Workload-Aware RDF Partitioning and SPARQL Query Caching for Massive RDF Graphs stored in NoSQL Databases

Luiz Henrique Zambom Santana , Ronaldo dos Santos Mello

[1] Federal University of Santa Catarina (UFSC)
Florianópolis - SC - Brazil

luiz.santana@posgrad.ufsc.br, r.mello@ufsc.br

Abstract. *Governments, corporations, startups, open data initiatives and other organizations are increasingly considering RDF and SPARQL in a broad range of information management scenarios. To reduce SPARQL querying times has been the main issue for virtually all the recent RDF triplestores, yet SPARQL caching techniques have not been broadly considered. In this paper we present Rendezvous, a middleware that addresses workload-adaptive management of large RDF graphs with a caching strategy for SPARQL query results. Our middleware provides a novel RDF data partitioning approach based on a fragmentation strategy that maps RDF data into multiple NoSQL databases. The focus of this paper is also on Rendezvous caching, which can reduce average response time by up to an order of magnitude. Our experimental evaluation shows that the approach is promising, outperforming a recent key/value-based caching baseline.*

1. Introduction

RDF is a standardized data model that - along with other technologies like OWL, RDFS, and SPARQL - grounds the vision of Semantic Web as an initiative to foment interlinked machine-processable information [Berners-Lee et al. 2001]. In the last decade, RDF has been increasingly used in a wide range of data management scenarios (*e.g.*, data integration, search-engine optimization, data representation, information extraction) as a resource for better understanding of complex real-world entities and their relationship. However, the current scale of data intensive applications (*e.g.*, Smart Cities, Sensor Networks, eHealth, IoT) - all of them very attractive for the Semantic Web vision -, prevents the efficient usage of existing RDF storage systems operating on a single node. In fact, such a kind of system is becoming quite a performance bottleneck giving the actual generation of massive RDF data which goes beyond its processing capacities. It raises the need for innovations in the frontier of Big Data and Semantic Web research fields.

This paper presents *Rendezvous*, a middleware that 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 and into account the ideas from Estocada [Bugiotti et al. 2015] to develop a multiformat RDF storage based on its query workload to decide which NoSQL data model is the best fit for each incoming RDF fragment. The main contributions of this work are: *(i)* a mapping of RDF data to the columnar, document NoSQL and key/value data models [Sadalage and Fowler 2012]; *(ii)* a complex caching mechanism to store query results

both in each server and remotely in a key/value database; *(iii)* a workload-aware partitioner based on the current graph structure and, mainly, in the typical application workload; and, *(iv)* an experimental evaluation that compares our approach against a baseline (ScalaRDF [Hu et al. 2016]) by considering Apache Cassandra, MongoDB and Redis. Our high point is to process queries over large RDF graphs stored on multiple NoSQL servers with zero or a subtle amount data joining cost. An experimental evaluation shows that our middleware scales well, being able to process huge RDF datasets efficiently.

The rest of the paper is organized as follows. We give a brief background overview and discuss related works in Section 2. In Section 3 we detail the *Rendezvous* approach. We report our experimental evaluation in Section 4 and conclude the paper in Section 5.

2. Background and Related Work

The most important pillar of this work is the Semantic Web, as envisioned by Tim Berners-Lee in 2001 [Berners-Lee et al. 2001]. The Semantic Web offers, as practical value, the development of applications that can handle complex human queries based not only on simple matches of raw data but also on its meaning. When Semantic Web was presented, the exponential increase of information quantity could not be foreseen by most of the specialists, but the need for data integration was already argued as one of its fundamental purposes. Thus, in the recent years, the effort of developing the Semantic Web was harvested mainly in the form of well-established standards for expressing shared meaning, defined by WWW Consortium (W3C), 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. For instance, we can define a predicate *:owns* between two resources: *:person* (subject) and *:car* (object). SPARQL is a query language for searching and retrieving RDF information. A query statement in SPARQL consists of triple patterns, conjunctions, disjunctions, and optional patterns. The *triple pattern* defines the RDF subject, predicate and object to be searched, *conjunctions and disjunctions* express the intended relations between the searched resources, and the *optional patterns* combine two graph models. Moreover, sets of triple patterns define *Basic Graph Patterns (BGP)*, being each BGP a function that transforms the RDF dataset into mapping sets. Finally, these mapping sets are the answer to an SPARQL query in the form of RDF triples.

Traditionally, the SPARQL queries can be categorized into *star*, *chain* and *complex* queries [Gallego et al. 2011]. These shapes depend on the location of variables in triple patterns which can influence the query performance. The diameter of an SPARQL BGP is defined as the largest connected sequence of triple patterns, disregarding the edge direction. The *star* pattern has a diameter of one and is characterized by subject-subject joins within triple patterns as the join variable is located on the subject. *Chain* patterns are very common in graph querying (*e.g.*, friend-of-a-friend), being formed by object-subject joins, *i.e.*, the join variable is in the subject location in one pattern and on object location in the other one. *Complex* patterns are combinations of several star patterns connected by typically a single pattern. Other query structures are compositions of these major patterns.

The use of RDF to describe semantic data has seen a dramatic increase over the

last years, making RDF data almost ubiquitous (see LODStats[1]). As a consequence, recent surveys have highlighted the joint usage of RDF and NoSQL by Big Data applications [Ma et al. 2016]. NoSQL databases, as defined, in a consensual way, by Sadalage and Fowler [Sadalage and Fowler 2012], means database systems that use more than one storage mechanism, including new types not compatible with the traditional relational databases . NoSQL databases are usually organized into the following categories w.r.t. their data models in *key-value*, *document*, *columnar* and *graph*.

In this paper, we focus on the document, columnar, key/value NoSQL data models. The document data model is suitable to store semistructured data in one of the formats considered by current computational systems (*e.g.*, XML and JSON). *MongoDB* is the most popular NoSQL document database, being already tested for storing RDF as JSON documents [Mulay and Kumar 2012]. The columnar data model aims at storing data that do not respect a rigid schema but belong to the same domain of data, *i.e.*, data instances that usually hold a standard set of properties (columns), but may have a different number of columns. Apache Cassandra is the most popular NoSQL columnar database, and it is also used for RDF storage in the CumulusRDF approach [Ma et al. 2016]. Finally, the Key/value databases are also helping the RDF solutions to scale, for instance, *ScalaRDF* [Hu et al. 2016] is a triple store that stands out for persisting data on the key/-value database Redis as a distributed memory storage to speed up query performance.

There are many works proposed on polyglot NoSQL databases and scalable RDF data management [Ma et al. 2016], denoting that these are very hot topics. Among polyglot NoSQL databases the *Estocada* [Bugiotti et al. 2015], stands out as an architecture for handling highly heterogeneous datasets that provide a middleware capable of hosting multiple data sets as a set of potentially overlapping fragments, distributing the various fragments of each dataset across different stores, including SQL and NoSQL databases. *Rendezvous* goes beyond *Estocada* by focusing the dataset type in the RDF format and the storage exclusively in NoSQL storages. We also propose both a fragment expansion and a partition scheme for avoiding joins between data coming from different NoSQL nodes that can reduce the response time, especially in queries touching a large amount of data. ScalaRDF [Hu et al. 2016] is a important representative of scalable RDF data management, by presenting a distributed in-memory RDF triple store that uses Redis in a fault-tolerant store and query mechanism. *Rendezvous* polyglot capabilities for data storage - we use document, columnar and key/value databases -, along with the *n-hop* fragmentation scheme and the workload-awareness and complex caching solution, makes our approach more suitable to dynamic query workload and offers interactive querying response time over large RDF graphs.

3. Rendezvous

Rendezvous is a middleware for partitioning and storing RDF data in multiple NoSQL database nodes. Although its focus is on storing RDF triples, its inspiration comes from *Estocada* (see Section 2), 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. Figure 1 presents an overview of *Rendezvous* architecture. A *RDF-based application* issues storing or querying requests to *Rendezvous*, that is nor-

[1]http://stats.lod2.eu/

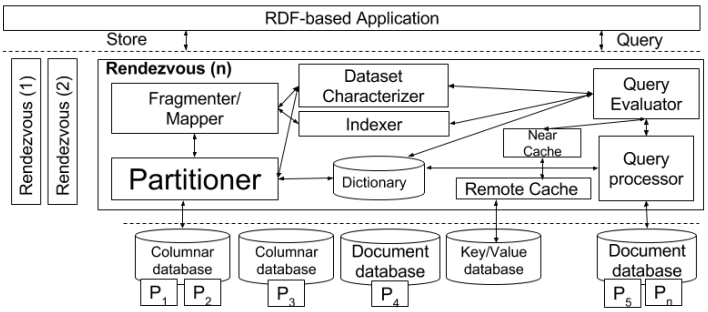

Figure 1. Rendezvous Architecture

mally deployed into multiple dedicated physical node. Thus, one could integrate several applications on top of *Rendezvous* using RDF as a common data model. Another idea that we borrowed from *Estocada* is the development of a fragment-based storage which is entirely transparent to the client applications. The data flow in *Rendezvous* is most of the time in the format of fragments. As defined in the following, a fragment is essentially a part of the RDF graph to be stored and retrieved into/from NoSQL databases. A **RDF Fragment** is a set $F_{RDFi} = \{t_{RDF}\}$ of triples whose content may overlap with other fragment F_{RDFj}, where t_{RDF} is a RDF triple $t_{RDF} = (s, p, o)$ where $t_{RDF}.s$ is the subject, $t_{RDF}.p$ is the predicate and $t_{RDF}.o$ is the object.

When an *RDF-based Application* issues a store request for a triple to the *Fragmenter/Mapper* component, *Rendezvous* expands this triple to a fragment F_{RDFi} and maps F_{RDFi} to the target NoSQL database(s). This process (see Section 3.1) is performed by the *Dataset Characterizer*, the main component of our middleware. During a triple storage, it decides on translating F_{RDFi} to a columnar or document data (or both) according to the usual query workload, and indexes it in the *Indexer*. Once F_{RDFi} is created, the *Partitioner* register this fragment into the *Dictionary* repository - designed as a *in-memory hashmap* - and stores it in the NoSQL databases (see Section 3.1). When an *RDF-based Application* issues a query request, the *Query Evaluator* component decomposes this query into star-shaped and chain-shaped subqueries and reports to the *Dataset Characterizer* about these queries (see Section 3.2). In the following, the *Query Evaluator* verifies, with the aid of the *Dictionary*, the partitions in which the triples for the query are potentially located. Based on this information, the *Query Processor* translates the SPARQL query to columnar and/or document database queries. Finally, again with the aid of the *Dictionary*, the component *Query Evaluator* translates back the query results to RDF triples and returns to the *RDF-based Application*.

The primary purpose of *Rendezvous* is to store large RDF graphs. In such scenario, the number of RDF triples can easily surpass the performance capacity (*e.g.*, disk, memory, CPU) of a single server. When it occurs, *Rendezvous* distributes the RDF fragments among potentially many NoSQL nodes. Notice that a fragment is our smallest grain of distribution, *i.e.*, during the partitioning process we deal with fragments instead of triples. In *Rendezvous*, as defined in the following, an RDF partition is a set of fragments stored in the same physical NoSQL node, and a fragment can be replicated in multiple partitions. A **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. Nevertheless,

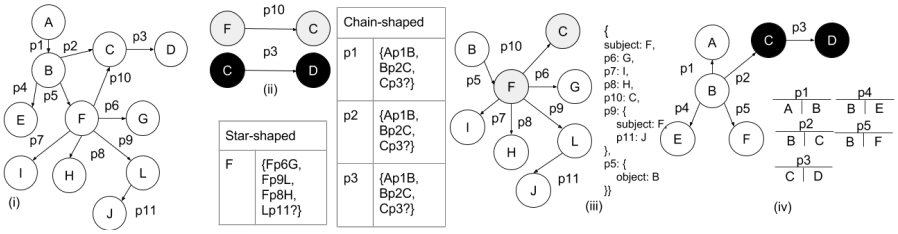

Figure 2. Fragmentation process

a query can eventually access data in multiple partitions, forcing *Rendezvous* to join the data from different NoSQL nodes. Since a join operation is very costly, we try to avoid join processes by replicating fragments that are potentially part of a join. As defined in the following, if the *typical workload* for a fragment spans more than one partition, our partition scheme replicates the boundary fragments of the partition. 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 $o_i = s_j$ been s_j subject of any other triple $t_j F_{P_j}$ of partition P_j where $t_j F_{P_j} = (s_j, p_j, o_j)$.

On RDF indexing, a traditional approach is to build indexes for the full set of permutations of each triple component (subject, predicate, object). Although this method has been designed to accelerate joins by some orders of magnitude, the overhead with the large index space limits its scalability and makes it heavyweight. Hence, we decided to develop a hashmap index with subject and object keys following the patterns *S-PO* and *O-PS* [Weiss et al. 2008]. In *Rendezvous*, the component *Indexer* is responsible to manage these indexes, being accessed in two situations: *(i)* during the fragment creation, explained in the Section 3.1 and *(ii)* to process queries that inform only the object or the subject, but not the predicate.

3.1. Storing: Fragmentation, Mapping and Partition

The data mapping is the most prominent *Rendezvous* task during a *storing* process. As stated before, our proposal is based on RDF fragmentation, so we first define our fragmentation strategy and the supported types of fragments. An RDF fragment is created when a new RDF triple to be stored (called *core triple*) is expanded with all of its neighbors according to a *n-hop replication horizon*. The parameter *n* is the number of predicates from the *core triple* that our fragment expansion process considers. This parameter is controlled by the *Dataset Characterizer* component, being defined as the 75th percentile of the total number of joins of the typical workload over the *core triple* elements. Figure 2 (i) shows an RDF graph, Figure 2 (ii) represents two core triples and Figure 2 (iii and iv) is the 1-hop fragment for both core triples.

The main reasoning for our fragmentation strategy in several RDF fragments is to maximize the NoSQL query capability and to minimize the cost of joining data in the *Rendezvous* node. As presented in Figure 2, if two core triples (*F p10 C* and *C p3 D*) have to be stored in the RDF graph of Figure 2 (i), we invoke the component *Dataset Characterizer*, responsible for keeping track of the typical query workload. It manages two *in-memory hashmaps*: one for the *star-shaped* queries, where the key is a subject or a object (for subject-subject and object-object joins, respectively - see Section

2), and another for the *chain-shaped* queries, where the key is the predicate. The *Dataset Characterizer* also decides the size of the fragment (*n*-hop) for the star-shaped case (*n* is the longest chain among the queries in the hashmap), and for the chain-shaped case (*n* is the size of the biggest query in the hashmap). In the Figure 2, the n-hop for star-shaped fragments with F as the subject is 2, and for the predicates, p1, p2, and p3 is also 2. If the subject of the new triple is defined in the star-shaped hashmap, the triple is converted into a document fragment, or an existing document fragment is updated in the NoSQL document database (Figure 2 (iii)). Otherwise, if the predicate of the new triple is defined in the chain-shaped hashmap, we create and store a columnar fragment in the columnar NoSQL database, or we update the subject and/or object of this triple in an existing column family (Figure 2 (iv)).

The same fragment can be mapped to both document and columnar fragments if the subject or object of the core triple of this fragment is in the star-shaped hashmap and its predicate is in the chain-shaped at the same time. If an RDF fragment is translated to a *document fragment* we have a mapping to a JSON document[2], which is the standard format for NoSQL document databases. If an RDF fragment is translated to a *columnar fragment*, we have a mapping to a column family which is the typical logical structuring for NoSQL column databases. The definition of these two types of a fragment, as well as the mapping of RDF fragments to them, are given in the following.

Document Fragment is a tuple $f_{doc} = (k_d, A)$ where $f_{doc}.k_d$ is the JSON document key and $f_{doc}.A$ is a set of attributes (key-value pairs) $f_{doc}.A = \{(k_\alpha : v)\}$, being k_α the attribute key and v a value whose domain can be atomic, a list, a set, or a tuple. The **RDF-to-Document Fragment Mapping** of F_{RDFi} to f_{doc} proceeds as follows: (i) given the core triple $t_{core} \in F_{RDFi}$: $f_{doc}.k_d \leftarrow t_{core}.s$; (ii) for each $t_{core}.p \in F_{RDFi}$ ($t_{core}.p$ is a t_{core} outgoing edge $\wedge$ F_{RDFi} IS a 1-hop fragment): $a_j.k_\alpha \leftarrow t_{core}.p$ and $a_j.v \leftarrow t_{core}.o$, being $a_j \in f_{doc}.A$ and $t_{core}.o$ reached from $t_{core}.p$; (iii) for each $t_{core}.p \in F_{RDFi}$ ($t_{core}.p$ is a t_{core} outgoing edge $\wedge$ F_{RDFi} IS NOT a 1-hop fragment): $a_j.k_\alpha \leftarrow t_{core}.p$ and $a_j.v$ is an inner document generated from the mapping of t_j to a document fragment recursively, being t_j an RDF triple where $t_j.s$ is the object reached from $t_{core}.p$. In short, the core triple t_{core} in the RDF fragment F_{RDFi} is mapped to 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. One example is shown in Figure 2 (iii).

Columnar Fragment is a tuple $f_{cf} = (k_cf, C)$ where $f_{cf}.k_cf$ is the name (key) of the column family and $f_{cf}.C$ is a set of columns (key-value pairs) $f_{cf}.C = \{(n_c : v)\}$, being n_c the column name (or column key) and v an atomic value. The **RDF-to-Columnar Fragment Mapping** of F_{RDFi} to columnar fragments proceeds as follows: for each predicate $t_k.p$ of a triple $t_k \in F_{RDFi}$: generate a columnar fragment $f_{cfi} = (k_cfi, \{c1, c2\})$ such that $f_{cf}.k_cfi \leftarrow t_k.p$, $f_{cf}.C.c1 \leftarrow t_k.s$ and $f_{cf}.C.c2 \leftarrow t_k.o$. A columnar fragment maintains a triple predicate of an RDF fragment. For each predicate we define a column family - named with the predicate label - and two columns to hold

[2]https://www.w3.org/TR/json-ld/

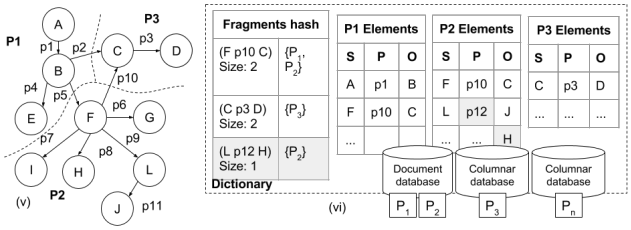

Figure 3. Fragment partitioning

the triple subject and triple object names. One example is shown in Figure 2 (iv). The decision for such a mapping strategy allows that, given a predicate, we query the columnar database using the subject and/or object as a filter, which is usually the scenario for chain-shaped queries.

If the number of RDF triples exceeds the performance capacity (*e.g.*, disk, memory, CPU) of a single server, *Rendezvous* distributes the RDF fragments among potentially many NoSQL nodes. In general, an RDF graph can be partitioned vertically, horizontally, and hash-based [Ma et al. 2016]. In a hash-based partitioning, a triple is placed into a partition based on the result of a hashing algorithm. The benefit of this approach is to distribute data evenly among the nodes, but several cross partition joins can be necessary to respond to a query request. To obtain better performance, the *Rendezvous* fragmentation process leads to horizontal partitioning [Mulay and Kumar 2012] for storage in a document database to deal with star-shaped queries, and vertical partitioning [Abadi et al. 2009] for a columnar database to tackle chain-shaped queries.

In *Rendezvous* the most basic element of partition is an RDF fragment. Given the RDF graph of Figure 2 (iii) the fragments (iv) were stored respectively in the document partition P_2 and the columnar partition P_3, and each fragment location is stored in the *Dictionary* repository, as presented in Figure 3. This repository maintains a hashmap with a key based on the core triple and its size and the partitions where this fragment is stored as the value. We also use three hash sets for each partition to keep track of the RDF elements (subjects, predicates, and objects) stored in this partition, so during a query request, we can avoid accessing unnecessarily partitions that cannot answer this query. If a *Rendezvous* node manages more than one partition of a NoSQL type, in face of a core triple, we have to decide which is the best partition to store its fragments. For doing so, the information provided by the *Dataset Characterizer* is used to find the typical workload for the triples present in 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, in the sense that the joins outside the fragment can be answered within a single partition. For instance, for a fragment with core triple *L p12 H* the best fit would be P_2, since all the typical workload queries ($\{$*Fp6G, Fp9L, Fp8H, Lp11?*$\}$) can be answered in this single partition.

Another point to highlight is that our workload-awareness is limited since the queries issued against *Rendezvous* are dynamic. Thus, in order to increase the fraction of queries that can be processed by accessing only one partition, we have two options: *(i)* to increase the n-hop replication horizon, which would ultimately result in a high amount of replicated data, or *(ii)* to choose a reasonable replication extent, *e.g.*, n = 2, and

add systematic replication fragments in the boundary of each partition. We advocate the second option since, as we show in Section 4, it generates a small storage overhead. In Figure 3, the boundary replication (with size n=2) is presented between partitions the P_1, P_2, and P_3. For instance, the fragment with core triple F p10 C and size 2 is in partitions P_1 and P_2, and the elements F, p10, and C are placed in more than one partition.

3.2. Querying Decomposition and Caching

Another important task accomplished by *Rendezvous* is the query decomposition. The input SPARQL queries are analyzed by the *Query Evaluator*. It, in turn, classifies a query into *simple*, *star-shaped*, *chain-shaped* or *complex*. A query is called *simple* if it does not involves a join in any triple component. If the query is classified as *complex*, it can be decomposed into *star-shaped* and/or *chain-shaped* subqueries. The *Query Evaluator* then reports to *Dataset Characterizer* in order to keep the workload metrics up-to-date, and accesses the *Dictionary* to get the partitions storing the triples for these queries.

The *star-shaped* queries (object-object or subject-subject joins) are converted to queries over NoSQL document databases. For instance, the object-object star-shaped query Q1 in the following is converted to the access method D1 (MongoDB NoSQL database syntax), and the subject-subject star-shaped Q2 is converted to the access method D2. The `$exists` function of MongoDB filters the JSON documents that have all the predicates of each query, moreover in D2 we also filter by the subject M.

Q1: `SELECT ?x WHERE {x? p5 y? . y? 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-shaped* queries are converted to queries over NoSQL columnar databases. For example, given the query Q3 in the following, with object-subject and subject-object joins, *Rendezvous* translates it to the set of queries C1 according to the Cassandra NoSQL columnar database query language syntax.

Q3: `SELECT ?x WHERE {x? p1 y?. y? p2 z?. z? p3 w?.}`
C1:
```
SELECT S1,O1 FROM p1
  SELECT S2,O2 FROM p2 WHERE O=S1
  SELECT S3,O3 FROM p3 WHERE O=S2 AND S=D (C1)
```

The processing of joins occurs when a query as a whole cannot be executed on a single partition, and it needs to be decomposed into a set of subqueries, being each subquery evaluated separately and joined at the *Rendezvous* node. For instance, if we consider the graph of Figure 3, 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 Decomposer* 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 *p5* (the connection between P1 and P2).

Q4: `SELECT ?x WHERE {x? p2 y?.y? p3 z?.x? p5 w?.w? p9 k?.L p11 k?.}`
SQ5: `SELECT ?x WHERE {x? p2 y?. y? p3 z?. x? p5 w?.}`
SQ6: `SELECT ?x WHERE {x? p5 w?. w? p9 k?. L p11 k?.}`

As explained before, a complex query is a combination of the star-shaped and chain-shaped patterns, potentially connected by simple queries. The query Q5 in the fol-

lowing presents an example, where *x? p1 y? . y? p2 z? . z? p3 w?* is a chain-shaped pattern, *z? p5 ?k* is a simple query, and *k? p6 G . k? p7 I . k? p8 H* is a star-shaped. In this case, our decomposition process works as follows: *(i)* it first normalizes the query by sorting the patterns by subject and object; *(i)* if we can identify a subset with two or more patterns with the same subject or object, we consider this a star-shaped subquery, like the P1 query below. Then we identify chains in the remaining of the query patterns, i.e.; *(iii)* for each pattern, we navigate from object to subject creating chains. Then we pick the longest chain and consider this a chain-shaped sub-query, like query P2 below. We repeat the step *(iii)* until there are no more chains, or there are only simple patterns, like the query P3 below. Each star-shaped subquery and chain-shaped subquery is processed separately, and the join of the results - along with the simple patterns - is performed at the *Rendezvous* node. In case of ambiguity, *i.e.*, a pattern that is present in more than one query type, we consider the following priority: *(1)* subject-based star-shapes; *(2)* object-based star-shapes; *(3)* the longest chain shapes; and *(4)* simple patterns.

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`}

Rendezvous also provides caching management during the query decomposition. Basically, the *Cache* component verifies, during a query processing, if a fragment (or even a query result) is maintained in the *Rendezvous* cache. The cache is organized as key/value fragments, as defined in the following. **Key/Value Fragment** is a tuple $f_{kv} = (k_kv, V)$ where $f_{kv}.k_kv$ is the name of the key with the form $t_{core}.s : t_{core}.p : t_{core}.o$ (the concatenation of the t_{core} components), and the value $f_{kv}.V$ is a set $f_{kv}.V = \{(t_{1-hop})\}$, being each $t_{1-hop} \in f_{kv}.V$ a triple with a 1-hop distance of t_{core}. *Rendezvous* holds two types of cache: the *Near Cache*, designed as an in-memory *TreeMap*[3] located on each *Rendezvous* server, and a *Remote Cache* maintained as a remote key-value NoSQL database (see Figure 1). When a query is issued against a *Rendezvous* server, it firstly checks its *near cache* to get all the fragments that are already available in the server. Then, the server accesses the *remote cache* to get all the missing fragments, and finally queries the document and columnar databases. The *TreeMap* used in the *near cache* provides an efficient way of storing key/value pairs in sorted order, which is specially interesting to the chain-shaped queries. On using a key-value NoSQL database in the *remote cache*, it is possible to search for keys with wild-cards and patterns (*e.g.*, *Redis*). If the database does not provide this feature, we create all the permutations of indexes for an RDF triple.

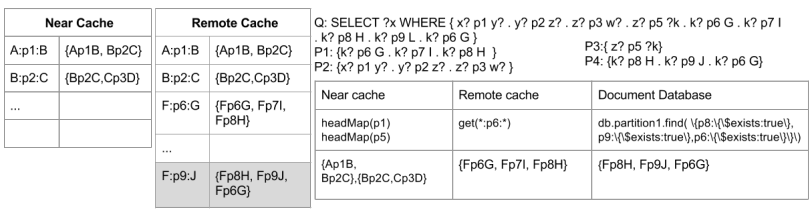

Figure 4. Types of *Rendezvous* Caching

[3]https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

Figure 4 represents the state of the caches when the query *Q* is issued to a *Rendezvous* server containing the graph of Figure 2 *(i)*. First of all, the server decomposes *Q* into *P1*, *P2*, *P3*, and *P4*. Then, it checks, for the simple and chain-shaped subqueries in the near cache (in this case, *P3* and *P2*, respectively), if any of the triple predicates is the root of a tree containing this chain (in Figure 4, it is represented by the Java commands *headMap(p1)* and *headMap(p5)*). Concurrently, it gets the star-shaped fragments of subquery *P1* from the *remote cache* (*e.g.*, *get(*:p6:*)*). In Figure 4, *P1* returns the necessary triples, but no triples are found in the caches for *P4*. Thus, the server queries the NoSQL databases, as explained in Section 3.2. The triples returned from the queries are stored in both the *near* and *remote caches*. In Figure 3, it is represented by the triples *Fp8H, Fp9J, Fp6G* retrieved from subquery *P4*.

When the cache is almost full, *Rendezvous* automatically evicts some keys. *Rendezvous* currently implements a cache eviction policy for *Most Recently Used (MRU)* as well as *Most Frequently Used (MFU)*. Moreover, the *near cache* is designed to not have false positives, *i.e.*, we prefer to double check with the *remote cache* instead of returning inconsistencies to the client. Due to it, we had developed a communication channel between the *Rendezvous* servers that is able to communicate each other about any update in an existing triple. In Figure 4, if the triple *t=(A, p1, B)* is updated to *t'=(A, p1, B')* in one of the servers S_i, *Rendezvous* removes triple *t* from S_i *remote cache* and send a signal to all the other servers to remove all the fragments that contains *t* from their *near caches*.

4. Experimental Evaluation

This section presents an evaluation of the proposed approach through an experimental evaluation. The considered dataset comes from the *Lehigh University Benchmark (LUBM)* [Guo et al. 2005]. LUBM features an ontology for the University domain, synthetic RDF data scalable to any size, 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 twelve queries with joins, all of them have at least one subject-subject join, and six of them also have at least one subject-object join. We ran experiments for data loading and querying to test the performance and scalability of *Rendezvous*. *Rendezvous* was developed using Apache Jena version 3.2.0 with Java 1.8, and we use Redis 3.2, MongoDB 3.4.3, and Apache Cassandra 3.10 as the key/value, document and columnar NoSQL databases, respectively, on considering their maturity as representatives of these three families of NoSQL storages. All the nodes are Amazon m3.xlarge spot instances[4] with 7.5 GB of memory and 1 x 32 SSD capacity. For all the experiments, the nodes represent the number of MongoDB + Cassandra servers, always with half of each database. Moreover, we created a cluster of Redis in 3 nodes with the same configuration. We also created one partition for each server, and the *Rendezvous* 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 *Rendezvous* was inexpressive.

The total dataset size, the loading time, and the average querying time are shown in Figure 5 (a) to (c), respectively. In Figure 5 (a) and Figure 5 (b), we notice that the dataset and the loading time grow exponentially with the number of nodes and the n-

[4]https://aws.amazon.com/ec2/instance-types/

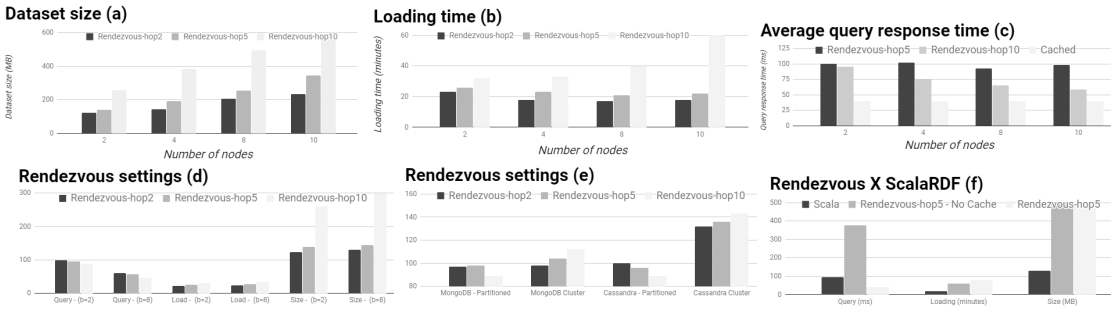

Figure 5. Summary of the Experimental Evaluation

hop, ramping up from around 102 GB, loaded around 20 minutes in the 2 nodes with 2-hop configuration, to more than 500 GB loaded in more than 60 minutes in the 10 nodes with 10-hop configuration. These results can make *Rendezvous* very costly in cloud environments that charge per storage usage. We are investigating compression techniques to mitigate this problem. In Figure 5 (c), we studied both the not cached configuration and cached configuration. In the not cached configuration, the best response time was achieved with 10 nodes and a 10-hop. In such a configuration, we did not register any join outside of the fragment - with this significant fragment size all the queries could be solved within a unique NoSQL access - and each server CPU and Memory load were very small. The cached configuration presents the same response time (around 40 ms in average) regardless the hop size. The results show that the fragmentation and partition solution of *Rendezvous* is scalable and if the tradeoff for the dataset size is acceptable, the average response time can be subtle, and that cache is a good solution for scalability. In Figure 5 (d) and (e) we compared different settings of *Rendezvous*. In Figure 5 (e), we show that the systematic replication of fragments in the boundary of each partition ("b" parameter) increases the speed on the query response, without a big impact in the total size of the dataset and the data load time. This is because the size of the boundaries' triples is not very significant in such big dataset. This result is motivating new studies on the optimal boundary replication size to accelerate the query response. In Figure 5 (d) we compared our partitioning solution to the NoSQL database partition solutions. We analyzed here *Rendezvous* accessing each NoSQL database server separately (as a partition), as well as accessing the servers as a cluster (delegating the data partition to the NoSQL database). The results show that, especially for Cassandra, the graph awareness of the proposed schema plus the replication boundary lead to better performance. For MongoDB, we can conclude that the most important factor is the size of the fragment (n-hop) since a bigger fragment will typically lead to a smaller number of database accesses. Finally,in Figure 5 (f) we compared the performance of *Rendezvous* 5-hop - cached and not cached - with the recent related work *ScalaRDF*[5]. Our cached solution is 30 percent faster on average. This results is mainly due our Near Cache component - which is not present in ScalaRDF - and avoid network latency between the Rendezvous server and Redis. The downsides of this comparison are the loading time - is almost twice slower - and the dataset size - almost five times bigger.

[5]The code for ScalaRDF was found in https://github.com/xinghuayu007/ScalaRDF/

5. Conclusion

This paper presents *Rendezvous*, a novel workload-aware RDF partitioning approach for the persistence of RDF data into NoSQL Databases. We based it on a middleware that can, according to the typical shape of the main SPARQL queries, define RDF fragments and store them into the document, columnar and key/value NoSQL databases. The partition is used when the dataset is bigger than each server capabilities. In this case, we considered a replication boundary to avoid cross-server joins and speed up the query response time. Our experiments reveal that a bigger replication boundary can accelerate the queries without a negative impact regarding storage space and load time. Besides, *Rendezvous* outperformed a recent disk-based baseline, denoting that our proposal is promising. In general, *Rendezvous* is a contribution to the problem of efficient mapping of the RDF data model to NoSQL data models. Even so, we have some future works in mind. First of all, we are considering implementing algorithm for triples compression. The lack of this feature makes *Rendezvous* uses exponentially more storage space as the n-hop horizon grows. We also intend to consider update and deletion operations, other NoSQL types in the *Rendezvous* architecture as well as cluster capabilities in the *Rendezvous* server. With these improvements, we aim at comparing it again with the related work.

References

Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2009). Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 18(2):385–406.

Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.

Bugiotti, F., Bursztyn, D., Diego, U. C. S., and Ileana, I. (2015). Invisible Glue : Scalable Self-Tuning Multi-Stores. *Cidr 2015*.

Gallego, M. A., Fernández, J. D., Martínez-Prieto, M. A., and de la Fuente, P. (2011). An empirical study of real-world sparql queries. In *USEWOD workshop*.

Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, S. and Agents on the WWW*, 3(2):158–182.

Hu, C., Wang, X., Yang, R., and Wo, T. (2016). Scalardf: a distributed, elastic and scalable in-memory rdf triple store.

Ma, Z., Capretz, M. A., and Yan, L. (2016). Storing massive resource description framework (rdf) data: a survey. *The Knowledge Engineering Review*, 31(4):391–413.

Mulay, K. and Kumar, P. S. (2012). Spovc: a scalable rdf store using horizontal partitioning and column oriented dbms. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, page 8. ACM.

Sadalage, P. J. and Fowler, M. (2012). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019.