Semantic Partitioning for RDF Datasets

Georgia Troullinou, Haridimos Kondylakis, Dimitris Plexousakis

Institute of Computer Science, FORTH, N. Plastira 100, Heraklion, Greece

{troulin, kondylak, dp}@ics.forth.gr

Abstract. Today we are witnessing an explosion in the size and the amount of the available RDF datasets. As such, conventional single node RDF management systems give their position to clustered ones. However most of the currently available clustered RDF database systems partition data using hash functions and/or vertical and horizontal partition algorithms with a significant impact on the number of nodes required for query answering, increasing the total cost of query evaluation. In this paper we present a novel semantic partitioning approach, exploiting both the structure and the semantics of an RDF Dataset, for producing vertical partitions that significantly reduce the number of nodes that should be visited for query answering. To construct these partitions, first we select the most important nodes in a dataset as centroids, using the notion of *relevance*. Then we use the notion of *dependence* to assign each remaining node to the appropriate centroid. We evaluate our approach using three real world datasets and demonstrate the nice properties that the constructed partitions possess showing that they significantly reduce the total number of nodes required for query answering while introducing minimal storage overhead.

1 Introduction

The recent explosion of the Data Web and the associated Linked Open Data (LOD) initiative have led to an enormous amount of widely available RDF datasets. For example, data.gov comprises in more than 5 billion triples, the Linked Cancer Genome Atlas currently consists of more than 7 billion triples and is estimated to reach 30 billion [27] whereas the LOD cloud contained already 62 billion triples since January 2014 [25].

To store, manage and query these ever increasing RDF data, many systems were developed by the research community (e.g. Jena, Sesame etc.) and by many commercial vendors (e.g. Oracle and IBM) [10]. Although, these systems have demonstrated great performance on a single node, being able to manage millions, and, in some cases, billions of triples, as the amount of the available data continues to scale, it is no longer feasible to store the entire dataset on a single node. Consequently, under the light of the big data era, the requirement for clustered RDF database systems is becoming increasingly important [6].

In principle the majority of the available clustered RDF database systems, such as SHARD [23], YARS2 [6], and Virtuoso [20] partition triples across multiple nodes using hash functions. However, hash functions require in essence contacting all nodes for query answering and when the size of the intermediate results is large, the inter-node communication cost can be prohibitively high. To face this limitation, other systems try to partition RDF datasets into clusters such that the number of queries that hit partition boundaries is minimized. However most of these systems either treat RDF as simple graphs, exploiting graph partitioning algorithms, [7] or cluster triples based on locality measures with limited semantics [17].

Although RDF datasets can be interpreted as simple graphs, besides their structural information they have also attached rich semantics which could be exploited to improve the partition algorithms and dictate a different approach. As such, in this paper, we focus on effectively partitioning RDF datasets across multiple nodes exploiting all available information, both structural and semantic. More specifically our contributions are the following:

- We present *RDFCluster*, a novel platform that accepts as input an RDF dataset and the number of the available computational nodes and generates the corresponding partitions, exploiting both the semantics of the dataset and the structure of the corresponding graph.
- We view an RDF dataset as two distinct and interconnected graphs, i.e. the schema and the instance graph. Since query formulation is usually based on the schema, we generate vertical partitions based on schema clusters. To do so we select first the most important schema nodes as centroids and assign the rest of the schema nodes to their closest centroid similar to [11]. Then individuals are instantiated under the corresponding schema nodes producing the final partitions of the dataset.
- To identify the most important nodes we reuse the notion of *relevance* based on the established measures of the *relative cardinality* and the *in/out degree centrality* of a node [30]. Then to assign the rest of the schema nodes to a centroid we define the notion of *dependence* assigning each schema node to the cluster with the maximum dependence between that node and the corresponding centroid.
- We describe the aforementioned algorithm and we present the computational complexity for computing the corresponding partitions given a dataset and the available computational nodes.
- Then, we experiment with three datasets, namely CRM$_{dig}$, LUBM and eTMO, and the corresponding queries and we show the nice properties of the produced partitions with respect to query answering, i.e. the high quality of the constructed partitions and the low storage overhead it introduces.

Our partitioning scheme can be adopted for efficient storage of RDF data reducing communication costs and enabling efficient query answering. Our approach is unique in the way that constructs data partitions, based on schema clusters, constructed combining structural information with semantics. We have to note that in this paper we are not interested in benchmarking clustered RDF systems but only on the corresponding partition algorithm.

The rest of the paper is organized as follows. Section 2 introduces the formal framework of our solution and Section 3 describes the metrics used to determine how the cluster should be formulated and the corresponding algorithm. Then, Section 4 describes the evaluation conducted and Section 5 presents related work. Finally, Section 6 concludes the paper and presents directions for future work.

2 Preliminaries & Example

In this paper, we focus on datasets expressed in RDF, as RDF is the de-facto standard for publishing and representing data on the Web. The representation of knowledge in RDF is based on triples of the form (*subject*, *predicate*, *object*). RDF datasets have attached semantics through RDFS[1], a vocabulary description language. Here, we will follow an approach similar to [12], which imposes a convenient graph-theoretic view of RDF data that is closer to the way the users perceive their datasets.

Representation of RDF data is based on three disjoint and infinite sets of resources, namely: URIs (U), literals (L) and blank nodes (B). We impose typing on resources, so we consider 3 disjoint sets of resources: classes ($C \subseteq U \cup B$), properties ($P \subseteq U$), and individuals ($I \subseteq U \cup B$). The set C includes all classes, including RDFS classes and XML datatypes (e.g., xsd:string, xsd:integer). The set P includes all properties, except *rdf:type* which connects individuals with the classes they are instantiated under. The set I includes all individuals (but not literals).

In this work, we separate between the schema and instances of an RDF dataset, represented in separate graphs (G_S, G_I respectively). The schema graph contains all classes and the properties they are associated with (via the properties' domain/range specification); note that multiple domains/ranges per property are allowed, by having the property URI be a label on the edge (via a labelling function λ) rather than the edge itself. The instance graph contains all individuals, and the instantiations of schema properties; the labelling function λ applies here as well for the same reasons. Finally, the two graphs are related via the τ_c function, which determines which class(es) each individual is instantiated under. Formally:

Definition 1 (RDF Dataset). An RDF dataset is a tuple $V = \langle G_S, G_I, \lambda, \tau_c \rangle$such that:

- G_S is a labelled directed graph $G_S = (V_S, E_S)$ such that V_S, E_S are the nodes and edges of G_S, respectively, and $V_S \subseteq C \cup L$.
- G_I is a labelled directed graph $G_I = (V_I, E_I)$ such that V_I, E_I are the nodes and edges of G_I respectively, and $V_I \subseteq I \cup L$.
- A labelling function λ: $E_S \cup E_I \longmapsto P$ that determines the property URI that each edge corresponds to (properties with multiple domains/ranges may appear in more than one edge).
- A function τ_c: $I \longmapsto 2^C$ associating each individual with the classes that it is instantiated under.

For simplicity, we forego extra requirements related to RDFS inference (subsumption, instantiation) and validity (e.g., that the source and target of property instances should

[1] https://www.w3.org/TR/rdf-schema/

be instantiated under the property's domain/range respectively), because these are not relevant for our results below and would significantly complicate our definitions. In the following, we will write $p(v_{1,} v_2)$ to denote an edge e in G_S (where $v_{1,} v_2 \in V_S$) or G_I (where $v_{1,} v_2 \in V_I$) from node v_1 to node v_2 such that $\lambda(e) = p$. In addition for brevity we will call *schema node* a node $c \in V_S$, *class node* a node $c \in C \cap V_S$ and *instance node* a node $u \in I \cap V_I$. In addition a *path* from $v_1 \in V_S$ *to* $v_2 \in V_S$, i.e. *path*(v_1, v_2), is the finite sequence of edges, which connect a sequence of nodes, starting from the node v_1 and ending in the node v_2. In this paper we will focus on class and instance nodes due to lack of space, but our approach can be easily generalized to include literals as well.

Now as an example consider the LUBM ontology[2] part shown in Fig. 1 used to describe a university domain. This example contains 20 classes and many properties. Now assume that we would like to partition the corresponding RDF dataset into three partitions revealing discriminating features for each one of them. One way to do that for example would be to identify first the three most importance schema nodes of the dataset, allocate each one of those nodes to the corresponding cluster as a centroid and finally place into the same cluster the schema nodes that *depend* on those selected nodes. The clusters generated using our approach are shown in Fig. 1. The most important schema nodes, as identified by our algorithm, are the "*Professor*", the *"Publication"* and the "*Person*" classes. These are used as centroids and the remaining schema nodes are assigned to the appropriate clusters by identifying the schema nodes that depend on those centroids. Finally the instance nodes are assigned to the class nodes that are instantiated under. In this paper we will use the term *cluster* to refer only to the schema graph and the term *partition* to refer to the entire dataset.

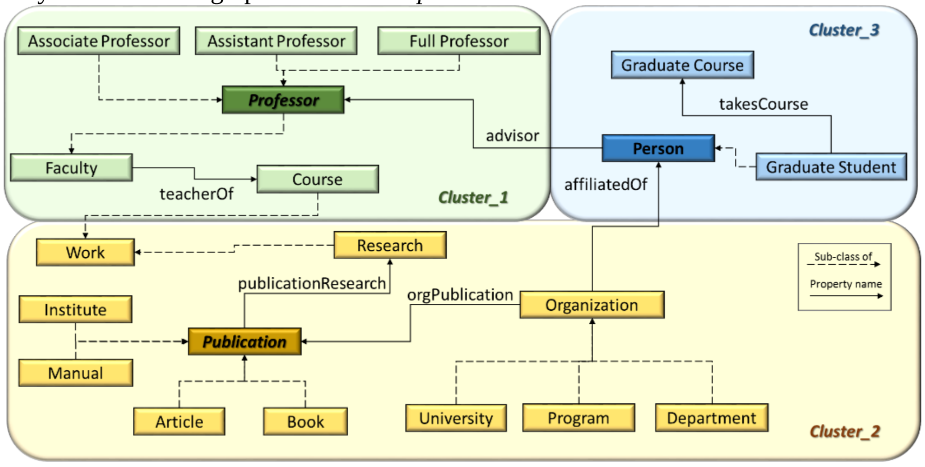

Fig. 1. An example RDF Dataset and the corresponding partitions of our algorithm

When data is partitioned across multiple machines the particular partitioning method, can have a significant impact on the amount of data that needs to be shipped over the

[2] http://swat.cse.lehigh.edu/projects/lubm/

network at query time. Ideally we would like the constructed partitions to increase the likelihood that query answers can be computed locally reducing the communication cost. In general, in distributed query processing, where multiple nodes are available, query answering proceeds by first breaking the query into pieces, all of which can be evaluated independently within individual partitions. The query pieces are then evaluated in the relevant partitions obtaining partial matches and they are joined to produce the final answer. Again, in this paper we are not interested on the technicalities of query answering but only on the aforementioned partitioning algorithm and how the careful placement of the nodes within partitions could optimize the overall number of nodes to be visited for query answering.

Assume for example the following SPARQL query involving 3 classes and 2 user-defined properties, requesting all publications of the persons belonging to an organization:

```
SELECT ?X, ?Y, ?Z WHERE{
  ?X rdf:type ub:Person.
  ?Y rdf:type ub:Organization.
  ?P rdf:type ub:Publication.
  ?Y ub:affiliatedOf ?Z .
  ?Y ub:orgPublication ?P .
}
```

If data was partitioned using a simple hash partitioning algorithm, then obviously all nodes would have to be examined. If however, the data was partitioned as shown in Fig. 1 then only two nodes would have to be contacted, as instances of the "*Organization"* and "*Publication*" classes can be found in the second partition and the instances of *"Person"* can be located at the third partition. We therefore, instead of using simple hash or graph partitioning algorithm are looking for a more advanced method, partitioning the schema into appropriate clusters, considering the semantics of the nodes and the structural information of the corresponding graph.

3 Metrics

Our clustering framework follows the K-Medoids clustering method [11]; we select the most centrally located point in a cluster as a centroid, and assign the rest of points to their closest centroids. To identify the most centrally located point in a cluster we use the notion of relevance. Then dependence is used for extracting nodes, highly relevant to the specific important nodes (centroids) connecting other nodes to the most important ones.

3.1 Identifying Centroids

Initially, the notion of centrality [30] is used to quantify how central is a class node in a specific RDF dataset. To identify the centrality of a class node *c* in a dataset *V*, we initially consider the instances it contains by calculating *its relative cardinality*. The relative cardinality $RC(p(c, c_i))$ of an edge $p(c, c_i)$, which connects the class nodes c and c_i in the schema graph, is defined as the number of the specific instance connections between the corresponding two classes divided by the total number of the connections of the instances that these two classes have. Then, we combine the data distribution with the number of the incoming/outgoing edges, aka properties, of this class. As such, the in/out-centrality (C_{in}/C_{out}) is defined as the sum of the weighted relative cardinalities of the incoming/outgoing edges:

Definition 2 (Centrality). Assume a node $c \in C \cap V_S$ in a dataset $V = \langle G_S, G_I, \lambda, \tau_c \rangle$. The *in-centrality* $C_{in}(c)$ (respectively, the *out-centrality* $C_{out}(c)$) of c is defined as the sum of the weighted relative cardinality of the incoming $p(c_i, c) \in E_s$ (respectively, outgoing $p(c, c_i) \in E_S$) edges:

$$C_{out}(c) = \sum_{p(c,c_i) \in E_s} RC(p(c, c_i)) * w_p \qquad C_{in}(c) = \sum_{p(c_i,c) \in E_s} RC(p(c_i, c)) * w_p$$

The weights in the above formula have been experimentally defined [30] and vary depending on whether edges that correspond to properties are user-defined or RDF/S, giving higher importance to user-defined ones (in our experiments we used w_p=0.8 for user-defined properties and w_p=0.2 for RDF/S ones). This is partly because user-defined properties correlate classes, each exposing the connectivity of the entire schema, in contrast to hierarchical or other kinds (e.g., *rdfs:label*) of RDF/S properties. Consider now the *"Article"* class shown in Fig. 1. Assume also that there are not any instances in the corresponding dataset. Then the relative cardinality of all nodes is initialized to a constant a=0.03. As such $C_{in}(University) = 0$ since there are no incoming edges and $C_{out}(University) = RC(_{rdf:type}) * w_{rdf:type}$ = 0.03*0.2=0.06.

Now that centrality is defined we are going to define *relevance*. The notion of relevance [30] has been proposed as adequate for quantifying the importance of a class in an RDF dataset. In particular, relevance is based on the idea that the importance of a class should describe how well the class could represent its neighborhood. Intuitively, classes with many connections with other classes in a dataset should have a higher importance than classes with fewer connections. Thus, the relevance of a class is affected by the centrality of the class itself, as well as by the centrality of its neighboring classes. Moreover, since the version might contain huge amounts of data, the actual data instances of the class should also be considered when trying to estimate its importance, namely relevance. Formally, relevance is defined as follows:

Definition 3 (Relevance). Assume a node $c \in C \cap V_S$ in a dataset $V = \langle G_S, G_I, \lambda, \tau_c \rangle$. Assume also that $c_1, \ldots, c_n \in E_S$ are the incoming edges of c ($p(c_i, c) \in E_S$) and $c'_1, \ldots, c'_k \in E_S$ are the outgoing edges of c ($p(c, c'_i) \in E_S$). Then the relevance of c, i.e. *Relevance*(c), is the following:

$$Relevance(c) = \frac{C_{in}(c) * n + C_{out}(c) * k}{\sum_{j=1}^{k}\left(C_{out}(c_j)\right) + \sum_{i=1}^{n}\left(C_{in}(c_i)\right)}$$

The aforementioned metric identifies class nodes being able to represent an entire area and as a consequence those nodes can be used as the centroids of the corresponding graph. In our example, shown in Fig. 1, *Relevance*(*University*) = C_{in}(*University*) + C_{out}(*University*) / C_{out}(*Organization*)+0 = 0+0.06/0.048 = 1.25.

3.2 Assigning nodes to centroids

Having a method to identify the most important nodes (centroids) in an RDF dataset we are now interested on identifying to which cluster the remaining nodes should be assigned to. Our first idea to this direction comes from the classical information theory; that infrequent words are more informative that frequent ones. The idea is also widely used in the field of instance matching [24]. The basic hypothesis here is that the greater the influence of a property on identifying a corresponding instance the less times it is repeated. According to this idea, we try to initially identify the dependence between two classes based on their data instances.

In our running example, the node "*Person*" has a high relevance in the graph and as a consequence a great probability to be used as a centroid. Assume also two nodes "*SSN"* and *"Work"* directly connected it. Although an instance of "*Person*" can have only one social security number, many persons can be employed by the same employer and as such a person cannot be characterized by his work. As a consequence, the dependence between "*Person*" and "*SSN*" is higher than the dependence between "*Person*" and "*Telephone*". Based on this observation, we define the measurement of *cardinality closeness* of two adjacent schema nodes.

Definition 4 (Cardinality Closeness). Let c_k,c_s be two adjacent schema nodes and u_i, $u_j \in G_I$ such that $\tau_c(u_i)= c_k$ and $\tau_c(u_j)= c_s$. The *cardinality closeness* of $p(c_k, c_s)$, namely the *CC*($p(c_k, c_s)$), is the following:

$$CC(p(c_k,c_s)) = \frac{1}{|c|} + \frac{DistinctValues(p(u_i,u_j))}{Instances(p(u_i,u_j))}$$

where |*c*|, $c \in C \cap V_S$ the number of nodes in the schema graph, *DistinctValues*($p(u_i, u_j)$) the number of distinct $p(u_i, u_j)$ and *Instances*($p(u_i, u_j)$) the number of $p(u_i, u_j)$. When there are no instances *Instances*($p(u_i, u_j)$)=1 and *DistinctValues*($p(u_i, u_j)$)=0.

The constant 1/|*c*| is added in order to have a minimum value for the CC in case of no available instances. Having defined the *cardinality closeness* of two adjacent schema nodes we proceed further to identify their dependence. As such we identify the dependence between two classes as a combination of their cardinality closeness, the relevance of the classes and the number of edges between these two classes:

Definition 5 (Dependence of two schema nodes). The *dependence* of two schema nodes c_s and c_e, *i.e. the Dependence*(c_s, c_e), *is given by the following formula*

$$Dependence(c_s,c_e) = \frac{1}{|path(c_s,c_e)|^2} * \left(Relevance(c_s) - \sum_{i=s+1}^{e} \frac{Relevance(c_i)}{CC(p(c_{i-1},c_i))} \right)$$

Obviously as we move away from a node, the dependence becomes smaller by calculating the differences of relevance across a selected path in the graph. We penalize additionally dependence dividing by the distance of the two nodes. The highest the dependence of a path, the more appropriate is the first node to describe the final node of the path. Also note that the *Dependence(c_s, c_e)* is different than *Dependence(c_e, c_s)*. For example, *Dependence(Publication, Book)* ≥ *Dependence(Book, Publication).* This is happening, since the dependence of a more relevant node toward a less relevant node is higher than the other way around, although, they share the same cardinality closeness.

3.3 The Clustering algorithm

Having defined both the relevance for identifying the most important nodes and the dependence of two schema nodes we are now ready to define the semantic partitioning problem:

Definition 6 (Semantic Partitioning Problem). Given an RDF Dataset $V = \langle G_S, G_I, \lambda, \tau_c \rangle$, partition *V* into k subsets $V_1, V_2, \ldots, V_k$ such that:

1. $V = \bigcup_{i=1}^{k} V_i$
2. Let $top_k = \{c_1, \ldots, c_k\}$ be the *k* schema nodes with the highest relevance in *V*. Then $c_1 \in V_1, \ldots, c_k \in V_k$
3. Let d_j be a schema node and $d_j \notin top_k$. Then $Dependence(d_j, c_p) = \max_{0 \le x \le k} \mathrm{Dependence}(d_j, c_x) \rightarrow \exists\, d_j$ in V_p , $(1 \le p \le k)$
4. $\forall u \in G_I$, such that $\tau_c(u) \in G_S$ and $\tau_c(u) \in V_j \rightarrow \exists\, u$ in V_j

The first requirement says that we should be able to recreate *V* by taking the union of all V_i $(1 \le i \le k)$. The second one that each cluster should be based on one of the nodes with the top *k* relevance (the top_k set) as a centroid, and the third that each node that does not belong to the top_k should appear at least in the cluster with the maximum dependence between the specific node and the corresponding centroid. Note that a node can appear in multiple clusters. The idea originates from social networks where an individual can simultaneously belong to several communities (family, work etc.), similarly an RDF resource might belong to more than one clusters. As such, in order to include a schema node in the V_p cluster $(1 \le p \le k)$ we are looking for the path maximizing the *Dependence*. In the selected path however there might exist nodes not directly assigned to V_p. We include those nodes in the cluster as well since they have also high dependence to the centroid. Finally all instances are replicated under the corresponding schema nodes.

The corresponding algorithm is shown in Fig. 2. The algorithm gets as input an RDF dataset and the number of computational nodes (*k*) and partitions the dataset into *k* partitions. Bellow we explain in more detail each of the steps of the algorithm.

The algorithm starts by calculating the relevance of all schema nodes (lines 2-3). More specifically for each node in G_S we calculate the corresponding relevance according to Definition 3. Having calculated the relevance of each node we would like to get the *k* most important ones to be used as centroids in our clusters. Those are selected (line 4) and then assigned to the corresponding cluster (lines 5-6).

Then the algorithm examines the remaining schema nodes to determine to which cluster they should be placed at. For each node we calculate the dependence between the selected node and all centroids (line 7). We select to place the node in the cluster with the maximum dependence between the aforementioned node and the *k* centroids (line 8). However we are not only interested in placing the selected node to the identified cluster but we place the whole path and specifically the nodes contained in the path, which connects the selected node with the appropriate centroid *(path_with_max_depedence*), maximizing the dependence of the selected node in that cluster (line 9) as well. Next, we add to each cluster the corresponding instance nodes to the schema nodes they are instantiated under. Finally, we return the partitions to the user. The correctness of the algorithm is immediately proved by construction.

Algorithm 1: *RDFCluster*(*V*, *k*)
Input: An RDF dataset $V = \langle G_S, G_I, \lambda, \tau_c \rangle$, k the number of the available nodes.
Output: A set *S* of *k* partitions $S=\{V_1,\dots, V_k\}$.

```
1.
2.  for each node $c_i \in G_S$
3.          $r_i := calculate\_relevance(V, c_i)$
4.  $top_k := select\_top\_nodes(r, k)$
5.  for each node $c_i \in top_k.$
6.          $V_i = V_i \cup c_i$
7.  for each node $c_i \notin top_k$
8.          $j = find\_cluster(c_i, \mathrm{top}_k)$
9.          $V_j = V_j \cup path\_with\_max\_dependence(c_i, V)$
10. for each node $c_i$ in $V_j$
11.     $V_j = V_j \cup Instances(c_i)$
12. Return $S=\{V_1,\dots, V_k\}$
```

Fig. 2. The RDFCluster algorithm

To identify the complexity of the algorithm we should first identify the complexity of its various components. Assume |V| the number of nodes, |E| the number of edges and |I| the number of instances. For identifying the relative cardinality of the edges we should visit all instances and edges once. Then for calculating the schema node centralities we should visit each node once whereas for calculating the relevance of each node we should visit twice all nodes O(|I|+|E|+2|V|). Then we have to sort all nodes according to their relevance and select the top *k* ones O(|V|log|V|). To calculate the dependence of each node we should visit each node once per selected node O(k|V|), whereas to identify the path maximizing the dependence we use the weighted Dijkstra algorithm with cost $O(|V|^2)$. Finally we should check once all instances for identifying the clusters to be assigned O(|I|). As such the time complexity of the algorithm is polynomial $O(|I|+|E|+2|V|) + O(|V|\log|V|) + O(k|V|) + O(|V|^2) \leq O(|V|^2)$.

4 Evaluation

To evaluate our approach and the corresponding algorithm we used three RDF datasets:

CRM$_{\mathbf{dig}}$[3]. CRM$_{\text{dig}}$ is an ontology to encode metadata about the steps and methods of production ("provenance") of digitization products and synthetic digital representations created by various technologies. The ontology contains 126 classes and 435 properties. For our experiments we used 900 real instances from the 3D-SYSTEK[4] project. In addition we used 9 template queries published in [28] with an average size of 6 triple patterns.

LUBM. The Lehigh University Benchmark (LUBM) is a widely used benchmark for evaluating semantic web repositories. It contains 43 classes, and 32 properties modeling information about universities and is accompanied by a synthetic data generator. For our tests we used the default 1555 instances coming from a real dataset. The benchmark provides 14 test queries that we used in our experiments with an average size of 4 triple patterns.

eTMO. This ontology has been defined in the context of MyHealthAvatar[5] EU project [16] and is used to model various information within the e-health domain. It is consisted of 335 classes and 67 properties and it is published with 7848 real instances coming from the MyHealthAvatar EU project. For querying we used 8 template queries specified within the project for retrieving relevant information, with an average size of 17 triple patterns per query.

Each dataset was split into 2, 5 and 10 partitions and we used all queries available for query answering. For a fixed dataset, increasing the number of partitions is likely to increase the number of nodes required for answering queries as the data becomes more fragmented. However, it increases the number of queries that can be answered independently in parallel reducing the computation task for a single node. As we have already mentioned, our task is not to measure end-to-end query answering times involving multiple systems but to evaluate the quality of the constructed partitions with respect to the query answering

As such, for each V_1, …, V_k (k=2, 5, 10) we measure the following characteristics: i) The quality of constructed partitioning algorithms, i.e. the percentage of the test queries that can be answered only by a single partition, (ii) the number of partitions that are needed in order to answer each query and (iii) the space overhead that our algorithm introduces in both schema nodes and the dataset.

We compare our approach with a) subject-based hash partitioning similar to YARS2 [6] and Trinity.RDF [34] called *Hashing*, and b) METIS used by [17] [7] for clustering RDF Datasets. *Hashing* is distributing triples in partitions by applying a hash function to the subject of the triple in order to guarantee that star queries can be evaluated locally. METIS [13] on the other hand calculates n disjoint sets of nodes such that all sets are of similar sizes and the number of edges in connecting nodes in

[3] http://www.ics.forth.gr/isl/index_main.php?l=e&c=656
[4] http://www.ics.forth.gr/isl/3D-SYSTEK/
[5] http://www.myhealthavatar.eu/

distinct sets is minimized. In this work we focus only on the partitioning schemes of the aforementioned works. All datasets and queries used in our experiments along with the detailed results can be found online[6].

4.1 Quality

We perceive the quality of a partitioning algorithm with respect to query answering as the percentage of queries that can be answered by a single computational node without requiring to visit additional nodes to provide answers to the user. The results for all queries in our three datasets for the three algorithms in 2, 5 and 10 partitions are shown in Table 1.

We can easily identify that RDFCluster is better in almost all the cases showing the high quality of the produced partitions with respect to query answering. The only case that METIS is better than RDFCluster is in LUBM when we have 5 partitions where one more query can be answered by a single one. However, for LUBM, even in 5 partitions as we shall see in the sequel (Section 4.2) our algorithm requires less nodes to be visited on average for answering the benchmark queries. In addition we expect that as the number of partitions increases the average number of queries that can be answered by an individual partition decreases as the data are distributed to more nodes. Our expectations are confirmed by our results.

In addition as expected, smaller queries (LUBM with an average of 4 triple patterns per query and CRM_{dig} with an average of 6 triple patterns per query) show a greater likelihood to be answered by a single node than queries with more triple patterns such as eTMO with an average of 17 triple patterns per query.

Table 1. The quality of the three clustering algorithms Hashing (H), Metis (M) and RDFCluster (RC) in 2, 5 and 10 partitions.

Partitions	**CRM_{dig}**			**LUBM**			**eTMO**		
	H	**M**	**RC**	**H**	**M**	**RC**	**H**	**M**	**RC**
2	22%	22%	100%	14%	14%	36%	0%	0%	88%
5	0%	0%	44%	14%	21%	14%	0%	0%	13%
10	0%	0%	22%	7%	7%	14%	0%	0%	13%

4.2 Number of clusters required for answering a query

Besides evaluating the quality of our algorithm, another interesting dimension is to evaluate how much work is required for answering the queries in each case in terms of the nodes required to be visited. The nodes to be visited give us an intuition about how many joins will be required to construct the final answer that will be returned to the user. This is critical because, in order to ensure the completeness of query answers, all partial matches in all partition elements must be computed and joined together.

[6] http://www.ics.forth.gr/~kondylak/ISWC2016_Evaluation.zip

The results are shown in Fig. 3 where we can see that in all cases RDFCluster requires on average less nodes to be visited for query answering, showing again the nice properties of our algorithm. Note that even for large queries (eTMO with an average of 17 triple patterns) our algorithm requires only three partitions to be visited on average for query answering and this applies even in the case of 10 partitions.

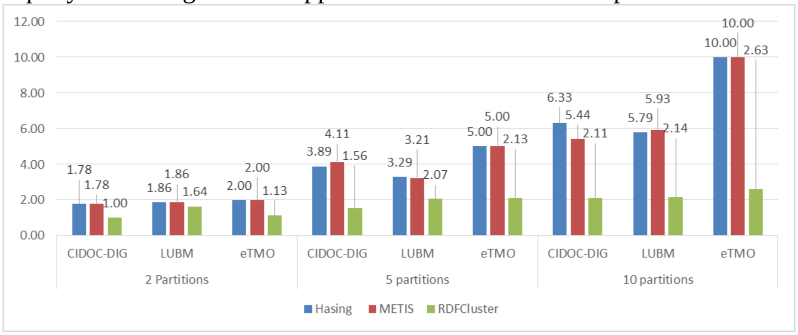

Fig. 3. The number of nodes required for answering the benchmark queries.

4.3 Storage overhead

The storage overhead provides us with an indication of how much space is needed for our approach compared to the space required for storing all datasets in a single node. Since Hashing and METIS algorithms construct non-overlapping clusters they have no storage overhead. However for simple variations of hash allowing duplication the overhead can be really high (e.g. 2-hop duplication can lead to an overhead up to 430% [7]). In our case, since we allow a class node and the corresponding instances to be replicated in multiple nodes we expect as the number of clusters increases to increase the storage overhead as well.

Table 2. Schema Nodes Overhead as the number of clusters increases

Clusters	CRM$_{dig}$	LUBM	eTMO
2	1.55%	3.33%	0.65%
5	1.55%	8.33%	4.90%
10	6.20%	15.00%	7.19%

Table 3. Total Storage Overhead as the number of clusters increases

Clusters	CRM$_{dig}$	LUBM	eTMO
2	0.10%	0.12%	0.04%
5	0.10%	0.89%	0.29%
10	16.73%	1.13%	2.78%

To identify and understand the overhead introduced by our algorithm first we focus only on the schema graph and identify the overhead introduced there. This is shown in Table 2 calculating the percentage $|G_{SV1}|$+ …+ $|G_{SVK}|$ - $|G_S|$ / $|G_S|$. As shown the overhead is minimal introducing at most 15.00% additional schema nodes for LUBM whereas for eTMO and CRM$_{dig}$ is only 7.19% and 6.20% respectively.

The impact of these additional schema nodes to the overhead of the entire dataset is shown in Table 3. The table shows the total storage overhead introduced by our algorithm, i.e. the percentage |V1|+ …+ |Vk | - |V| / |V|. As shown, the total storage overhead introduced from our algorithm is at most 16.73% for CRMdig and for the majority of the cases less than 1%. Another interesting observation is that in almost all the cases the schema nodes overhead is greater than the corresponding total storage overhead showing that our algorithm succeeds in replicating only nodes with small additional overhead that however significantly improve query answering as shown in previous sections.

Overall, as the experiments show although our algorithm chooses to sacrifice equal data distribution on the nodes to achieve a better performance with respect to query answering the imposed overhead is really low reaching at most 16.73% overhead on our test cases.

5 Related Work

Graph clustering has received much attention over the latest years [35], aiming to partition large graphs into several densely connected components, with many application such as community detection in social networks, identification of interactions in protein interaction networks etc. The problem proved to be an NP-complete problem [5]. Typical algorithms of this class include local search based solutions (such as KL [15] and FM [4]), which swap heuristically selected pairs of nodes, simulated annealing [8], and genetic algorithms [3]. Algorithms in this category focus on the topological structure of a graph so that each partition achieves a cohesive internal structure and there are approaches based on normalized-cut [26], modularity [20], structural density [33], attribute similarity [29] or combinations between those [35]. To scale up to graphs with millions of nodes, multi-level partitioning solutions, such as Metis [13], Chaco [9], and Scotch [22], and variations over these have been proposed.

To this direction, several approaches try to represent RDF datasets as graphs and exploit variations of the aforementioned data for data partitioning. For example, Wang et al. [31] focus on providing semantic-aware highly parallelized graph partitioning algorithms for generic-purpose distributed memory systems whereas Huang et al. [7] apply graph partitioning over the Hadoop MapReduce framework trying to reduce as much as possible the communication costs. Our approach however, does not focus only on the structural part of the graph for partitioning the RDF datasets but considers in addition semantic information (such as the number of instances, the distinct instance values, assigns different weights according to the type of the properties) with the same target however, i.e. to reduce as much as possible the communication costs among partitions when these partitions are used for query answering.

Other clustered RDF database systems, such as SHARD [23], YARS2 [6], and Virtuoso [20] partition triples across multiple nodes using hash functions. However, portioning data using hashing requires a lot of communication overhead for query answering since in essence all nodes have to be contacted. The same problem appear in other works that adopt vertical [2] or horizontal partitioning schemes based on triples

[18] ignoring however the correlation between triples, leading to a large number of join operators among the compute nodes. Other algorithms, but with the same problem use hybrid hierarchical clustering [19] combining an affinity propagation clustering algorithm and the k-Means clustering algorithms. To overcome that problem Lee et al. [17] proposed to by use locality sensitive hashing schemes. Although this approach moves to the same direction with ours, trying to exploit semantics, the adopted solution is limited to only the fact that triples are anchored at the same subject or object node. In addition according to our experiments our solution outperforms similar approaches.

Finally there are approaches that try to monitor the execution of SPARQL queries [1] or assume that query patterns are already available [32] and keep track of records that are co-accessed and physically cluster them using locality sensitive hashing schemes. Our approach uses a similar "profiling" mechanism but instead of focusing on queries, we focus on profiling "data" identifying and combining the knowledge of the instance distribution with structure and semantics. A more thorough overview of the different partition schemes for RDF datasets can be found on [10].

6 Conclusions and Future Work

In this paper we present a novel method that gets as input and RDF dataset and the number of available computational nodes and returns a set of partitions to be stored on the aforementioned nodes. To select the centroids for the each cluster initially our algorithm selects the most important nodes based on the notion of relevance. Then to assign the remaining nodes to a cluster we use the notion of dependence eventually assigning the remaining schema nodes to the cluster maximizing the dependence with the corresponding centroid. Having constructed the appropriate "schema clusters" we place next the instances on the corresponding classes they belong to. Our algorithm exploits both structural and semantic information in order to both select the most important nodes and then to assign the remaining nodes to the proper clusters. In addition, since both our constructed clusters and user queries are based on schema information we argue that this partitioning scheme will have a beneficial impact on query evaluation limiting significantly the nodes that should be visited to answer frequent queries.

The quality of our partitioning scheme is verified by our experiments. We use three RDF Datasets, namely CRM_{dig}, LUBM and eTMO with their corresponding template queries and we show that the clusters produced significantly limit the number of clusters to be contacted for query answering. Obviously, as the number of clusters increases, eventually the number of nodes required for query answering increases as well, leading to trade-offs among load-balancing and the number of nodes to be used. However, as shown, our algorithm achieves better performance than existing systems with respect to query answering, requiring at most 3 nodes for our template queries even when the dataset is partitioned in 10 nodes. In addition, although in order to these results we allow replication, we show that the impact is minimal imposing at most at most 16.73% total storage overhead.

As future work we intend to explore how our algorithm shall be adapted when no schema is available in an RDF dataset; it is true that RDF datasets do not have always a predefined schema which limits their use to express queries or to understand their content. To this direction approaches are starting to emerge discovering the types of the data using clustering algorithms [14]. Furthermore, we plan to deploy our clustering algorithm in a real clustered environment and to measure the actual improvement on query execution times, comparing our solution with other competitive approaches. In addition our clustering method does not considers limiting the number of nodes that are included in each cluster. However, an idea would be to try to limit the nodes assigned to each cluster trying in parallel to maximize the total dependence of the selected nodes. The problem is well-known to be NP-complete, requires complex variation algorithms over Steiner-Tree problem and we have already started to explore interesting approximations [21]. Obviously as the size and complexity of data increases, partitioning schemes are becoming more and more important and several challenges remain to be investigated in the near future.

Acknowledgements

This research is implemented through IKY scholarships programme and co-financed by the European Union and Greek national funds through the action entitled ”Reinforcement of Postdoctoral Researchers”, in the framework of the Operational Programme ”Human Resources Development Program, Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) 2014 − 2020

References

1. G. Aluç, M.T. Özsu, and K. Daudjee. Clustering RDF Databases Using Tunable-LSH. *CoRR abs/1504.02523*, 2015.
2. S. Álvarez-García, N.R. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto and G. Navarro. Compressed vertical partitioning for efficient RDF management. *Knowl. Inf. Syst*. 44(2): 439-474, 2015.
3. T.N. Bui and B.R. Moon. Genetic algorithm and graph partitioning, *IEEE Trans. Computers*, 45(7), pp. 841– 855, 1996.
4. C. M. Fiduccia et al., A linear-time heuristic for improving network partitions, DAC, 1982.
5. M.R. Garey, D.S Johnson, and L.J. Stockmeyer. Some Simplified NP-Complete Problems. *STOC*, 1974.
6. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. *ISWC/ASWC*, 2007.
7. J. Huang, D.J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB* 4(11):1123-1134, 2011.
8. D.S. Johnson, C.R. Aragon, L.A. McGeoch, et al. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning, *Oper. Res.*, 37:865–892, 1989.
9. B. Hendrickson and R. Leland. The chaco user’s guid, version 2.0, *Technical Report SAND94-2692*, Sandia National Laboratories, 1995.
10. Z. Kaoudi, I. Manolescu. RDF in the clouds: a survey. *VLDB J*. 24(1): 67-91, 2015.

11. L. Kaufman and P. J. Rousseeuw. Clustering by means of medoids. *Statistical Data Analysis based on the L1 Norm*, 1987.
12. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis and M. Scholl. RQL: a declarative query language for RDF. *WWW*, 2002.
13. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal on Scientific Computing*, 20 (1): 359, 1999.
14. K. Kellou-Menouer and Z. Kedad. A Clustering Based Approach for Type Discovery in RDF Data Sources. *EGC*, 2015.
15. B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs, *Bell Systems Journal*, 49:291–307, 2013.
16. H. Kondylakis, M. Spanakis, S. Sfakianakis, et al. Digital Patient: Personalized and Translational Data Management through the MyHealthAvatar EU Project, EMBC, 2015.
17. K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14): 1894-1905, 2013.
18. K. Lee, L. Liu, Y. Tang, Q. Zhang and Y. Zhou. Efficient and Customizable Data Partitioning Framework for Distributed Big RDF Data Processing in the Cloud, *CLOUD*, 2013.
19. Y. Leng, Z. Chen, F. Zhong and H. Zhong. BRDPHHC: A Balance RDF Data Partitioning Algorithm Based on Hybrid Hierarchical Clustering. *HPCC/CSS/ICESS*, 2015.
20. M.E.J. Newman and M. Girvan. Finding and evaluating community structure in networks. *In Phys. Rev. E*, 69, 026113, 2004.
21. A. Pappas, G. Troullinou, G. Roussakis, et al., Exploring Importance Measures for Summarizing RDF/S KBs, ESWC, 2017.
22. F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, HPCN, 1996.
23. K. Rohloff and R.E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. *PSI EtA*, 4, 2010.
24. H. Seddiqui, R. P. D. Nath and M. Aono. An Efficient Metric of Automatic Weight Generation for Properties in Instance Matching Technique, *JWS*, 6(1):1-17, 2015.
25. M. Schmachtenberg, C. Bizer and H. Paulheim, State of the LOD Cloud: http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/, Accessed: 2016-04-30.
26. J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.
27. The Cancer Genome Atlas project. http://cancergenome.nih.gov/. Accessed: 2016-04-30.
28. M. Theodoridou, Y. Tzitzikas, M. Doerr, et al. Modeling and querying provenance by extending CIDOC CRM. Distributed and Parallel Databases 27(2): 169-210, 2010.
29. Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. *SIGMOD*, 2008.
30. G. Troullinou, H. Kondylakis, E. Daskalaki and D. Plexousakis. RDF Digest: Efficient Summarization of RDF/S KBs, *ESWC*, 2015.
31. L. Wang, Y. Xiao, B. Shao, H. Wang. How to partition a billion-node graph. *ICDE*, 2014.
32. X. Wang, T. Yang, J. Chen, L. He, X. Du. RDF partitioning for scalable SPARQL query processing. *Frontiers of Computer Science*, 9(6): 919-933, 2015.
33. X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. Scan: a structural clustering algorithm for networks. *KDD*, 2007.
34. K. Zeng, J. Yang, H. Wang, B. Shao, Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. PVLDB 6(4): 265-276, 2013.
35. Y. Zhou, H. Cheng and J.X. Yu. Graph Clustering Based on Structural/Attribute Similarities. *PVLDB* 2(1): 718-729, 2009.