query-time. Instead of directly referencing an entity attribute `rml:reference` (e.g., line 7 Listing 1.2), we reference an FnO function that alters the values of the attribute (line 9 Listing 1.2). For example in Listing 1.2, the attribute `InID` (line 18) is indirectly mapped to the ontology term `drm:worksFor` via the `#FunctionMap`. This implies that the attribute values are to be transformed using the function represented by the `#FunctionMap`, `grel:string_toUppercase` (line 16). The latter sets the `InID` attribute values to uppercase.

Squerall visits the mappings at query-time and triggers specific Spark and Presto operations over the query intermediate results whenever a transformation declaration is met. In Spark, a *map()* transformation is used, in Presto corresponding string or numeral SQL operations are used. For the uppercase example, in Spark `upper(DataFrame column)` function inside a *map()* is used, in Presto the SQL `upper()` string function is used.

3.3 Data Wrapping and Querying

We implement Squerall engine using two popular frameworks: Apache Spark and Presto. Spark is a general-purpose processing engine and Presto a distributed SQL query engine for interactive querying, both base their computations primarily in memory. We leverage Spark's and Presto's connector concept, which is a wrapper able to load data from an external source into their internal data structure (ParSet), performing *flattening* of any non-tabular representations. Spark's internal data structure is called *DataFrame*, which is a tabular structure *programmatically* queried in SQL. Their schema corresponds to the schema of the ParSet, *a column per star predicate*. As explained with ParSets, DataFrames are created from the relevant entities, and incrementally joined. Other non-join operations found in SPARQL query (e.g., selection, aggregation, ordering) are translated to equivalent SQL operations; they are applied either at the level of individual DataFrames, or the level of the final results DataFrame, whichever is more optimal. As an optimization, in order to reduce

---

[4] URL: `http://purl.org/db/nosql#`, details are out of the scope of this article.
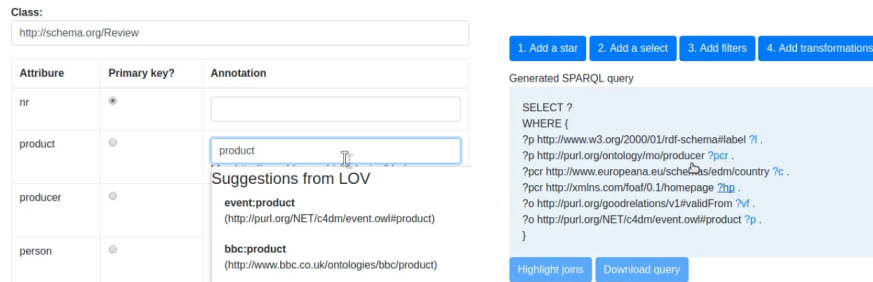
(a) SPARQL UI.   (b) Mapping UI.

Fig. 3: Screenshots from Squerall GUIs.

the intermediate results and, thus, data to join with, we push the selection and transformation to the level of individual DataFrames. We leave aggregation and ordering to the final results DataFrame, as those have results-wide effect. Presto also loads data into its internal native data structure. However, unlike Spark, it does it transparently; it is not possible to manipulate those data structures. Rather, Presto accepts one self-contained SQL query with references to all the relevant data sources, e.g., `SELECT cassandra.cdb.product C JOIN mongo.mdb.producer M ON C.producerID = M.ID`. ParSets in this case are views (SELECT sub-queries), which we create, join and optimize similarly to DataFrames.

Spark and Presto make using connectors very convenient, users only provide values to a pre-defined list of *options*. Spark DataFrames are created as follows: `spark.read.format(sourceType).options(options).load`. In Presto, options are added to a simple file. Leveraging on this simplicity, Squeral supports out-of-the box five data sources: Cassandra, MongoDB, Parquet, CSV and JDBC (MySQL tested). We chose Spark and Presto as they have a good balance between the number of connectors, ease of use and performance [33,17].

3.4   User Interfaces

Squerall is provided with three user interfaces allowing to generate its three needed input files (config, mappings and query), respectively described as follows:

Connect UI shows and receives from users the required options that enable the connection to a data source, e.g., host, port, password, cluster settings, etc.

Mapping UI uses connection options to extract data schema (entity and attributes). It then allows the users to fill or search existing ontology catalogues for equivalent ontology terms (cf. Figure 3a).

SPARQL UI guides non-SPARQL experts to build correct SPARQL queries by means of widget offered for different SPARQL constructs (cf. Figure 3b).

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

Table 1: Data sources and corresponding number of tuples loaded.

4 Performance Analysis

4.1 Setup

Datasets: As there is no Data Lake dedicated benchmark with SPARQL support, we opt for BSBM [3], a benchmark conceived for comparing the performance of RDF triple stores with SPARQL-to-SQL rewriters. We use its data generator to generate SQL dumps. We pick five tables: Product, Producer, Offer, Review, and Person tables, pre-process them to extract tuples and load them into five different data sources (cf. Table 1). Those were chosen to enable up to 4-chain joins of five different data sources. We generate three scales: 500k, 1.5M and 5M (number of products)[5].

Queries: Since we only populate a subset of the generated BSBM tables, we have to alter the initial queries accordingly. We discard joining with tables we do not consider, e.g., Vendors, and replace them with others populated. All queries[6] result in a cross-source join, from 1 (between 2 sources e.g., Q3) to 4 (between 5 sources e.g., Q4). Queries with yet unsupported syntax, e.g., `DESCRIBE`, `CONSTRUCT`, are omitted.

Metrics: We evaluate results *accuracy* and query *performance*. For accuracy, we compare query results against a centralized relational database (MySQL), as it represents data at its highest level of consistency. For performance, we measure query execution time wrt. the three generated scales. A particular emphasis is put on the impact of the *number of joins* on query time. We run each query three times and calculate their mean value. The timeout threshold is set to 3600s.

Environment: We ran our experiments in a cluster of three machines each having DELL PowerEdge R815, 2x AMD Opteron 6376 (16 cores) CPU and 256GB RAM. In order to exclude any effect of caching, all queries are run on a cold cache. Also, no optimization techniques from the query engines were used.

4.2 Results and Discussion

We compare the performance of Squerall's two underlying query engines: Spark and Presto, in absence of a similar work allowing to query all five data sources and SPARQL fragment that Squerall supports.

Accuracy. The number of results returned by Squerall was in all queries of scale 0.5m identical to MySQL, i.e., 100% accuracy. MySQL timed out with data of

---

[5] The 1.5M scale factor generates 500M RDF triples, and the 5M factor 1,75B triples.

[6] See `https://github.com/EIS-Bonn/Squerall/tree/master/evaluation`

(a) Scale 0.5m.
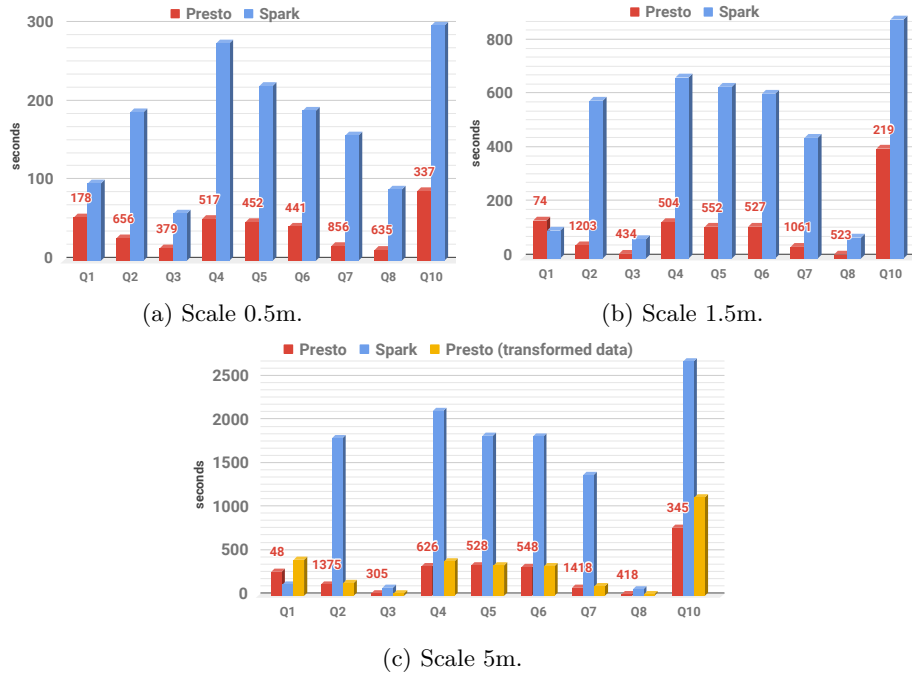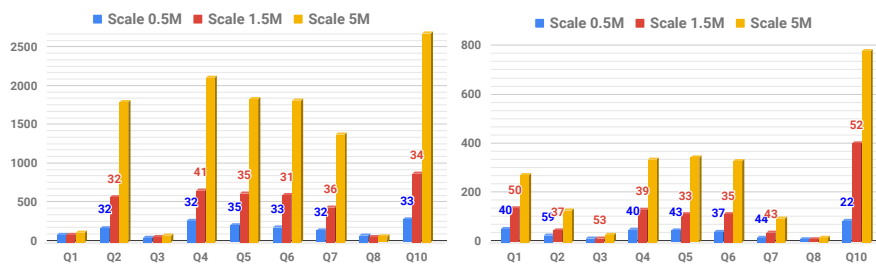
(b) Scale 1.5m.

(c) Scale 5m.

Fig. 4: Query execution Time (seconds). The labels on top of Presto's columns show the percentage between Presto's and Spark's execution times, e.g., in (a) in Q2, 178 means that Presto-based Squerall is 178% faster than Spark-based.

scale 1.5M, starting from which we compared the performance between the two engines, and results were also identical.

Performance. The results (cf. Figure 4) suggest that Squerall overall exhibits reasonable performance throughout the various queries, i.e., different number of joins, with and without filtering and ordering. Presto-based Squerall exhibits significantly better performance than Spark-based, up to an order of magnitude. With the 0.5M data scale, query performance is superior across all the queries with an increase of up to 800%. With scales 1.5M and 5M, Presto-based is superior in all queries other than Q1, with an increase of up to 1300%. This superiority is due to a number of factors. Presto is built following the MPP (Massively Parallel Processing) principles specifically to optimize SQL querying. Another factor is that Presto has far less preparation overhead (e.g., mid-query fault-tolerance, resource negotiation) than Spark. Spark, in the other hand, is a general-purpose system, basing its SQL library on its native in-memory tabular data structure not originally conceived for *ad hoc* querying. Also, it incurs overhead to guarantee query resiliency and manage resources.

Query performance was homogeneous across all the scales and between the two engines, except for Q2, which was among the fastest in Presto contrarily to Spark. This query is special in that it projects out most *Product* attributes joining the largest entity, *Offer*, without any filtering, which may indicate that Presto handles intermediate results of unselective queries better. Q3 was the

(a) Spark-based Squerall query times.   (b) Presto-based Squerall query times.

Fig. 5: Numbers above the bars denote time percentage differences between the scales, e.g., in (a) Q2 execution time in scale 0.5M is 32% of that in scale 1.5M, which is 32% of that in 5M. On average, percentages are ≈30% in all cases (both engines), which is proportional to the data scale (5m=30%1.5m=30%0.5m).

fastest, as it has the lowest number of joins. Followed by Q1 and Q8, which contain more joins, but with significantly filtered number of products. The rest of the queries are notably slower because they join with the large entity *Offer*. The presence of the LIMIT clause did not have a direct effect on query time, it is present in Q1-Q5, Q8 and Q10, across which the performance varies significantly.

Although the current data distribution does not represent the best-case scenario (e.g., query performance would be better if *Review* data was loaded into Cassandra instead), we intentionally stored the large data into the better performing data sources. Our purpose in this experiment series was to observe Squerall behavior using Spark and Presto across the various scales and queries. For example, we observed that, although the largest data entity was loaded into a very efficient database, MongoDB, the queries implicating this entity were the slowest anyway. This gives an indication of the performance of those queries if the same entity was loaded into a less capable source, e.g., Parquet or CSV.

Increasing data size did not diminish query performance; query times were approximately proportional to the data size (cf. Figure 5), and remained under the threshold. In order to evaluate the effect of the query-time data transformations, we intentionally introduce variations to the data so it becomes *unjoinable*. In table Product, we decrease the column *pr* values by 71, in table Producer, we append the string "-A" to all values of column *pr*, and in table *Review* we prefix the values of column *person* with the character "P". We declare the necessary transformations accordingly. The results show that there is a negligible cost in the majority of the cases. This is attributed to the fact that both Spark and Presto base computations in memory. In Spark, those transformations involve only *map* function, which is executed locally very efficiently, not requiring any data movement. Only few queries in 5M in Presto-based Squerall exhibited noticeable but not significant costs, e.g., Q1 and Q10. Due to the insignificant differences and to improve readability, we only add the results of the transformation cost in the scale 5M Figure 4c. Our results could be considered as a performance comparison between Spark and Presto, of which few exist [17,33].

5   Availability, Sustainability, Usability and Extensibility

Squerall is integrated[7] into SANSA [18] since version 0.5, a large framework for distributed querying, inference and analytics over knowledge graphs. SANSA has been used across a range of funded projects, such as BigDataEurope, SLIPO, and BETTER. Via the integration into SANSA, Squerall becomes available to a large user base. It benefits from SANSA's various deployment options, e.g., Maven Central integration and runnable notebooks. SANSA has an active developer community ($\approx$20 developers), an active mailing list, issue tracking system, website and social media channels. Prior to its integration, Squerall features were recurrently requested in SANSA, to allow it to also access large non-RDF data. Squerall sustainability is ensured until 2022 thanks to a number of contributing innovation projects including Simple-ML, BETTER, SLIPO, and MLwin. Further, Squerall is being adopted and extended in an internal use-case of a large industrial company; we plan to report this in a separate adequate submission.

The development of Squerall was driven by technical challenges identified by the H2020 BigDataEurope project[8], whose main technical result, the Big Data Integrator platform, retains a significant amount of interest by the open-source community. In the absence of appropriate technical solutions supporting Data Lake scenarios, the platform development was constrained to transform and centralize most of the data in the use-cases. The need to invest in architectures, tools and methodologies to allow for decentralized Big Data management was highly emphasized by the project. Further, demonstrating the feasibility and effectiveness of OBDA on top of the ever increasing movement of NoSQL has a positive impact on the adoption of Semantic Web principles. This indicates some clear evidence of Squerall's value and role in the community.

Squerall is openly available under *Apache-2.0* terms; it is hosted on GitHub[9] and registered in Zenodo[10] (DOI: *10.5281/zenodo.2636436*). Squerall makes use of open standards and ontologies, including SPARQL, RML, and FnO. It can easily be built and used thanks to its detailed documentation. Its usage is facilitated by the accompanied user interfaces, which are demonstrated in a walkthrough screencast[11]. Further, we provide a Docker image allowing to easily setup Squerall and reproduce the presented evaluation. Squerall was built with extensibility in mind; it can be programmatically extended[12] by (1) adding a new query engine, e.g., Drill, due to its modular code design (cf. Figure 6), and (2) supporting more data sources with minimal effort by leveraging Spark/Presto connectors. A mailing list and a Gitter community are made available for the users.

---

[7] `https://github.com/SANSA-Stack/SANSA-DataLake`

[8] `www.big-data-europe.eu` & `https://github.com/big-data-europe`

[9] `https://github.com/EIS-Bonn/Squerall`

[10] `https://zenodo.org/record/2636436`

[11] `https://github.com/EIS-Bonn/Squerall/tree/master/evaluation/screencasts`

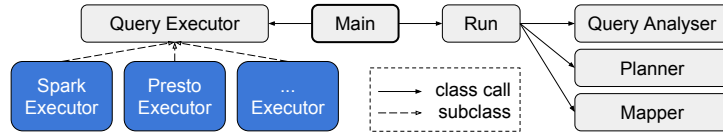[12] `https://github.com/EIS-Bonn/Squerall/wiki/Extending-Squerall`

Fig. 6: Squerall class call hierarchy. Engine classes (colored) are decoupled from the rest (grey). A new engine can be added by extending only Query Executor class (implementing it methods).

6 Related Work

There are several solutions for mapping relational databases to RDF [28], and OBDA over relational databases [34], e.g., Ontop, Morph, Ultrawrap, Mastro, Stardog. Although we share the OBDA concepts, our focus goes to the heterogeneous non-relational and distributed scalable databases. On the non-relational side, there has been a number of efforts, which we can classify into ontology-based and non-ontology-based.

For non-ontology-based access, [6] defines a mapping language to express access links to NoSQL databases. It proposes an intermediate query language to transform SQL to Java methods accessing NoSQL databases. However, query processing is neither elaborated nor evaluated, e.g., cross-database join is not mentioned. [13] suggests that computations performance can be improved if data is shifted on query-time between multiple databases; the suitable database is decided on a case-to-case basis. Although it demonstrates that the overall performance, including the planning and data movement, is higher when using one database, this is not proven to be true with large data. In real large-scale settings, data movement and I/O can dominate the query time. [26] allows to run CRUD operations over NoSQL databases; beyond, the same authors in [27] enable joins as follows. If the join involves entities in the same database, it is performed locally, if not or if the database lacks join capability, data is moved to another capable database. This implies that no intra-source distributed join is possible, and, similarly to [13], moving data can become a bottleneck in large scales. [1] proposes a unifying *programming* model to interface with different NoSQL databases. It allows direct access to individual databases using *get*, *put* and *delete* primitives. Join between databases is not addressed. [16] proposes a SQL-like language containing invocations to the native query interface of relational and NoSQL databases. The learning curve of this query language is higher than other efforts suggesting to query solely using plain (or minimally adapted) SQL, JSONPath or SPARQL. Although its architecture is distributed, it is not explicitly stated whether intra-source join is also distributed. Besides, the code-source is unfortunately not available. A number of efforts, e.g., [29,30,4], aim at bridging the gap between relational and NoSQL databases, but only one database is evaluated. Given the high semantic and structural heterogeneity found across NoSQL databases, a single database cannot be representative of all the family. Among those, [30] adopts JSON as both conceptual and physical data model. This requires physically transforming query's intermediate results, costing the engine transformation price (a limitation also observed in other efforts). More-

over, the prototype is evaluated with only small data on a single machine. [22] presents SQL++, an ambitious general query language that is based on SQL and JSON. It covers a vast portion of the capabilities of query languages found across NoSQL databases. However, the focus is on the query language, and the prototype is only minimally validated using a single database: MongoDB. [31] considers data duplicated in multiple heterogeneous sources, and identifies the best source to send a query to. Thus, joins between sources are not explored. For ontology-based access, Optique [14] is a reference platform with consideration also for dynamic streaming data. Although based on the open-source Ontop, sources of the Big Data instance are not publicly available. Ontario[13] is a very similar (unpublished) work; however, we were not able to run it due to the lack of documentation, and it appears that wrappers are manually created. [5] considers simple query examples, where joins are minimally addressed. Distributed implementation is future work.

In all the surveyed solutions, support for data source variety is limited or faces bottlenecks. Only support for a few data sources (1-3) is observed, and wrappers are manually created or hard-coded. In contrast, Squerall does not reinvent the wheel and makes use of the many wrappers of existing engines. This makes it the solution with the broadest support of the Big Data Variety dimension in terms of data sources. Additionally, Squerall has among the richest query capabilities (see full fragment in[14]), from joining and aggregation to various query modifiers.

7   Conclusion and Future Work

In this article, we presented Squerall —a framework realizing the Semantic Data Lake, i.e., querying heterogeneous and large data sources using Semantic Web techniques. It performs distributed cross-source join operation and allows users to declare transformations that enable joinability on-the-fly at query-time. Squerall is built using state-of-the-art Big Data technologies, Spark and Presto. Relying on the latter's connectors to wrap the data, Squerall relieves users from hand-crafting wrappers —a major bottleneck in supporting data variety throughout the literature. It also makes Squerall easily extensible, e.g., in addition to the five sources evaluated here, Couchbase and Elasticsearch were also tested. There are dozens of connectors already available[15]. Furthermore, due to its modular code design, Squerall can also be programmatically extended to use other query engines. In the future, we plan to support more SPARQL operations, e.g., `OPTIONAL` and `UNION`, and also to exploit the query engines' own optimizations to accelerate query performance. Finally, in such a heterogeneous environment, there is a natural need for retaining provenance at data and query results levels.

---

[13] `https://github.com/SDM-TIB/Ontario`

[14] `https://github.com/EIS-Bonn/Squerall/tree/master/evaluation`

[15] `https://spark-packages.org` `https://prestodb.io/docs/current/connector`

Acknowledgment

References

1. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to non-relational database systems: The SOS platform. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) In CAiSE. vol. 7328, pp. 160–174. Springer (2012)
2. Auer, S., Scerri, S., Versteden, A., Pauwels, E., Charalambidis, A., Konstantopoulos, S., Lehmann, J., Jabeen, H., Ermilov, I., Sejdiu, G., et al.: The BigDataEurope platform–supporting the variety dimension of big data. In: International Conference on Web Engineering. pp. 41–59. Springer (2017)
3. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(2), 1–24 (2009)
4. Botoeva, E., Calvanese, D., Cogrel, B., Corman, J., Xiao, G.: A generalized framework for ontology-based data access. In: International Conference of the Italian Association for Artificial Intelligence. pp. 166–180. Springer (2018)
5. Curé, O., Kerdjoudj, F., Faye, D., Le Duc, C., Lamolle, M.: On the potential integration of an ontology-based data access approach in NoSQL stores. International Journal of Distributed Systems and Technologies (IJDST) **4**(3), 17–30 (2013)
6. Curé, O., Hecht, R., Le Duc, C., Lamolle, M.: Data integration over NoSQL stores using access path based mappings. In: International Conference on Database and Expert Systems Applications. pp. 481–495. Springer (2011)
7. Das, S., Sundara, S., Cyganiak, R.: R2rml: Rdb to rdf mapping language. working group recommendation, w3c, sept. 2012
8. De Meester, B., Dimou, A., Verborgh, R., Mannens, E.: An ontology to semantically declare and describe functions. In: ISWC. pp. 46–49. Springer (2016)
9. De Meester, B., Maroy, W., Dimou, A., Verborgh, R., Mannens, E.: Declarative data transformations for linked data generation: the case of DBpedia. In: European Semantic Web Conference. pp. 33–48. Springer (2017)
10. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A generic language for integrated RDF mappings of heterogeneous data. In: LDOW (2014)
11. Dixon, J.: Pentaho, Hadoop, and Data Lakes (2010), `https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes`, online; accessed 27-January-2019
12. Endris, K.M., Galkin, M., Lytra, I., Mami, M.N., Vidal, M.E., Auer, S.: MULDER: querying the linked data web by bridging RDF molecule templates. In: Int. Conf. on Database and Expert Systems Applications. pp. 3–18. Springer (2017)
13. Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T., Stonebraker, M.: The bigdawg polystore system and architecture. In: High Performance Extreme Computing Conference. pp. 1–6. IEEE (2016)
14. Giese, M., Soylu, A., Vega-Gorgojo, G., Waaler, A., Haase, P., Jiménez-Ruiz, E., Lanti, D., Rezk, M., Xiao, G., Özçep, Ö., et al.: Optique: Zooming in on big data. Computer **48**(3), 60–67 (2015)

15. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 query language. W3C recommendation **21**(10) (2013)
16. Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., Pereira, J.: CloudMdsQL: querying heterogeneous cloud data stores with a common language. Distributed and Parallel Databases **34**(4), 463–503 (2016)
17. Kolychev, A., Zaytsev, K.: Research of the effectiveness of SQL engines working in HDFS. Journal of Theoretical & Applied Information Technology **95**(20) (2017)
18. Lehmann, J., Sejdiu, G., Bühmann, L., Westphal, P., Stadler, C., Ermilov, I., Bin, S., Chakraborty, N., Saleem, M., Ngomo, A.C.N.: Distributed semantic analytics using the SANSA stack. In: ISWC. pp. 147–155. Springer (2017)
19. Mami, M.N., Graux, D., Scerri, S., Jabeen, H., Auer, S.: Querying data lakes using spark and presto. To appear in The WebConf - Demonstrations (2019)
20. Michel, F., Faron-Zucker, C., Montagnat, J.: A mapping-based method to query MongoDB documents with SPARQL. In: International Conference on Database and Expert Systems Applications. pp. 52–67. Springer (2016)
21. Miloslavskaya, N., Tolstoy, A.: Application of big data, fast data, and data lake concepts to information security issues. In: International Conference on Future Internet of Things and Cloud Workshops. pp. 148–153. IEEE (2016)
22. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR, abs/1405.3631 (2014)
23. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Journal on Data Semantics X. Springer (2008)
24. Quix, C., Hai, R., Vatov, I.: GEMMS: A generic and extensible metadata management system for data lakes. In: CAiSE Forum. pp. 129–136 (2016)
25. Saleem, M., Ngomo, A.C.N.: Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation. In: Ext. Semantic Web Conf. Springer (2014)
26. Sellami, R., Bhiri, S., Defude, B.: Supporting multi data stores applications in cloud environments. IEEE Trans. Services Computing **9**(1), 59–71 (2016)
27. Sellami, R., Defude, B.: Complex queries optimization and evaluation over relational and NoSQL data stores in cloud environments. IEEE Trans. Big Data **4**(2), 217–230 (2018)
28. Spanos, D., Stavrou, P., Mitrou, N.: Bringing relational databases into the semantic web: A survey. Semantic Web pp. 1–41 (2010)
29. Unbehauen, J., Martin, M.: Executing SPARQL queries over mapped document stores with SparqlMap-M. In: 12th Int. Conf. on Semantic Systems (2016)
30. Vathy-Fogarassy, Á., Hugyák, T.: Uniform data access platform for SQL and NoSQL database systems. Information Systems **69**, 93–105 (2017)
31. Vogt, M., Stiemer, A., Schuldt, H.: Icarus: Towards a multistore database system. 2017 IEEE International Conference on Big Data (Big Data) pp. 2490–2499 (2017)
32. Walker, C., Alrehamy, H.: Personal data lake with data gravity pull. In: 5th International Conf. on Big Data and Cloud Computing. pp. 160–167. IEEE (2015)
33. Wiewiórka, M.S., Wysakowicz, D.P., Okoniewski, M.J., Gambin, T.: Benchmarking distributed data warehouse solutions for storing genomic variant information. Database **2017** (2017)
34. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: A survey. IJCAI (2018)
35. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. HotCloud **10**(10-10), 95 (2010)