Querying Data Lakes using Spark and Presto

Mohamed Nadjib Mami*
Fraunhofer IAIS & University of Bonn
mami@cs.uni-bonn.de

Damien Graux
Fraunhofer IAIS
damien.graux@iais.fraunhofer.de

Simon Scerri
Fraunhofer IAIS & University of Bonn
scerri@cs.uni-bonn.de

Hajira Jabeen
University of Bonn
jabeen@cs.uni-bonn.de

Sören Auer
TIB and L3S Research Center
auer@l3s.de

ABSTRACT

Squerall is a tool that allows the querying of heterogeneous, large-scale data sources by leveraging state-of-the-art Big Data processing engines: Spark and Presto. Queries are posed on-demand against a Data Lake, i.e., directly on the original data sources without requiring prior data transformation. We showcase Squerall's ability to query five different data sources, including inter alia the popular Cassandra and MongoDB. In particular, we demonstrate how it can jointly query heterogeneous data sources, and how interested developers can easily extend it to support additional data sources. Graphical user interfaces (GUIs) are offered to support users in (1) building intra-source queries, and (2) creating required input files.

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

Heterogeneous Databases, Data Lake, SQL, NoSQL, SPARQL, Query

ACM Reference Format:
Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, and Sören Auer. 2019. Querying Data Lakes using Spark and Presto. In *Proceedings of the 2019 World Wide Web Conference (WWW'19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3308558.3314132

1 INTRODUCTION

During the last four decades, a variety of data storage and management techniques have been developed in both research and industry. Today, we benefit from a multitude of storage solutions, varying in their data model (e.g. tabular, document, graph) or their ability to scale storage and querying. There are dozens of continuously evolving storage and data management solutions; for the NoSQL family,

*This research was partially supported by the European Union's H2020 research and innovation programme BETTER under the Grant Agreement number 776280.

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3314132

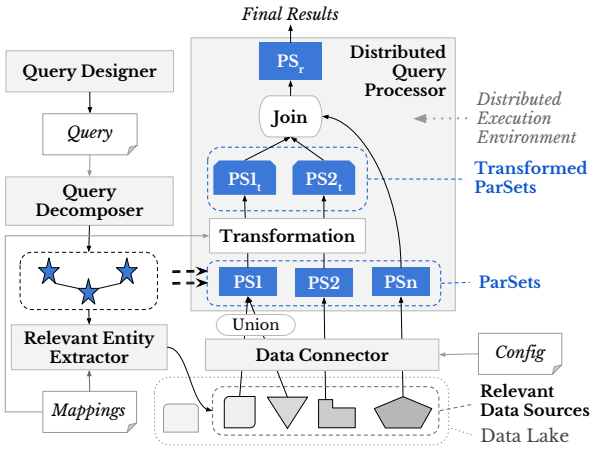

Figure 1: Squerall High-level Architecture.

Cassandra, MongoDB, Couchbase, HBase, Neo4j, DynamoDB, are just a few examples. As a result, users can choose a system that suits their individual application needs.

However, those systems do not inter-operate, every stored datum is locked in the respective system it is stored in. For example, an e-commerce company might store *product* information in a Cassandra database, *offers* in MongoDB to benefit from its capability to store hierarchical multi-level values, and information about *Producers* obtained from an external source in a relational format. Without transforming and moving the data into a unified (scalable) data management solution, the data can hardly be explored and business insights be extracted using ad hoc uniform querying.

We have taken on the mission of bridging this gap and developed Squerall[1], a software that allows to query heterogeneous data directly in its original form and source. We have used standardized and time-proven Semantic Web techniques to enable the uniform querying of heterogeneous data. We support the mapping of terms in the original data to terms in higher-level ontologies, and the querying of the resulting uniform view using SPARQL [13].

Similar efforts to integrate and query large data sources exist in the literature. For instance, [4] defines a mapping language to express access links to NoSQL databases. [12] allows to run CRUD operations over NoSQL databases. [1] proposes a unifying *programming* model to directly access databases using *get*, *put* and *delete* primitives. [8] proposes a SQL-like language containing invocations

[1]Associated website: <https://eis-bonn.github.io/Squerall/>

to the native query interface of relational and NoSQL databases. [7] is a hybrid platform with consideration for both heterogeneous and dynamic data sources (streams). However, Squerall offers the highest number of supported data sources while providing the richest query capability, including joining, aggregation and ordering.

We demonstrate Squerall's ability and efficiency in querying five different data sources (namely: CSV, Parquet, Cassandra, MongoDB and MySQL), and how it can be easily extended to support additional data sources, through several application scenarios.

2 PRINCIPLES AND ARCHITECTURE

In this section we introduce the basic principles and terminology needed to understand the architecture, subsequently described.

- **Mappings:** Squerall implements the so-called Ontology-Based Data Access (OBDA) [11] paradigm. In OBDA, data schemata are mapped to higher-level ontologies, forming a middleware against which SPARQL queries are posed. For example, *Offers(published, closed)* is a MongoDB collection mapped as follows: *(published → bsbm:validFrom, closed → bsbm:validTo, Offers → bsbm:Offer)* with *bsbm:validFrom*, *bsbm:validTo* and *bsbm:Offer* are two properties and a class of the BSBM ontology, respectively. Mappings are a core component of OBDA systems. Therefore, in order for every data source entity (Cassandra tables, MongoDB documents, etc.) to be queried using Squerall, having a mapping is a prerequisite.
- **Distributed Query Execution:** In Squerall, queries are executed in a separate distributed *environment*, which is, in particular, resilient to faults (node failure does not halt the entire query execution), and elastic and horizontally-scalable (more nodes can be added to accommodate more expensive computations). This is required for the following reasons:

- As we are dealing with dispersed data sources, intra-source query execution does not happen locally, but has to be brought into a more resilient, elastic and distributed environment, e.g. HDFS [3], or Spark's RDDs [14]. An exception is when certain sub-queries can be pushed down to the original source, which reduces what is loaded into the query execution environment.
- As we are dealing with large data volumes, query execution cannot be delegated to a single machine, as the intermediate results can overflow both its storage and computational capacity.
- In many modern NoSQL databases, in order to guarantee query performance, certain traditional query operators, e.g., join [10] are dropped. Such missing operations have to be executed in a higher-level capability environment.

- **Enabling joinability:** As data from different sources, e.g., *Product* and *Producer*, is generated by different applications, they may not be able to be readily cross-joined. Thus, modifications on the possible join values ought to be incorporated, e.g., to enable *Product* to join with *Producer*, modify `Product.producer_id` attribute values.

Squerall is comprised of five components (see Figure 1):

(1) **Query Decomposer:** Validates and analyzes SPARQL queries provided by a user. Particularly, the Query Decomposer extracts the Basic Graph Pattern (BGP, i.e., the conjunctive set of triple patterns given in the `where` clause) of the query and decomposes it into star-shaped sub-graph patterns having the same subject, *stars* for short. This component also detects links between stars, e.g., the triple `?product bsbm:producer ?producer` represents a link between the two stars represented by the variables `?product` and `?producer` (e.g., see colored boxes in Figure 1).

(2) **Relevant Entity Extractor:** Each star is analyzed separately; this component searches in the mappings for entities that are mapped to every predicate of the star. For example, given the graph pattern `{ ?x bsbm:validFrom ?y . ?x bsbm:validTo ?z ...}`, the entity *Offers(published, closed)* —a MongoDB collection, and the mappings *(published → bsbm:validFrom, closed → bsbm:validTo)*, this component decides that the entity *Offer* is *relevant* to the star of `?x`. Star type (when present in the query) is taken into account, e.g., `?x a bsbm:Offer`.

(3) **Data Connector:** Once relevant data entities are detected, they are connected to the distributed execution environment. Every detected entity is loaded into what we call a *ParSet* (short of Parallel dataSet), which are data structures that can be distributed and operated on in parallel. The Data Connector expects external users to input the necessary connection metadata, e.g., *user, password, host, port, cluster name*, etc.

(4) **Distributed Query Processor:** Following the principles introduced earlier, queries are executed in parallel. Query execution occurs on and across the ParSets. Links between stars retrieved by the Query Decomposer are transformed into joins between the relevant detected data entities. Figure 2 depicts the process: stars and links between stars (left colored boxes) are mapped to ParSets and joins between ParSets (step I). All stars are incrementally joined forming *Results ParSet* (step II). When disjoinability points are known, join values are altered to enable joinability. We incorporate *transformations* which users need to declare to determine what are the possible changes to make,

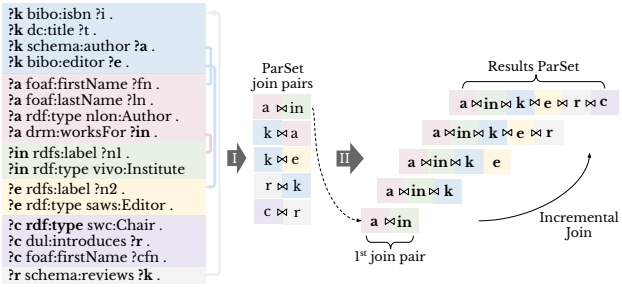

Figure 2: Example of query decomposition, and ParSets join.

Table 1: Query execution times (seconds) using Presto and Spark and the difference percentage between them (%).

	SPARQL SELECT Queries from BSBM (adapted)								
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q10
– Scale 1: scale factor of 0.5 millions products –									
Presto	55.34	28.89	15.84	53.63	49.24	43.38	18.63	14.38	89.08
Spark	98.78	189.57	59.96	277.30	222.76	191.26	159.51	91.38	300.38
Diff. %	**178.48**	**656.17**	**378.57**	**517.06**	**452.40**	**440.88**	**856.31**	**635.33**	**337.21**
– Scale 2: scale factor of 1.5 millions products –									
Presto	139.15	48.59	16.30	133.45	115.35	116.08	42.16	14.37	405.84
Spark	102.86	584.67	70.76	673.12	637.18	611.65	447.27	75.19	888.98
Diff. %	73.92	**1203.36**	**434.05**	**504.41**	**552.40**	**526.93**	**1060.83**	**523.31**	**219.05**
– Scale 3: scale factor of 5 millions products –									
Presto	276.91	131.87	30.58	340.61	350.04	334.29	98.11	18.96	784.01
Spark	132.37	1813.69	93.19	2131.10	1846.59	1833.47	1390.99	79.33	2703.43
Diff. %	47.80	**1375.40**	**304.71**	**625.67**	**527.54**	**548.46**	**1417.80**	**418.47**	**344.82**

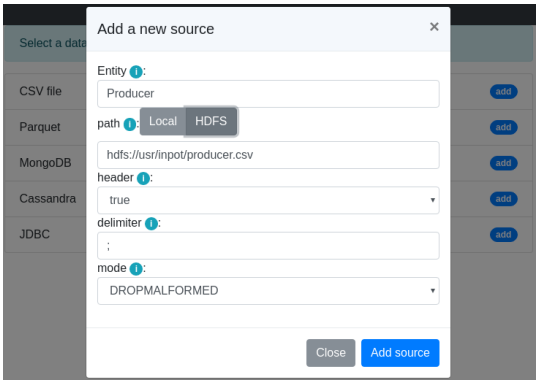

Figure 3: Data Connection GUI.

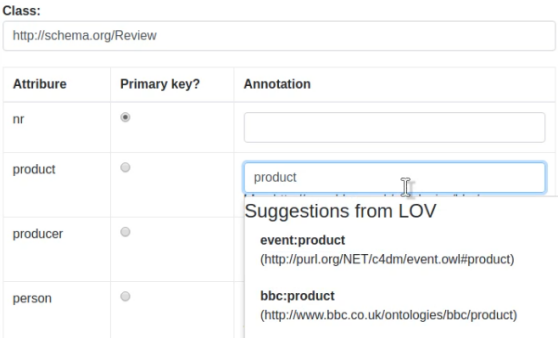

Figure 4: Data Mapping GUI.

e.g., increase the values of the `producer_id` attribute in the *Product* Cassandra table by a constant value.

(5) **Query Designer:** Querying the Data Lake using SPARQL as a uniform query language assumes a minimum level of knowledge about SPARQL. Therefore, we see the necessity of supporting users in their query creation. Users are not expected to provide queries only in plain-text, but can interact with the Query Designer to create correct queries in an intuitive way.

3 ACHIEVED PERFORMANCE

We evaluate the performance of `Squerall` in querying five different data sources: Cassandra, MySQL, MongoDB, Parquet, and CSV. As evaluation data we choose the BSBM [2] benchmark because it allows to (1) generate increasing scales of data (2) generate data in a friendly format, relational, and (3) provide a set of SPARQL queries. We pick five relational tables (*Product, Producer, Offer, Review, and Person*) and load them into the five data sources. To measure accuracy, we load the relational data into a relational database and run equivalent SQL queries. For sizes beyond the relational database capability (query time exceeds 3600s threshold) we self-compare Spark-based and Presto-based `Squerall`. For performance and scalability we evaluate the execution time of BSBM queries against three increasing data sizes: 0.5m, 1,5m and 5m scale factors (number of products). We experiment with all BSBM `SELECT` queries with some adaptation, i.e., queries are modified to involve only the five tables we populated. All the queries have join, from one between 2 data sources to 4 between all the data sources.

The evaluation was carried out in a cluster of three nodes, each having a 16-core DELL PowerEdge processor, 256GB RAM, and 3TB SATA disk. The evaluation results show 100% accuracy across all the three scales. For performance and scalability, `Squerall` exhibits reasonable performance across all three scales (see Table 1). Presto-based `Squerall` exhibits better performance to the Spark-based alternative in the majority of the cases. To measure this difference, we calculate the difference, in percentage, between Presto and Spark execution times (third row under each scale result in Table 1), e.g., for Q2 in the first scale Presto-based `Squerall` is 656% faster.

4 DEMONSTRABLE SCENARIOS

`Squerall` is a realization of the high-level architecture presented in section 2. We here describe its core technologies as well as concrete demonstrable scenarios using it.

1. Core technologies: *RML* [6] along FNO [5] are used to express mappings and to declare query-independent transformations. *Apache Spark* and *Presto* are used to implement both the Data Connector and Distributed Query Processor. Spark is a general-purpose processing engine, and Presto is a distributed SQL query engine for running interactive SQL queries. In Spark, ParSets are represented by *DataFrames*, and the Distributed Execution Environment is the memory pool where DataFrames are stored and transformed. DataFrames are tabular data structures, which are created and queried using *Spark SQL API*. Presto, in the other hand, is used like a database: users directly issue SQL queries without programmatically dealing with its underlying data structure. Thus, ParSets are represented by Presto's internal data structures, which are transparent to the user. Both Spark and Presto provide wrappers, called connectors[2] to connect to a data source. They load the data (fully, or partially if part of the query is pushed down to the data source) to their internal in-memory data structures. Dozens connectors are available relieving developers from creating their own wrappers.

2. User Interfaces: We provide 3 GUIs to help users produce `Squerall`'s inputs: *Config* and *Mappings* files and *SPARQL query*.

(a) **Data Connection** (Figure 3): This interface shows users the query engine-specific options needed to connect to a particular data source, e.g., *user, password, port, host, cluster name,* etc.

(b) **Mapping Creation** (Figure 4): using the connection information from the Data Connection GUI, this interface extracts the data schema and prompts users to map it. It has a built-in search functionality that sends requests to the LOV catalog[3] to search for adequate terms from existing ontologies.

(c) **Query Designing SPARQL** (Figure 5): This interface offers users a number of widgets to interact with. It is able to automatically generate a syntactically correct SPARQL query based on users' input. For example, users are prompted to only enter variable values or to pick from a pre-existing menu list, other

[2] <https://spark-packages.org> and <https://prestodb.io/docs/current/connector.html>
[3] Linked Open Vocabularies is a Web service to publish, search and visualize ontologies <https://lov.linkeddata.es/>.

constant query constructs are pre-entered.

3. Scenarios: Users can explore Squerall through a six-scenario story. They start by using Squerall's 3 GUIs to build the inputs Squerall needs for query execution. Afterwards, they run queries and observe the execution steps and results. Finally, they uncover Squerall's code and learn how it can be extended to support more data sources and query engines. Those scenarios use similar data as the ones used in the performance evaluation section 3.

(1) **Providing connection metadata.** Guided by the dedicated GUI, Figure 3, users provide necessary information to enable Spark and Presto to know and connect to a data source. For example, to load a CSV file, users specify the *delimiter* and select the *strategy* to adopt when the parser encounters a mal-formed line, e.g., dropping the line; or to connect to a MongoDB cluster, users specify the replica-set name.
(2) **Creating mappings.** Users leverage the dedicated interface, Figure 4, to create mappings for the previously connected data. They visualize the data schema and map it to ontology terms (properties and classes). They can try to find terms in existing ontologies using the LOV catalog-powered search; in absence of suitable results, they can input their own terms.
(3) **Building SPARQL queries.** Users explore and use the capabilities of the *Query Designing* GUI, Figure 5. They use the different widgets and incrementally augment their SPARQL query. They start building the query by forming stars (triple patterns of the same subject) and linking them together, then include other query operations like filtering. Users can experiment with the full spectrum of SPARQL fragment supported, which includes *filters, aggregate functions, solution modifiers, and regex*.
(4) **Data Transformation Declaration.** Users are invited to experiment with the two offered options of declaring transformations: (1) *as part of SPARQL query:* they add a new clause at the very end of a SPARQL query; (2) *as part of the mappings:* they amend RML mappings to, instead of mapping an entity attribute, e.g., column `producer_id`, directly to an ontology term, they first modify its values using declared FNO functions. The former is used when transformations vary from a query to query, the latter is used when transformations are fixed.
(5) **Executing SPARQL queries.** Users experiment with querying the data using Squerall. They can either evaluate BSBM queries or their own queries. They explore the query execution steps by visualizing the logs, from query parsing to stars and joins detection, to mapping lookup, and then to query execution.
(6) **Extending Squerall with more data sources and new query engines.** We show how developers can conveniently support more data sources in both Spark and Presto implementations. We also show them how they can programmatically incorporate a new query engine implementation, alongside Spark and Presto, thanks to Squerall's modular code design.

A screencast walking through the various interfaces and the query execution is available[4].

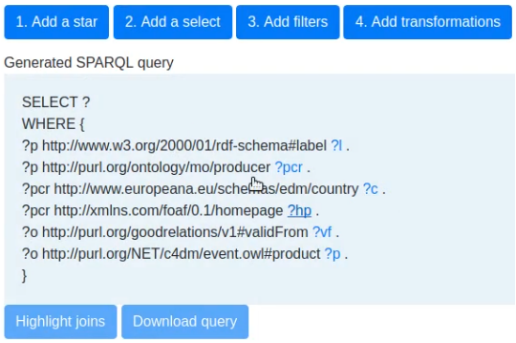

Figure 5: Query Designing GUI.

5 CONCLUSION

Squerall addresses the *Variety* challenge of Big Data, which remains poorly addressed, by making use of Semantic Web standards and best practices. Squerall can conveniently be (pragmatically) extended to embrace new data sources, by making use of the query engines' own wrappers. This approach solves one of the most tedious tasks acknowledged across the literature, i.e., handcrafting wrappers. For example, in addition to the five sources evaluated here, other sources like Couchbase or Elasticsearch can also be easily supported. As a result, Squerall is both innovative and unique in its capability to support a high number of data source types. Additionally, Squerall has been integrated[5] into SANSA [9], a framework for scalable processing and analysis of large-scale RDF data, widening its scope to also access non-RDF data sources. As future work, we plan to expand the supported SPARQL fragment to include OPTIONAL and UNION. Squerall source code is publicly available under an Apache-2.0 license on GitHub[6], accompanied by detailed documentation on installation and usage. This is further facilitated with a Dockerfile to quickly run the BSBM use-case described here (including the input files and adapted queries).

REFERENCES

[1] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. 2012. Uniform access to non-relational database systems: The SOS platform. In *International Conference on Advanced Information Systems Engineering*. Springer, 160–174.
[2] Christian Bizer and Andreas Schultz. 2009. The berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems* 5, 2 (2009), 1–24.
[3] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53 (2008), 1–13.
[4] 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.
[5] Ben De Meester, Wouter Maroy, Anastasia Dimou, Ruben Verborgh, and Erik Mannens. 2017. Declarative data transformations for Linked Data generation: the case of DBpedia. In *European Semantic Web Conference*. Springer, 33–48.
[6] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data.. In *LDOW*.
[7] 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.
[8] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jimenez-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.

[4] <https://github.com/EIS-Bonn/Squerall/blob/master/evaluation/screencasts>
[5] <https://github.com/SANSA-Stack/SANSA-DataLake>
[6] <https://github.com/EIS-Bonn/Squerall>

[9] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Nilesh Chakraborty, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Hajira Jabeen. 2017. Distributed Semantic Analytics using the SANSA Stack. In *ISWC Resources Track*.

[10] 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.

[11] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. 2008. Linking data to ontologies. In *Journal on Data Semantics X*. Springer, 133–173.

[12] Rami Sellami and Bruno Defude. 2018. Complex queries optimization and evaluation over relational and NoSQL data stores in cloud environments. *IEEE Transactions on Big Data* 4, 2 (2018), 217–230.

[13] W3C SPARQL Working Group et al. 2013. SPARQL 1.1 Overview. http://www.w3.org/TR/sparql11-overview/.

[14] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.