Querying in a Workload-Aware Triplestore Based on NoSQL Databases

Luiz Henrique Zambom Santana(✉) and Ronaldo dos Santos Mello

Universidade Federal de Santa Catarina, Florianópolis, Brazil
luiz.santana@posgrad.ufsc.br, r.mello@ufsc.br

Abstract. RDF and SPARQL are increasingly used in a broad range of information management scenarios (*e.g.*, governments, corporations, and startups). Scalable SPARQL querying has been the main issue for virtually all the recent RDF triplestores. This paper presents *WA-RDF*, a middleware that addresses workload-adaptive management of large RDF graphs. Our middleware not only employs all the most used NoSQL data models but also provides a novel RDF data partitioning approach based on a fragmentation strategy that maps RDF data into multiple NoSQL databases. This workload-aware partitioning scheme provides, in turn, efficient processing of SPARQL queries over these NoSQL databases. Our experimental evaluation shows that the solution is promising, outperforming three recent baselines.

Keywords: RDF · SPARQL · NoSQL · Workload · Triplestore

1 Introduction

In the last decade, RDF, the standardized data model that, along with other technologies, like RDFS, and SPARQL, grounds the vision of the Semantic Web, was affected by a wide range of data management problems. The main reason for that is the current scale of Big Data intensive applications, which generates very large datasets and need to efficiently store massive RDF graphs that goes beyond the processing capacities of existing RDF storage systems operating on a single node. This scenario includes innovations in the frontier of Semantic Web research fields. For example, semantic technologies can enhance the storage of moving object trajectories [6], generating huge datasets about traffic, people behaviour and citizen routine. The scale of this kind of domain raises the need for new triplestores that can, for instance, take advantage of NoSQL databases to store and access large volumes of RDF data.

This paper presents *WA-RDF*, a triplestore composed of a middleware and multiple NoSQL databases. Our middleware includes a novel RDF data partitioning approach with a fragmentation strategy that maps pieces of an RDF graph into NoSQL databases with different data models. We consider a workload-aware partitioning approach based on the ideas from *Estocada* [1] to develop a

© Springer Nature Switzerland AG 2019
S. Hartmann et al. (Eds.): DEXA 2019, LNCS 11707, pp. 159–173, 2019.
https://doi.org/10.1007/978-3-030-27618-8_12

multiformat RDF storage that takes into account the query workload to decide which NoSQL data model is the best fit for each incoming RDF fragment.

The main contributions of this paper are: *(i)* a workload-aware RDF data partitioning approach based on the current graph structure and, mainly, on the typical application queries; *(ii)* a query processing mechanism that takes advantage of the partitioning approach to define efficient query planning to access RDF data; *(iii)* a set of experiments that evaluate our solution against three baselines (*Rainbow* [2], *ScalaRDF* [4] and S2RDF [8]) by considering the NoSQL databases MongoDB and Neo4J. Our strong point is the ability to process queries over large RDF graphs stored on multiple NoSQL database servers with a subtle amount data joining cost. The experimental evaluation shows that our middleware scales well.

The rest of the paper is organized as follows. Section 2 contains the background and related work. Sections 3 and 4 detail the *WA-RDF* approach. Section 5 reports the experimental evaluation and Sect. 6 concludes the paper.

2 Background and Related Work

The most important pillars of this work are the Semantic Web and the NoSQL databases movement.

Currently, the Semantic Web is defined mainly in terms of well-established standards for expressing shared meaning, defined by WWW Consortium (W3C)[1], like *Resource Description Framework (RDF)* and *the Simple Protocol and RDF Query Language (SPARQL)*. RDF is expressed by triples that define a relationship between two resources. RDF triples can be modeled as graphs, where the resources, called *subject* and *object*, are vertexes, and the relationship, called *predicate*, is a directed edge from the subject to the object. SPARQL is a query language for searching and retrieving RDF information. The most important part of a SPARQL query is the *triple pattern*, which defines the RDF subject, predicate and object to be searched. Moreover, sets of triple patterns define *Basic Graph Patterns (BGP)*, being each BGP a function that transforms the RDF datasets into the answer of a SPARQL query in the form of RDF triples. Traditionally, SPARQL queries can be categorized into *star*, *chain* and *complex* queries [8]. These query shapes depend on the location of the variables in the triple patterns, which can heavily influence the query performance [8].

There are many works that employs NoSQL systems for scalable RDF data management [5]. Among the recent works, we highlight *Rainbow* [2] (a polyglot NoSQL-based triplestore), *ScalaRDF* [4] (an in-memory solution) and *S2RDF* [8] (a scalable query processor). *Rainbow* is a distributed triplestore that uses the *HBase* columnar database and the *Redis* key-value database (K/V) as distributed storages to speed up query processing. Based on a previous analysis of the dataset and the expected workload, it decides on which NoSQL database the RDF data will be maintained. *ScalaRDF* introduces a distributed in-memory

[1] https://www.w3.org/.

triple store that uses *Redis* as a fault-tolerant and distributed RDF store. Additionally, *S2RDF* proposes a *Spark*-based SPARQL query processor that offers very fast response time for star queries by extending the vertical partitioning. Their partition scheme uses the *Apache Parquet*[2] columnar format to store the triples excluding unnecessary data from query processing. In order to reduce the intermediate results, S2RDF maintains statistics about the size of the dataset tables and places the subqueries corresponding to the smallest tables at the beginning of a joining in order to reduce the intermediate result size.

WA-RDF represents an advance on the state-of-the-art in sense that it is the first triplestore that considers the typical workload to decide which is the best NoSQL database to store an RDF triple.

3 WA-RDF

WA-RDF is a workload-aware middleware for storing and querying RDF data in multiple NoSQL database nodes. Its inspiration comes from *Estocada*, which argues that a mixed-model layer, relying on a set of diverse and heterogeneous data stores, can provide performance advantages for the applications using this layer. However, *Estocada* is neither a workload-aware approach nor a storage solution for RDF data. Another idea we borrowed from *Estocada* is a *fragment-based* storage that is entirely transparent to the client applications. It means that the data flow in *WA-RDF* is most of the time in the format of fragments. Figure 1 gives an overview of the *WA-RDF* architecture.

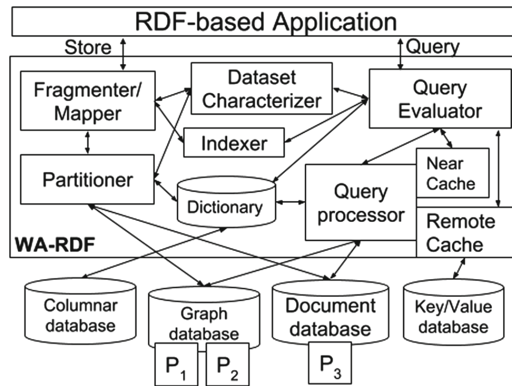

Fig. 1. WA-RDF architecture

An *RDF-based application* issues *store* or *query* requests to *WA-RDF*, which is normally deployed into multiple dedicated physical nodes. When an *RDF-based Application* submits a *store* request for a triple to the *Fragmenter/Mapper*

[2] https://parquet.apache.org/.

component, *WA-RDF* expands this triple to a fragment F_{RDFi} and maps F_{RDFi} to the target NoSQL database(s). This process is performed by the *Dataset Characterizer*, which is the main component of our middleware. During a triple storage, it decides on translating F_{RDFi} to a NoSQL document or graph database (or both) according to the usual query workload, and indexes it with the aid of the *Indexer* component. Once F_{RDFi} is created, the *Partitioner* registers this fragment into the *Dictionary* repository - supported by a NoSQL columnar database - and stores it in the NoSQL databases.

When an *RDF-based Application* submits a SPARQL *query* request, the *Query Evaluator* component decomposes this query into subqueries and reports to the *Dataset Characterizer* about them. In the following, the *Query Evaluator* verifies, with the aid of the *Dictionary*, the partitions on which the triples for the query are potentially located. Based on this information, it checks which triples are available in the *Near Cache* (a data structure in the main memory of the server) and the *Remote Cache* (a remote NoSQL key/value database), and sends the SPARQL subqueries for the missing triples to the *Query Processor* component that, in turn, translates them to graph and/or document NoSQL database queries. Finally, the *Query Processor* sends back the query results to the *Query Evaluator* that translates them back to RDF triples with the aid of the *Dictionary*, and returns the result to the *RDF-based Application*.

The main purpose of *WA-RDF* is to store large RDF graphs. In such a scenario, the number of RDF triples can easily surpass the performance capacity (*e.g.*, disk, memory, CPU) of a single server. When it occurs, *WA-RDF* distributes the RDF fragments among potentially many NoSQL nodes. A fragment is our smallest grain of distribution, *i.e.*, during the partitioning process we deal with fragments instead of triples. Nevertheless, a query can eventually access data in multiple partitions, forcing *WA-RDF* to join data from different partitions. Since a join operation is very costly, we try to avoid join processes by replicating fragments that are potentially part of a join. In short, whenever the typical workload for a fragment spans more than one partition, our *partitioning* scheme replicates the boundary fragments of the partition. Boundary fragments have triples that are connected to triples present in other partitions.

WA-RDF also provides an RDF indexing strategy. In this context, a traditional approach is to build indexes for the full set of permutations of each triple component (*subject (S), predicate (P)* and *object (O)*). Although this method has been designed to accelerate joins by some orders of magnitude, the overhead with large index space limits its scalability and makes it heavyweight. Hence, we developed a *hashmap index* with subject and object keys following the patterns *S-PO* and *O-PS*. In *WA-RDF*, the *Indexer* component is responsible to manage these indexes. It is accessed in two situations: *(i)* during the fragment creation; and *(ii)* to process queries with one triple pattern.

WA-RDF is an evolution of *Rendezvous* [7]. In this version, all the NoSQL databases are employed. Also, as stated before, a graph database replaced the columnar database for triple storage, and the dictionary uses now a columnar database as the main storage.

4 The Workload-Aware Approach

A *workload-aware* approach is the cornerstone of *WA-RDF*. Based on it, *WA-RDF* decides where to place each triple, which influences mapping, partitioning and querying strategies. In order to be aware of the typical workload, *WA-RDF* registers information about the *triple patterns* of each incoming SPARQL query. We consider triple patterns because they determine BGPs that define the query shape (star, chain or complex). For instance, in the SPARQL query `SELECT ?x WHERE { ?x p1 B }`, the triple pattern is `?x p1 B`. WA-RDF registers historical information about the queries into two *hashmaps*, as shown in the example of Figs. 2(iii) and 3(iii) for the RDF graph (i) of both figures. One hashmap registers all the chain-shaped queries indexed by the predicate, and the other one all the star-shaped queries indexed by the subject. For example, Fig. 2 (iii) shows that a typical star query around *C* containing the triple patterns `C p6 G, C p9 I` and `C p8 H`, and Fig. 3(iii) shows a chain query starting on *p1* containing the triple patterns `A p1 B, B p5 C` *and* `C p6 ?`.

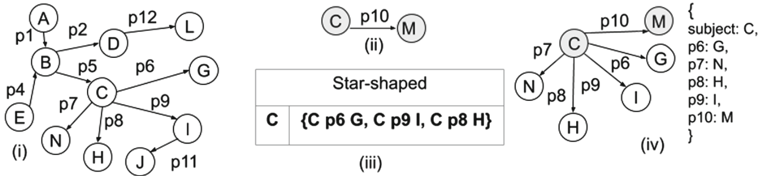

Fig. 2. Star fragmentation strategy

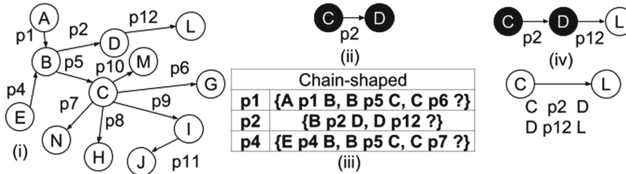

Fig. 3. Chain fragmentation strategy

4.1 Storage: Fragmentation and Partitioning

When a new RDF triple $t_{new} = (s, p, o)$ is inserted through *WA-RDF*, the hashmaps are checked to decide if t_{new} is more frequent on star or chain-shaped queries. Algorithm 1 shows the workload-based triple storage procedure. The input parameter is t_{new}, and it generates an RDF fragment f that is stored in one or more partitions. In Figs. 2(ii) and 3(ii), for instance, we have two new triples `C p10 M` and `C p2 D`, respectively. An RDF fragment represents an expansion of t_{new} (called *core triple*) with all of its neighbors according to a *n-hop replication horizon* managed by *WA-RDF*. The n-hop is used to avoid frequent

joins by wisely expanding the core triple to include its neighbors until a maximal distance n. The value n is calculated as the mode of the number of triple patterns in the queries related to t_{new}. For star queries, it is the most frequent diameter of the queries. For chain queries, it is the most frequent length of the queries. For example, the diameter mode for the triple `C p10 M` is 1 according to the frequent star-shaped query in the index of Fig. 2(iii).

Algorithm 1: Workload-based triple storage

```
    Input: Triple t_new
 1  if !exists(t_new) then
 2      f = new Fragment;
 3      f.core = t_new;
 4      indexSPO.put(t_new.s, t_new);
 5      indexOPS.put(t_new.o, t_new);
 6      f.shapes = getShapes(t_new);
 7      hop = 1;
 8      if f.shapes.contains('chain') then
 9          hop = chainHop(t_new);
10      if f.shapes.contains('star') then
11          if(starHop(t) > hop) hop = starHop(t_new);
12      f.triples = expand(t_new, hop, f.shapes);
13      writeToPartitions(f);
14  end
```

Back to Algorithm 1, if t_{new} does not exists (line 1), a new RDF fragment f is generated (line 2) and it initially holds the core triple (line 3). Next, the core triple is indexed in an SPO and OPS fashion (lines 4 and 5) in order to reduce response time of queries without joins and facilitate the query expansion. From line 6 to line 11, Algorithm 1 obtains the shapes and the n-hop size for the core triple. The n-hop size is defined as the size in terms of triple patterns of the biggest query in the typical workload for the core triple. It initially finds the shapes and registers them in the fragment f (line 6). If neither the predicate nor the subject exist in the chain and star hashmaps, respectively (no shape is found), it defaults to a star-shaped query with one triple n-hop size (hop = 1) (line 7). Otherwise, it determines the hop based on the found shapes (lines 8 to 11). In line 12, t_{new} is expanded to the n-hop size. *f.triples* is an array with up to 2 positions: one for the chain fragment and another one for the star fragment. In the example of Fig. 2, the new triple `C p10 M` is expanded to the RDF fragment in the left of Fig. 2(iv), and the new triple `C p2 D` to the RDF fragment in the top of Fig. 3(iv).

Formally, an *RDF Fragment* is a set $F_{RDFi} = \{t_{RDF}\}$ of RDF triples $t_{RDF} = (s, p, o)$ whose content may overlap with other fragment F_{RDFj}. After the document or graph fragment is created, *WA-RDF* distributes it among potentially NoSQL nodes (line 13). A NoSQL node can store one or more partitions. We discuss RDF data partitioning further on in this section.

It is important to observe here that a core triple can generate two RDF fragments (graph and document). It happens when the subject of this core triple is in the star hashmap and its predicate is in the chain hashmap at the same time. If an RDF fragment is translated to a document fragment, we have a mapping to a JSON document and it is stored into a NoSQL document database. If an RDF fragment is translated to a graph fragment, we have a mapping to a NoSQL graph database.

A *document fragment* is a tuple $f_{df} = (k_d, A)$ where $f_{df}.k_d$ is the JSON document key and $f_{df}.A = \{(k_\alpha : v)\}$ is a set of attributes, being k_α the attribute key and v a value whose domain can be atomic, a list, a set or a tuple. In short, the core triple t_{core} in the RDF fragment F_{RDFi} is mapped to a document whose key is $t_{core}.s$, and each outgoing predicate from the subject becomes a document attribute with a key $t_{core}.p$. If F_{RDFi} is 1-hop, the attribute value of each outgoing predicate is the object $t_{core}.o$ reached from it. Otherwise, the predicate value is an inner document that maintains the target object as the inner document key, and its outgoing predicates as attributes. If any of these outgoing predicates is, in turn, an n-hop, n $>$ 1, the generation of other inner documents proceeds recursively. Figure 2(iv) illustrates an RDF fragment (left) and its corresponding document fragment (right).

A *graph fragment* is a triple $f_{gf} = (s_g f, T, o_g f)$ where $s_g f$ is a vertex representing the first subject of a chain, $o_g f$ is a vertex representing the last object of a chain, and $T = \{t_n\}$ denotes an edge that holds a set of triples as property, *i.e.*, the intermediary triples between $s_g f$ and $o_g f$, including the object of the first triple and the subject of the last triple. A graph fragment summarizes a chain of triples by transforming this chain into a triple where the subject of the first triple and the object of the last triple are mapped to two vertexes, and the edge between these two vertexes is created with a property that maintains all the triples of the chain. In Fig. 3(iv) we see an RDF fragment (top) and a graph fragment obtained from it (bottom).

We now explain the partitioning strategy of *WA-RDF*. Given the RDF graph of Fig. 4 (the resulting graph after the storage of the triples `C p10 M` and `C p2 D` into the graph of Figs. 2(i) and 3(i)), the fragments are stored in document partitions (for instance, P_1) and/or in graph partitions (for instance, P_2 and P_3). In *WA-RDF*, a fragment is the finest unit for a partition. As defined in the following, a partition is a set of fragments stored into the same physical NoSQL node, and a fragment can be replicated in multiple partitions.

An *RDF Partition* P_m of an RDF graph G, such that $G \subseteq P_1 \cup P_2 \cup ...P_n$, is a set of RDF fragments $P_m = \{F_{RDFi}\}$, being not required that $P_m \cap P_t = \emptyset$, for m $\neq$ t. Also, given $SP = \{P_1, P_2, ..., P_n\}$ the set of RDF partitions, the *partition boundary* B_{P_i} of a partition $P_i \subset SP$ is the set of RDF fragments $B_{P_i} = Fb_{P_1} \cup Fb_{P_2} ... \cup Fb_{P_n}$, where $Fb_{P_k} \subset P_k$ for any k. Each $Fb_{P_i} \in B_{P_i}$ has one or more RDF triples $t_i F_{P_i} = (s_i, p_i, o_i)$ where $o_i = s_j$, being s_j the subject of any other triple $t_j F_{P_j}$ of a partition P_j where $t_j F_{P_j} = (s_j, p_j, o_j)$.

The *Dictionary* shown in Fig. 4 registers each fragment location. It holds three *hashsets* for each partition to keep track of the RDF elements stored in

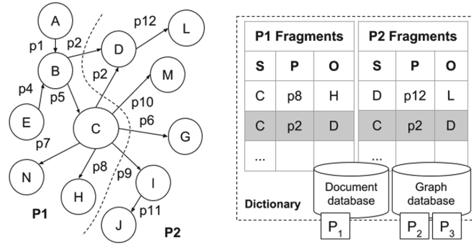

Fig. 4. Fragment partitioning

each partition (represented in the tables *P1 Fragments and P2 Fragments*), so during a query request we can avoid accessing unnecessary partitions that cannot answer this query. If a *WA-RDF* node manages more than one partition of a NoSQL database type, in face of a new core triple we have to decide which is the best partition to store its fragments. For doing so, *WA-RDF* finds out the typical workload for the triples that belong to the fragment generated by the core triple. With this information, we can query the partition sets in the *Dictionary* to verify in which partition this fragment can be more useful (this is represented by the line 13 of Algorithm 1) in sense that joins outside the fragment can be answered within a single partition. In Fig. 4, the size n = 1 for boundary replication repeats the fragment with core triple `C p10 D` in partitions *P1* and *P2*.

Algorithm 2 presents an overview of the query planning and partition processes. The input is the set of *triple patterns* from the query and the output is the result set *R*. If the query has only one triple pattern, the result is retrieved from SPO and OPS indexes (lines 1 and 2). Otherwise, Algorithm 2 looks for the shapes of the query to define its execution plan. Firstly (lines 4 to 6), *WA-RDF* loads the triple patterns into two multilevel hash tables *mhtSPO* and *mhtOPS* in order to speedup the further steps. Then, it looks for S-S star shapes (lines 11 to 14), O-O star shapes (lines 15 to 18) and chains (lines 20 to 26). The star shapes are identified when a subject has more than 2 entries in the *mhtSPO* (line 11), or an object have more than 2 entries on the *mhtOPS* (line 15). In this case, it expands the star shape with all the entries from the multilevel hash tables, registers the results in the star hashmap and add it to the query execution plan stored into the set *stars* that will be later translated to the document database query language (line 28). The triple patterns that do not define star shapes are expanded to chains (line 20). If the expanded chain has size 1 (*i.e.*, the triple pattern itself), the indexes are accessed to get the result triples (line 22 and 23). Otherwise, the expanded chain is registered in the chain hashmap and added to the query execution plan stored into the set *chains*, which is later translated to the graph database query language (line 29). Finally, with the aid of the *Dictionary*, after the *stars* and *chains* sets are processed by the document and graph databases, the algorithm returns the result set R (line 30).

Algorithm 2: Workload-based triple querying

```
   Input: SPARQL query triple patterns = {tp1, tp2, ..., tpn}, where
          tpi = (si, pi, oi)
   Output: Result set R = {t1, t2, ..., tm}
 1 if n == 1 then
 2     R.add(getFromIndex(tp1));
 3 else
 4     for i = 1 to n do
 5         mhtSPO.put(si, tpi);
 6         mhtOPS.put(oi, tpi);
 7     end
 8     stars = {};
 9     chains = {};
10     for i = 1 to n do
11         if mhtSPO.get(si).size() > 2 then
12             expandedStar = expandSubject(mhtSPO.get(si));
13             register(expandedStar, 'star', expandedStar.hop);
14             stars.add(expandedStar);
15         else if mhtOPS.get(oi).size() > 2 then
16             expandedStar = expandObject(mhtOPS.get(oi));
17             register(expandedStar, 'star', expandedStar.hop);
18             stars.add(expandedStar);
19         else
20             expandedChain = expandChain(tpi);
21             if expandedChain.horizon==1 then
22                 R.add(indexSPO.get(si));
23                 R.add(indexOPS.get(oi));
24             else
25                 register(expandedChain, 'chain', expandedChain.hop);
26                 chains.add(expandedChain);
27     end
28     R.add(readFromDocument(stars));
29     R.add(readFromGraph(chains));
30 return R;
```

4.2 Query Processing

From a performance point of view, the most important task accomplished by *WA-RDF* is the query processing. The queries analyzed by the *Query Evaluator* component are processed by the *Query Processor* component, which determines the best way to read data from the NoSQL databases.

The *Query Processor* usually has many options to process a query. Even so, its main strategy is to foster the early execution of triples with low selectivity to reduce the number of intermediate results and, consequently, to boost the query performance. Our work focuses on selectivity estimation of single BGPs based on statistics of the queried data.

Suppose, for example, the query Q in the following. It could be decomposed into BGPs that define star queries where `?x` and `?y` are the star shape centers, or BGPs that define chain queries that starts in `?x`, follows to `?y` and then goes to other nodes. However, during the query processing we have to decide if the process first execute one of the star or chain queries, as there are dependencies between the queries.

```
Q: SELECT ?x WHERE { ?x p1 ?y . ?x p2 ?z .
?x p3 ?w . ?y p5 ?k . ?y p6 G . ?k p7 ?l . ?l p8 H . ?l p8 J }
```

Suppose, for example, that the star-shaped BGP `?x p1 ?y . ?x p2 ?z . ?x p3 ?w` potentially returns 100 triples and the chain-shaped BGP `?x p1 ?y . ?y p5 ?k . ?k p7 l . ?l p8 J` returns only 10 triples. In this case, we would process first the chain-shaped BGP.

In short, the selectivity estimation is the number of triples that is returned for each BGP. This number depends on the shape of the query. For star shapes, it is calculated by the number of times that the center of the star (subject or object) is present in the *Dictionary*. For chain shapes, it is calculated as how many times the predicates of the chain are presented in the chain.

The selectivity is the input for the query translation processes accomplished by the *Query Processor* component into the target databases. The *star* queries (O-O or S-S joins) are converted to queries over NoSQL document databases. For instance, the *star* queries *Q1* (O-O) and *Q2* (S-S) in the following are converted to the access methods *D1* and *D2*, respectively (MongoDB NoSQL database syntax). The `$exists` function of MongoDB filters the JSON documents that have all the predicates of each query. In *D2*, we also filter by the subject `M`.

```
Q1: SELECT ?x WHERE {x? p5 y? . x? p2 z? .}
Q2: SELECT ?x WHERE {x? p9 y? . M p10 y? .}

D1: db.partition1.find({p5:{$exists:true},
p2:{$exists:true}}})
D2: db.partition1.find({p9:{$exists:true},
subject:M}})
```

The *chain* queries are converted to queries over NoSQL graph databases. For example, given the query *Q3* in the following, with O-S joins, *WA-RDF* translates it to the set of query *G1* according to the Cypher[3] query language of the Neo4J NoSQL database.

```
Q3: SELECT ?x WHERE {x? p1 y?. y? p2 z?. z? p3 w?.}

G1: MATCH (f:Fragment)
WHERE ANY(item IN f.p WHERE item = p1 OR
item = p2 OR item = p3)
RETURN p
```

[3] https://neo4j.com/developer/cypher-query-language/.

The processing of joins occurs when a query as a whole cannot be executed on a single partition. In this case, it needs to be decomposed into a set of subqueries, being each subquery evaluated separately and joined at the *WA-RDF* node.

For example, if we consider the graph of Fig. 4, the query *Q4* in the following is not able to be completed only querying the partitions *P1* or *P2* alone. In this case, the *Query Processor* divides it into subqueries *SQ5* and *SQ6*, issues it to the partitions *P1* and *P2*, respectively, and joins the result sets by matching the predicate `p9` (the connection between *P1* and *P2*).

Q4: `SELECT ?x WHERE {x? p1 y?. y? p5 z?.`
`z? p9 w?. w? p11 J.}`
SQ5: `SELECT ?x WHERE {x? p1 y?. y? p5 z?.`
`z? p9 w?.}`
SQ6: `SELECT ?x WHERE {z? p9 w?. w? p11 J.}`

As explained before, a complex query is a combination of the *star* and *chain* patterns, potentially connected by simple queries. Query *Q5* in the following is an example, where the BGP `x? p1 y? . y? p2 z? . z? p3 w?` is a *chain* pattern, the BGP `z? p5 ?k` is a simple query, and the BGP `k? p6 G . k? p7 I . k? p8 H` is a *star* pattern. In this case, the decomposition process works as follows: *(i)* it first sorts the triple patterns by subject and object; *(ii)* if it is identified a subset with two or more patterns with the same subject or object, it is considered a *star* subquery, like the subquery *P1* in the following. Then, chains are identified in the remaining query patterns, *i.e.*, *(iii)* for each triple pattern, we navigate from object to subject creating chains, and we pick up the longest chain and consider this a *chain* subquery, like subquery *P2*.

Q5: `SELECT ?x WHERE { x? p1 y? . y? p2 z? .`
`z? p3 w? . z? p5 ?k . k? p6 G . k? p7 I . k? p8 H }`
P1: `{k? p6 G . k? p7 I . k? p8 H }`
P2: `{x? p1 y? . y? p2 z? . z? p3 w?}`
P3: `{z? p5 ?k}`

We repeat step *(iii)* until there are no more chains, or there are only simple patterns, like the subquery *P3*. Each *star* and *chain* subquery is processed separately, and the join of the results (along with the simple patterns) is performed at the *WA-RDF* node. In case of ambiguity, *i.e.*, a pattern that is presented in more than one query type, we consider the following priority: *(1)* subject-based *star* query; *(2)* object-based *star* query; *(3)* the longest *chain* query; and *(4) simple* queries. The star queries are processed with high priority for two reasons: star queries are most common, and the MongoDB translation permits that we query mostly the document keys, what lets queries over documents much faster when compared to queries over graphs.

5 Experimental Evaluation

This section presents an evaluation of the proposed approach. The considered dataset comes from the *Lehigh University Benchmark (LUBM)* [3], which features an ontology for an *University* domain, synthetic RDF data, and 14 extensional queries representing a variety of properties. In our experiments, we generate a dataset with 4000 universities. The dataset size is around 100 GB and contains around 500 million triples. Regarding query complexity, we have 12 queries with joins, all of them having at least one star join, and 6 of them also having at least one chain join.

We ran experiments for data insertion and data querying to evaluate the performance and scalability of *WA-RDF*. *WA-RDF* was developed using Apache Jena version 3.2.0 with Java 1.8, and we use MongoDB 3.4.3 and Neo4J 3.2.5 as the document and graph NoSQL databases, respectively, on considering their maturity as representatives of these NoSQL data models. All the nodes are Amazon m3.xlarge spot instances[4] with 7.5 GB of memory and 1 × 32 SSD capacity. For all the experiments, the nodes represent the number of MongoDB + Neo4J servers, always with half of each database. We also create one partition for each server, and the *WA-RDF* servers were installed alone in each node. All the queries were issued from a server in the same network, so the latency between the client and *WA-RDF* was inexpressive.

We reproduce the query processing strategies of *Rainbow* and *ScalaRDF* because we could not find the implementation of these baselines in public repositories. To test *S2RDF*, we use the version found in GitHub[5], with small changes in the source code so we could use LUBM. The machines we use to run *Rainbow*, *ScalaRDF* and *S2RDF* are similar to the m3.xlarge of *WA-RDF*. We considered only one processing server for *Rainbow* and *ScalaRDF*, and we deployed an Apache Spark cluster with one master and 3 workers for *S2RDF* (the same size of our *WA-RDF* installation). The baselines were chosen because they hold different strategies: Rainbow also applies multiple databases by using Redis as a cache, ScalaRDF use a native storage along with Redis, and S2RDF uses Apache Spark.

Table 1 details the ingestion response time for three different triples. LUBM is a synthetic benchmark based on the educational domain, creating a model and data simulating a university with students, courses and professors. We first ran the queries *Q1* to *Q5* to provide workload information to *WA-RDF* and, in the following, we inserted the fragments *F1* to *F3*. The queries and the fragments are available at Appendix A. *F1* presents the insertion of a university. As shown in Table 1, the fragmentation is faster and only MongoDB was used. *F2* presents the insertion of a Department in the University of *F1*. During *F2* processing, the fragmentation phase is slower because the triples are expanded to include the University and, as the relation `ub:subOrganizationOf` is part of the chain in query *Q5*, it is added to Neo4J. *F3* inserts a professor that is also a chair of the department inserted before. It generates fragments for MongoDB and Neo4J so the fragmentation and partitioning tasks are slower than the other ones.

[4] https://aws.amazon.com/ec2/instance-types/.
[5] https://github.com/mxhdev/S2RDF_BSBM.

Table 1. Detailed ingestion time (ms)

Work	F1	F2	F3
WA-RDF - Parsing	9	12	13
WA-RDF - Fragmenting	13	19	23
WA-RDF - Indexing	5	6	4
WA-RDF - Partitioning	24	39	41
WA-RDF - Inserting MongoDB	102	-	204
WA-RDF - Inserting Neo4J	-	300	320
WA-RDF total	153	356	605
Rainbow	201	209	198
ScalaRDF	233	253	208
S2RDF	129	197	291

Table 2. Detailed query response time (ms)

Work	Q1	Q2	Q3	Q4	Q5
WA-RDF - Parsing	10	13	20	21	19
WA-RDF - Index access	13	14	11	18	15
WA-RDF - Decomposition	-	20	35	33	54
WA-RDF - MongoDB	-	70	102	123	132
WA-RDF - Neo4J	-	-	-	-	302
WA-RDF - Result set creation	5	20	30	40	60
WA-RDF total	28	144	199	233	582
Rainbow	33	162	203	594	1022
ScalaRDF	34	190	182	602	892
S2RDF	27	98	182	493	921

Table 2 details the querying response time for five different triples. The queries used here were proposed by Guo et al. [3]. For sake of simplicity, we discuss only a simple, a star, a chain and two complex queries, instead of all the queries available in LUBM. *Q1* is the most basic query, and it is solved directed by the *WA-RDF* OPS index. *Q2* is a small star-shape query around `X` that causes an access to MongoDB. *Q3* is composed of two stars connected by `Y`. It takes more time to generate the result set because some triples have to be cleaned. *Q4* is a big star composed of five BGPs. However, it is very fast to be processed by *WA-RDF* because we can solve it with only one MongoDB access. *Q5* is a complex query that is decomposed into two stars and a chain `(?X ub:memberOf ?Z . ?Z ub:subOrganizationOf ?Y . ?Y rdf:type ub:University.)`. It touches Neo4J and avoids multiple calls to MongoDB. As shown in Table 2, *WA-RDF* is specially interesting for complex queries like *Q4* and *Q5*.

6 Conclusion

This paper presents *WA-RDF*, a workload-aware RDF partitioning and querying approach for RDF data stored into NoSQL databases. We based it on a middleware that can, according to the typical shape of SPARQL queries, define RDF fragments and store them into the document and graph NoSQL databases. Our experiments show that *WA-RDF* outperformed three recent baselines in terms of large queries (*Q4* and *Q5*). For most of the other ones, we ran under the average of the baselines executions. However, there is still room for improvements regarding data ingestion time and storage size.

In general, *WA-RDF* is a contribution to the problem of efficient management of RDF data persisted into NoSQL databases. To the best of our knowledge, this is the first work that deals with RDF data fragmentation, partitioning and efficient query processing (including optimization issues to deal with intermediate results) for massive RDF graphs stored in multiple NoSQL databases. Even so, we have some future works in mind. First of all, we are considering the development of an algorithm for triples compression. The lack of this feature lets *WA-RDF* uses exponentially more storage space as the n-hop horizon grows. Moreover, we intend to consider update and delete operations and cluster capabilities in the *WA-RDF* server. With these improvements, we aim at comparing it again with the related work. Finally, we intend to evaluate *WA-RDF* against other benchmarks, like the *Waterloo SPARQL Diversity Test Suite (WatDiv)*.

A Fragments and Queries

F1 - Insert a university:
University0.edu rdf:type ub:University

F2 - Insert a department for the university:
Department0.University0.edu rdf:type ub:Department
Department0.University0.edu ub:subOrganizationOf University0.edu

F3 - Insert a professor for the department:
Professor0 rdf:type ub:Professor
Professor0 rdf:type ub:Chair
Professor0 ub:worksFor Department0.University0.edu

Q1 - SELECT ?X WHERE {?X rdf:type ub:UndergraduateStudent}

Q2 - SELECT ?X WHERE {?X rdf:type ub:GraduateStudent . ?X ub:takesCourse Department0.University0.edu GraduateCourse0}

Q3 - SELECT ?X, ?Y WHERE {?X rdf:type ub:Chair . ?Y rdf:type ub:Department . ?X ub:worksFor ?Y . ?Y ub:subOrganizationOf University0.edu}

Q4 - SELECT ?X, ?Y1, ?Y2, ?Y3 WHERE {?X rdf:type ub:Professor . ?X ub:worksFor Department0.University0.edu . ?X ub:name ?Y1 . ?X ub:emailAddress ?Y2 . ?X ub:telephone ?Y3}

Q5 - SELECT ?X, ?Y, ?Z WHERE {?X rdf:type ub:GraduateStudent .?Y rdf:type ub:University .?Z rdf:type ub:Department .?X ub:memberOf ?Z .?Z ub:subOrganizationOf ?Y . ?X ub:undergraduateDegreeFrom ?Y}

References

1. Bugiotti, F., Bursztyn, D., Diego, U.C.S., Ileana, I.: Invisible glue: scalable self-tuning multi-stores. In: CIDR 2015 (2015)
2. Gu, R., Hu, W., Huang, Y.: Rainbow: a distributed and hierarchical RDF triple store with dynamic scalability. In: Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014, pp. 561–566 (2015). https://doi.org/10.1109/BigData.2014.7004274
3. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. Web Semant. Sci. Serv. Agents WWW **3**(2), 158–182 (2005)
4. Hu, C., Wang, X., Yang, R., Wo, T.: ScalaRDF: a distributed, elastic and scalable in-memory RDF triple store (2016)
5. Ma, Z., Capretz, M.A.M., Yan, L.: Storing massive Resource Description Framework (RDF) data: a survey. Knowl. Eng. Rev. **31**(04), 391–413 (2016). https://doi.org/10.1017/S0269888916000217, http://www.journals.cambridge.org/abstract_S0269888916000217
6. Mello, R.D.S., et al.: Master: a multiple aspect view on trajectories. Trans. GIS (2019)
7. Santana, M.: Workload-aware RDF partitioning and SPARQL query caching for massive RDF graphs stored in NoSQL databases. In: Brazilian Symposium on Databases (SBBD), pp. 1–7. SBC (2017)
8. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on Spark. Proc. VLDB Endowment **9**(10), 804–815 (2016)