An Approach for Representing and Storing RDF Data in Multi-model Databases

Simen Dyve Samuelsen, Nikolay Nikolov, Ahmet Soylu, and Dumitru Roman

SINTEF AS, Oslo, Norway
{simendyve.samuelsen,nikolay.nikolov,ahmet.soylu,dumitru.roman}@sintef.no

Abstract. The emergence of NoSQL multi-model databases, natively supporting scalable and unified storage and querying of various data models, presents new opportunities for storing and managing RDF data. In this paper, we propose an approach to store RDF data in multi-model databases. We identify various aspects of representing the RDF data structure into a multi-model data structure and discuss their advantages and disadvantages. Furthermore, we implement and evaluate the proposed approach in a prototype using ArangoDB—a popular multi-model database.

Keywords: RDF · Multi-modal databases · Storage

1 Introduction

The adoption of the linked data paradigm and the RDF format has grown significantly over the past decade. Even though RDF is getting a wider acceptance, there are two major challenges: systems’ scalability and generality [9]. Working with RDF graphs, which are typically highly connected and distributed, results in matching and querying large volumes of data, thus making the issue with scalability more pressing. In this respect, NoSQL databases can handle larger volumes of data without restricting value types and data structures; however, most NoSQL databases support a single data model – either document, key-value storage or graph. Therefore, they either cannot handle relations between data very well (in the case of key-value and document stores), or they don’t perform as good when it comes to querying large amounts of homogeneous data stored on a node (in the case of graph stores).

Over the past recent years, new types of NoSQL solutions have emerged (referred to as multi-model databases) that attempt to combine the benefits of multiple storage methods from traditional NoSQL databases [4] and could offer a better alternative to earlier approaches aiming to store RDF on relational databases (e.g., [3,5]). In this paper, we describe a practical approach to model, represent and store RDF graph data in a multi-model database. This approach takes advantage of the flexible data modelling offered by graph-model-based databases and the schema-less design of NoSQL. The contributions of this paper
include: (i) an approach for mapping RDF data to a multi-model database representation; (ii) an implementation of the proposed approach using ArangoDB; and (iii) an evaluation of query performance for different types of NoSQL, multi-model, and RDF stores.

The rest of this paper is organised as follows. Section 2 introduces multi-model databases and discusses their benefits. Section 3 describes three different ways of defining the mapping between the RDF and multi-model database storage formats. Section 4 describes a prototype implementation of the proposed approach, while Sect. 5 provides an evaluation of the implementation. Section 6 summarises our contributions and provides suggested directions for future work.

2 Multi-model Databases

Multi-model databases (e.g., ArangoDB, OrientDB, and Redis) are not a new concept and have existed in different forms for a long time [6]. Initially, multi-model databases served as systems to process complex data models. With the emergence of NoSQL databases, the term “multi-model database” has been expanded to support multiple connected storage models – typically document, key-value, and graph. Thereby, such databases are able to offer the benefits of different data models simultaneously, such as scalability and query performance of document and key-value databases and the flexibility and extensibility of graph databases.

The main difference between triple stores and multi-model databases (and even traditional graph databases) is in how they model graphs. In RDF, the nodes of the graph tend to store fine-grained attributes of entities, i.e., attribute values. In the multi-model database context, the units of disclosure are richer entity objects (i.e., they include object attributes) and cross-entity relationships. Multi-model database implementation of the graph model allows for flexible modelling of data within one domain or namespace, rather than multiple domains as in RDF. Furthermore, the fact that entity collections in multi-model databases can be treated as simple key-value or document stores allows performing queries over large volumes of data without the performance penalty of graph traversal/matching. Thus, this class of databases trades off multi-domain standardisation and universal usage with better support within a single domain (scalability, flexibility of data model). An appropriate approach to store self-describing, semantically enriched multi-domain RDF data in a multi-model database can be used to overcome the shortcomings of both representations.

3 Representing RDF Data in ArangoDB

We chose ArangoDB for our study, since it is currently among the best performers for multi-model storage solutions\(^1\) and offers ease of use and non-functional support for different deployments. ArangoDB organizes data using two types of

\(^1\)https://www.arangodb.com/performance.
entities - documents (i.e., nodes) and edges, which are saved in separate stores, referred to as collections. We identified three mapping strategies: (i) direct representation with respect to the RDF data model – each node in the RDF mapping corresponds to a node in the document collection; (ii) direct representation storing the predicate data in edge documents, connecting the subject and object; and (iii) RDF flattening – using a set of heuristics for mapping RDF nodes and literals to the multi-model structure.

3.1 Direct Representation

This approach uses a direct representation that maps each node in an RDF triple to a node in the multi-model database connected through entries in the edge collections. As any RDF triple is built up by a subject, predicate and object, the direct representation would contain one node for each of these values and two edges connecting these nodes. Within each node, we define an attribute “rdf” to store the fully qualified name (URI), which allows to retain the semantics of the RDF node. This approach offers the most expressive representation in terms of querying and matching data within the generated graph. However, multi-model databases (and also graph databases) are not optimised to work with very large numbers of small objects (containing one attribute/value apart from the key), which makes this approach the least suitable for storage out of the three.

3.2 Direct Representation with Edge Values

The direct approach with edge values is similar to the direct approach, but instead of mapping each value of the RDF triple to one node each, the predicate value is stored directly on the edge connecting nodes. The result is then two nodes (the subject and object) connected with one edge (containing the predicate value), the expressiveness of the direct approach is kept while gaining a reduction in data size. This second approach handles larger datasets comparatively better than the direct mapping, but is still rather verbose.

3.3 RDF Flattening

When using normal graph databases, the challenge is to balance between having too large or too small objects when representing data. Using small objects results in extremely large graphs and thus a large number of traversals when querying data. Using too large objects increases query time when matching values, because of the need to go through all values within each node. Document databases, on the other hand, do handle large objects very well and store entries in attribute-value pairs allowing for high-performance querying. This approach to storing and handling RDF data within a multi-model database is, therefore, based on taking advantage of the data model of multi-model databases to store properties as object attributes within a document. This approach allows to store RDF in the most natural way with respect to graph/multi-model databases.

https://www.arangodb.com/docs/stable/data-modeling-concepts.html.
4 Implementation

The implementation of RDF flattening was done using ArangoDB through the following rules:

- URI nodes are mapped JSON objects, which serve as nodes in the representation.
- URIs, which in RDF uniquely identify nodes, are used to generate unique numeric keys for the JSON object (numeric keys enable more efficient storage and lookup). This is done using a standard hash function and the keys themselves are stored in a special attribute called _key.
- Edges between nodes are generated based on the predicates between URI nodes in the RDF mapping template. An exception to this rule applies to rdf:type mappings – in RDF, these are used to specify type mappings for RDF entities. Types in RDF are URI nodes, which point to the semantic classes in an ontology or vocabulary, similarly to classes in object-oriented programming. The classes specified are instead stored in a ‘type’ JSON attribute, which contains an array of all entity types.
- RDF literals are mapped to JSON attributes for the URI node objects. An exception to this rule applies to rdfs:label – in RDF, these mappings are used to denote textual labels of entities. In the multi-model mapping, these values are stored in a ‘label’ attribute.
- Prefixes and fully qualified RDF URIs are also stored in the resulting JSON object. The specified prefixes in the mapping are additionally kept in separate JSON objects in the document collection to avoid overlaps with other prefixes and for enabling namespace-based lookups (based on the RDF namespaces defined in the mapping).

The implementation of RDF flattening was done using the Grafterizer data transformation tool [8] (available in the DataGraft platform [7]) and published on GitHub 3. Two experimental instances of the ArangoDB database were deployed. They have two different configurations. The first configuration uses a three-node in-memory cluster. It was used for initial experimentation for lower-volume data, which was sharded over the three different instances. The second configuration uses a single-node deployment with the persistent storage engine of ArangoDB. The latter configuration has lower memory requirements (as it stores data persistently rather than in memory), which is more appropriate when dealing with Big Data. Both instances of the database were deployed using the Docker-based deployment option of ArangoDB.

5 Evaluation

To make a comparison of how the flattened RDF representation proposed in this paper performs compared to a triple store, we generated RDF data from the

3 https://github.com/samuelsen/RDF---NoSQL-test.
data dump [2] used in the NoSQL benchmark test from ArangoDB [1]. We also deployed instances of Neo4j\(^4\) and OrientDB\(^5\) (with default configuration) as was done in the benchmark. Each query from the test was re-written for both RDF and the flattened representation in ArangoDB. The RDF values were uploaded to a Jena Fuseki SPARQL server\(^6\), which was deployed on the test server where we validated the benchmark results. The test server uses the following configuration: Ubuntu 17.10 (4.13.13), 16x3 GHz AMD Ryzen 7 1700 Eight-Core Processor, 62.9GiB of memory and a SSD of 457GiB. Each test was ran five times averaging the results of these five runs. The tests performed were as follows:

- **Shortest-path** – For 1000 sets of IDs, the time taken to find the shortest path between all sets of IDs;
- **Neighbors** – For 1000 IDs find all neighbors of these IDs;
- **Neighbors 2** – For 1000 IDs find all neighbors of these and all neighbors of the retrieved neighbors;
- **Single read** – Average read time for (100.000 reads);
- **Aggregate** – For all entries count the number of occurrences of different edges.

| Database          | Shortest Path | Neighbors | Neighbors 2 | Single read | Aggregate |
|-------------------|---------------|-----------|-------------|-------------|-----------|
| ArangoDB (file system) | 1.28          | 1.04      | 4.71        | 3.84        | 0.15      |
| ArangoDB (In-memory)    | 2.49          | 0.96      | 4.75        | 9.08        | 0.19      |
| OrientDB            | 125.32        | 1.24      | 29.03       | 9.31        | 0.56      |
| Neo4j               | 1.47          | 0.28      | 3.76        | 1.81        | 0.25      |
| Jena Fuseki         | n/a           | 11.73     | 9.81        | d.n.f       | 1.60      |

* all results given in milliseconds

**Fig. 1.** Benchmarking results (with default configurations of the databases)

As can be seen in the Fig. 1, Neo4j performs best in the graph traversal benchmark due to its being specifically designed to support the graph model. However, Neo4j does not support document-key-value storage and respective querying capabilities, and was thus not our chosen solution, but is given as baseline. In terms of multi-model storage, we tested ArangoDB with the in-memory and file-system based storage, as well as OrientDB. Out of the tested multi-model databases, ArangoDB outperforms OrientDB in all benchmarks and comes closest to Neo4j in terms of graph traversal capabilities in multi-model stores. With respect to triple store, the tested solution – Jena Fuseki – has much worse performance. We were not able to perform the shortest path experiment due to a memory exception. Furthermore, Fuseki did not finish the test for single reads, which attempted to perform 100 000 reads to get the average response time.

\(^4\) https://neo4j.com.

\(^5\) https://www.orientdb.org.

\(^6\) https://jena.apache.org/documentation/fuseki2.
This is due to the lack of scalability of the SPARQL endpoint API in comparison to the proprietary APIs of the other storage solutions, which can support multiple simultaneous connections and larger load. In the other tests, Fuseki was up to an order of magnitude slower than ArangoDB. A notable limitation of this experiment is that all databases were configured using the default configuration.

6 Conclusions

In this paper, we presented an approach to model and store RDF data in a multi-model database. Using our proposed RDF flattening approach, it is possible to retrieve results when looking for specific values without needing graph traversals, which significantly improves query performance. Additionally, due to the use of the document structure to store predicate and literal values directly within an entity object (representing the RDF node), the data size is reduced. A possible direction for future work is to expand the comparison by using different benchmarks, especially ones that are traditionally used to evaluate RDF stores. Furthermore, the evaluation can be extended to implement a more extensive exploration of the currently available triple stores by including a larger number of them. Another possible direction for future work would be to set up procedures for producing RDF triples out of the graph database representation presented in this work.

Acknowledgements. The work in this paper was partly funded by the EC H2020 projects euBusinessGraph (732003), EW-Shopp (732590), and TheyBuyForYou (780247).

References

1. ArangoDB NoSQL Performance Benchmark. https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb
2. Pokec social network - data set dump. https://snap.stanford.edu/data/soc-pokec.html
3. Bornea, M., et al.: Building an efficient RDF store over a relational database. Proc. SIGMOD 2013, 121–132 (2013)
4. Lu, J., Holubová, I.: Multi-model databases: a new journey to handle the variety of data. ACM Comput. Surv. 52(3), 1–38 (2019)
5. Pan, Z., Heflin, J.: DLDB: Extending relational databases to support semantic web queries. In: Proceedings of PSSS1 2003, pp. 109–113 (2003)
6. Plucznik, E., Zgorzalek, K.: The multi-model databases - a review. Proc. BDAS 2017, 141–152 (2017)
7. Roman, D., et al.: Datagraft: one-stop-shop for open data management. Semant. Web 9(4), 393–411 (2018)
8. Sukhobok, D., et al.: Tabular data cleaning and linked data generation with Grafterizer. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 134–139. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47602-5_27
9. Zeng, K., et al.: A distributed graph engine for web scale rdf data. Proceedings of the VLDB Endowment 6(4), 265–276 (2013)