How to feed the Squerall with RDF and other data nuts?

Mohamed Nadjib Mami[1,2], 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 & L3S Research Center, Hannover University, Germany
{mami,scerri,jabeen,jens.lehmann}@cs.uni-bonn.de
damien.graux@iais.fraunhofer.de, auer@l3s.de

Abstract. Advances in Data Management methods have resulted in a wide array of storage solutions having varying query capabilities and supporting different data formats. Traditionally, heterogeneous data was transformed off-line into a unique format and migrated to a unique data management system, before being uniformly queried. However, with the increasing amount of heterogeneous data sources, many of which are dynamic, modern applications prefer accessing directly the original fresh data. Addressing this requirement, we designed and developed Squerall, a software framework that enables the querying of original large and heterogeneous data on-the-fly without prior data transformation. Squerall is built from the ground up with extensibility in consideration, e.g., supporting more data sources. Here, we explain Squerall's extensibility aspect and demonstrate step-by-step how to add support for RDF data, a new extension to the previously supported range of data sources.

1 Introduction

The term Data Lake [1] denotes a repository of schema-less data stored in its original form and format without prior transformations. We have built Squerall [2], a software framework implementing the so-called *Semantic Data Lake* concept, which enables querying Data Lakes in a uniform manner using Semantic Web techniques. In essence, Semantic Data Lake incorporates a 'virtual' schema over the schema-less data repository by mapping data schemata into high-level ontologies, which then can be queried in a uniform manner using SPARQL.

The value of a Data Lake-accessing system lays in its ability to query as much data as possible. For this sake, Squerall was built from the ground up with extensibility in consideration, so to allow and facilitate supporting more data sources. As we recognize the burden of creating a wrapper for every needed data source, we resort to leveraging the wrappers that data source providers themselves offer for many state-of-the-art processing engines. For example, Squerall uses Apache Spark and Presto as underlying query engines, both of which benefit from a wide range of connectors accessing the most popular data sources.

In this demonstration, we complement the published[5] work about Squerall [3] by (1) providing more details on the data source extensibility aspect, and (2) demonstrating extensibility by supporting a new data source, RDF.

2 Squerall and its Extensibility

2.1 Squerall: a Semantic Data Lake

Squerall is an implementation of the Semantic Data Lake concept, i.e., querying original large and heterogeneous data using established Semantic Web techniques and technologies. It is built following the Ontology-Based Data Access principles [5], where elements from the data schema (entities/attributes) are associated to elements from an ontology (classes/properties), by means of mapping language, forming a virtual schema against which SPARQL queries can be posed.

2.2 Squerall Extensibility

As we recognize the burden of creating wrappers for the variety of data sources, we chose not to reinvent the wheel and rely on the wrappers often offered by the developers of the data sources themselves or by specialized experts. The way a connector is used is dependent on the query engine:

- **Spark:** the connector's role is to load a specific data entity into a DataFrame using *Spark SQL API*. Its usage is simple, it only requires providing access values to a predefined list of *options* inside a simple connection template:

```
spark.read.format(sourceType).options(options).load
```

  Where `sourceType` designates the data source type to access, and `options` is a simple key-value list storing e.g., username, password, host, cluster settings, etc. The template is similar in most data source types. There are dozens connectors[6] already available for a multitude of data sources.
- **Presto:** access options are stored in a plain text file in a key-value fashion. Presto uses directly SQL interface to query heterogeneous data, e.g., `SELECT cassandra.cdb.product C JOIN mongo.mdb.producer M ON C.producerID = M.ID`, there is no direct interaction with the connectors. Presto internally and transparently uses the access options to load necessary data on query-time. Similarly, there are already several ready-to-use connectors for Presto[7].

Hence, while Squerall supports by default MongoDB, Cassandra, Parquet, CSV and various JDBC sources, interested users can easily provide access to other data sources leveraging Spark and Presto connectors[8].

[5] At ISWC-Resources track.
[6] https://spark-packages.org/
[7] https://prestosql.io/docs/current/connector.html
[8] Tutorial: https://github.com/EIS-Bonn/Squerall/wiki/Extending-Squerall

3 Supporting a New Data Source: Case of RDF Data

In case no connector is found for a given data source type, we show in this section the principles of supporting a new data source. The procedure concerns Spark as query engine, where the connector's role is to generate a DataFrame from an underlying data entity. Squerall did not previously have a wrapper for RDF data. With the wealth of RDF data available today as part of the Linked Data and Knowledge Graph movements, supporting RDF data is paramount. Contrary to the previously supported data sources, RDF does not require a schema, neither fixed nor flexible. As a result, lots of RDF data is generated without schema. In this case, it is required to exhaustively extract the schema from the data on-the-fly during query execution. Also, as per the Data Lake requirements, it is necessary not to apply any pre-processing, and to directly access the original data. If an entity inside an RDF data is detected as relevant to (part of) a query, a set of transformations are applied to flatten the (*subject,property,object*) triples and extract the schema elements needed to generate the DataFrame(s). Full procedure is shown in Figure 1 and is described as follows:

1. First, triples are loaded into Spark distributed dataset[9] of the schema *(subject : String, property : String, object : String)*.
2. Using Spark *transformations*, we generate a new dataset. We map (s,p,o) triples to pairs: (s,(p,o)), then group pairs by subject: (s,(p,o)+), then find class from p (p=`rdf:type`) and map the pairs to new pairs: (class,(s,(p,o)+])), then group them by class (class, (s, (p, o)+)+). Each class has one or more instances identified by 's' and contains one or more (p, o) pairs.
3. The new dataset is *partitioned* into a set of class-based DataFrames, columns of which are the properties and tuples are the objects. This corresponds to the so-called property table partitioning [6].
4. The XSD data types, if present as part of the object, are detected and used to type the DataFrame attributes, otherwise string is used.
5. Only the relevant entity/ies (matching their attributes against query properties) detected using the mappings is/are retained, the rest are discarded.

This procedure generates a (typed) DataFrame that can join DataFrames generated using other data connectors from other data sources. The procedure is part of our previously published effort: SeBiDa [4]. We made the usage of the new RDF connector as simple as the other Spark connectors:

```
val rdf = new NTtoDF()
df = rdf.options(options).read(filePath,sparkURI).toDF
```

Where `NTtoDF` is the connector's instance, `options` are the access information including RDF file path and the specific RDF class to load into the DataFrame.

[9] Called RDD: Resilient Distributed Dataset, a distributed tabular data structure.

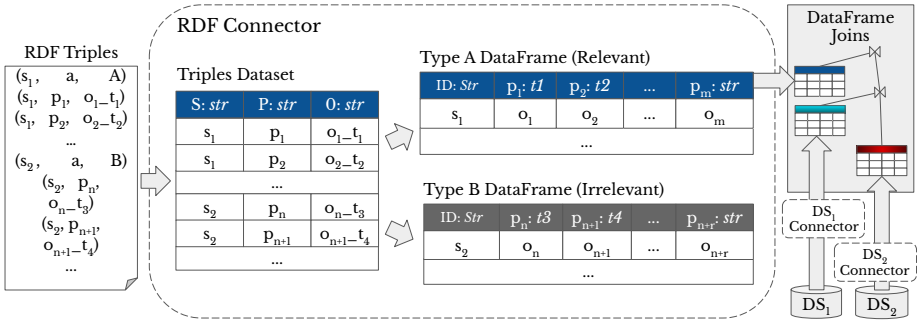

Fig. 1. RDF Connector. A and B are RDF classes, t_n denote data types.

4 Conclusion

In this demonstration article[10], we have described with more depth the extensibility aspect of Squerall in supporting more data sources. We have demonstrated extensibility principles by adding a support for RDF data. In the common absence of schema, RDF triples have to be exhaustively parsed and reformatted into a tabular representation on query-time, which only then can be queried. In the future, in order to alleviate the reformatting cost and, thus, accelerate query processing time, we intend to implement a light-weight caching technique, which can save the results of the flattening phase across different queries. Beyond Squerall context, we will investigate making the newly created connector (currently supporting NTriples RDF) available in Spark Packages (connectors) hub for the public to be able to process large RDF data using Apache Spark.

References

1. 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
2. Mami, M.N., Graux, D., Scerri, S., Jabeen, H., Auer, S.: Querying data lakes using spark and presto. In: The World Wide Web Conference. pp. 3574–3578. WWW '19, ACM, New York, NY, USA (2019)
3. Mami, M.N., Graux, D., Scerri, S., Jabeen, H., Auer, S., Lehman, J.: Squerall: Virtual ontology-based access to heterogeneous and large data sources. Proceedings of 18th International Semantic Web Conference (2019)
4. Mami, M.N., Scerri, S., Auer, S., Vidal, M.E.: Towards semantification of big data technology. In: International Conference on Big Data Analytics and Knowledge Discovery. pp. 376–390. Springer (2016)
5. 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)
6. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient rdf storage and retrieval in jena2. In: Proceedings of the First International Conference on Semantic Web and Databases. pp. 120–139. Citeseer (2003)

[10] Screencasts are publicly available from: https://git.io/fjyOO