A logic dimension on RDF partitioning, technical report

Jorge Galicia[1], Amin Mesmoudi[1,2], and Ladjel Bellatreche[1]

[1] LIAS/ISAE-ENSMA, France
{jorge.galicia,bellatreche}@ensma.fr
[2] Université de Poitiers, France
amin.mesmoudi@univ-poitiers.fr

Abstract. In the last years, scalable RDF processing systems distributing the data over a set of nodes to improve the performance have gained momentum. The triple is used as a distribution unit in these systems contrary to the relational model that defines the higher-level entities (tables) first and then partitions using tables' subsets. We believe that gathering the triples storing facts of the same logical entities contributes not only to avoid scanning irrelevant triples but also to create RDF partitions with an actual logical meaning. In this study, we give the formal definition and detail the algorithm to gather the logical entities, which we name segments, used as distribution units for RDF datasets. The logical entities proposed, harmonize with the notion of partitions by instances (horizontal) and by attributes (vertical) in the relational model. We propose allocation strategies for these segments, considering the case when replication is available and in which both fragments by instances and by attributes are considered. We finally propose a declarative partitioning definition language for RDF declaring the higher-level entities and partitions.

Keywords: RDF · Partitioning · Distributed Computing.

1 Introduction

The Resource Description Framework (RDF) has been widely accepted as the standard model for data interchange on the Web. It facilitates the exchange between applications providing a common framework to express information about resources on the Web. Web resources are described using facts represented as triples of the form $< subject, property, object >$. Formerly, RDF intended to be only a machine-readable exchange standard. Moreover, collections of RDF triples (known as Knowledge Bases) are extensive sources of information popularly queried and aggregated (using SPARQL). The size of the knowledge bases has considerably augmented in the last years. Likewise, the development of scalable RDF processing systems that can cope with several billions of triples.

Parallel and distributed solutions distribute RDF datasets over processing and storing nodes for scaling purposes. As it is the case in the relational model,

the fragmentation and allocation of an RDF graph is not trivial. Contrarily to relational databases, in RDF there is not a conceptual design stage being that RDF systems store the data at a higher granularity. The notion of higher-level entities (i.e. tables) storing its instances under the same structure is not present in RDF. An RDF triple (subject, predicate, object) represents a fact describing the resource in the subject position. This characteristic gives the RDF model flexibility and allows to represent with simple statements information about a resource. However, knowledge bases are often collections of scattered facts very hard to distribute and query efficiently.

Most of partitioning approaches for distributed RDF systems do not consider the identification of higher-level entities before partitioning. Partitioning approaches apply, for example, hashing or graph-partitioning strategies to cluster and distribute the data. The query execution on these systems relies in the exploration of very dense indexes (e.g. SPO, OPS, POS) or in massive processing frameworks (e.g. MapReduce) to find solutions to SPARQL queries. Additionally, the execution engine is dependent on the partitioning strategy used to distribute the data. Creating groups of triples describing the same high-level entity could help to prune invalid intermediate results similarly to what is done in the relational model in which the FROM clause indicates the only relations to be scanned.

There have been some efforts towards the addition of class hierarchies to RDF (e.g. RDFS and OWL). Furthermore, not all the entities in a knowledge base are annotated with this metadata and when a SPARQL query pattern does not include an annotated class, the system is unable to scan only the pertinent triples. We believe that the identification of higher-level entities might be useful to: i) avoid scanning not relevant triples, ii) declare more specific customized optimization strategies (like indexes), iii) avoid data skewness in some nodes in the allocation, and iv) use these entities in a declarative language to create partitions with a logical meaning.

Our work introduces a logical dimension to the partitioning process of RDF graphs. We define two types of high-level entities, which we name forward and backward segments. The forward and backward segments harmonize with the notion of partitions by instances (horizontal) and by attributes (vertical) in the relational model. We detail an identification algorithm of both types of entities. Next, we use these entities as allocation fragments. We propose allocation strategies for both segments, considering the case in which both partitioning schemas are considered. We finally propose a declarative partitioning definition language for RDF declaring the higher-level entities and partitions.

The contributions of this paper are:

1. The formalization of a logical dimension generalizing RDF partitions identifying high-level entities.
2. An analysis of allocation methods integrating both partitioning by attributes and by instances in RDF systems.
3. The proposition of a declarative definition language to declare and allocate the high-level entities (segments) in an RDF graph.

The rest of the paper is organized as follows. We start in Section 2 by stating the motivation of our work. Then, Section 3 gives an overview of the creation and allocation of forward and backward segments by the means of a motivating example. In Section 4, we formalize our solutions and define the segment allocation problem. Section 5 discusses related works. Finally, in Section 6 we conclude and give insights on our future work.

2 Motivation

The partitioning problem in relational databases is considered a mature problem. There are clearly two alternatives to partition the logical entities mapped to tables: i) by its attributes (vertical partitioning) or ii) by its instances (horizontal partitioning). The relational model has largely explored both alternatives. Even if the partitioning step is part of the physical design of the database, some partitions have a logical representation. For example, a table $T =$`Airplane(ID, Model, Length, Constructor)` is partitioned in $T_1 = \sigma_{Length<50}(T)$ and $T_2 = \sigma_{Length\geq 50}(T)$. This representation is useful to declare a partition applying the same declaration language used for the tables. For example:

```
CREATE TABLE AIRPLANE (ID long, MODEL varchar, LENGTH varchar,
                       CONSTRUCTOR varchar)
PARTITION BY RANGE(LENGTH)
(
PARTITION T1 VALUES LESS THAN (50),
PARTITION T2 VALUES LESS THAN (MAXVALUE)
);
```

Also, a logic representation of a partition contributes to create other optimization structures like indexes. An index could be created at different, or even the same, attributes but at different granularities avoiding the definition of useless and redundant structures. Finally, the logic representation of a partition could be used by the query optimizer at run-time to avoid unnecessary partition scans.

Meanwhile, data partitioning in the RDF model remains very close to the physical structure used to store the data. Most of RDF processing systems adopted the relational model to physically represent RDF triples. For example, some systems store an RDF graph using a big table of three columns (subject, predicate and object). A partitioning strategy for for this table could be to apply a hashing function to the values of one of the columns. There is no logical meaning of the partitions induced by this method. Besides, the partitioning strategy in several RDF systems is not customizable. Not only optimization structures like indexes cannot be implemented by the user but the execution engine depends on a specific partitioning strategy. We seek to provide a generic logical representation for RDF partitions. Our model contributes to generalize the partitioning methods proposed by several RDF systems. The logical representation of RDF partitions is similar to the representations of the relational model in which the entities are partitioned by instances, by its attributes, or by both.

We propose allocation strategies for the RDF partitions represented as logical entities. Finally, we enunciate a declarative partition language (PDL) to create customized RDF partitions. The next section uses a motivation example to get an overview of the model.

3 Logic partitioning overview: motivating example

Let us consider the RDF graph G shown in Figure 1. We simplified the N-Triples syntax to encode the dataset by adding a ":" representing the complete IRI (Internationalized Resource Identifier).

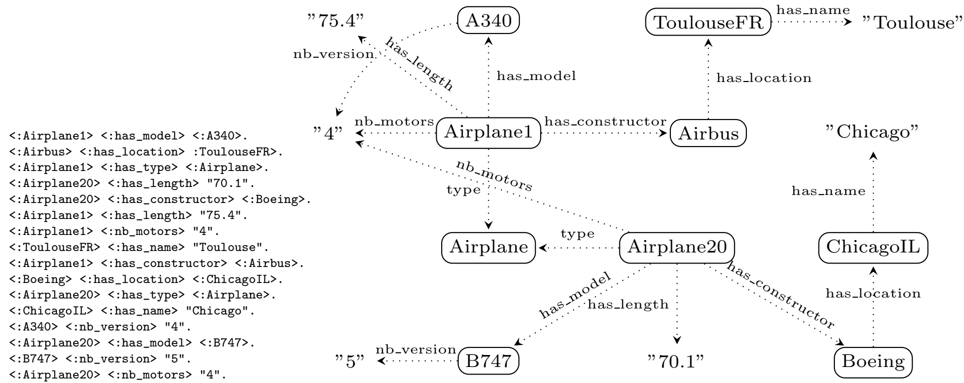


Fig. 1: RDF graph G

An RDF triple encodes a statement, a simple logical expression or claim about the world. For example, the triple $< Airplane1, has_model, A340 >$ states the fact that the *Airplane1*'s model is A340. In contrast to the conceptual design step in the relational model, in RDF the high-level entities (e.g. an airplane) are not defined at the creation moment of the database. This characteristic gives the RDF model flexibility but at the price of data dispersion. Raw RDF files store lots of disperse triples (sometimes billions) in which the facts of the same high-level entity are scattered through the batch of data. Querying raw RDF files is therefore a challenging task.

Some efforts have been done towards adding semantics as metadata to RDF. RDFS (RDF Schema[3]) and OWL[4] are some of the proposed standards. RDFS allows, for instance, the definition of classes and class hierarchies. The predicate `rdf:type` is used to specify that an individual resource belongs to a certain class. This is the case of both `Airplane1` and `Airplane20` in the example. Furthermore when the predicate `rdf:type` is not present in a SPARQL query, it

[3] https://www.w3.org/TR/rdf-schema/
[4] https://www.w3.org/TR/2012/REC-owl2-overview-20121211/

is impossible to determine the high-level entity(ies) concerned by it. This is the case of the queries shown on Figure 2. The query on Figure 2a for example, implicitly concerns the entity `Airplane` but since the predicate `rdf:type` is not explicitly stated in the query, the optimizer may scan many irrelevant entities. Additionally, the use of the `rdf:type` is not mandatory for all resources. Our approach gathers the triples belonging to the same high-level entity (e.g. Airplane, Constructor) based on its entire set of predicates and not only on the predicate `rdf:type`. We denote as an entity instance, to the cluster of a node and its direct outgoing edges and nodes. For example, `Airplane1` and its outgoing edges and nodes correspond to an *instance* of the class Airplane. We explain in detail this procedure in the next section.

```
SELECT ?y ?z
WHERE {
?x :has_model :B747 .
?x :has_length ?y  .
?x :has_constructor ?z  . }
```

(a) Example query 1

```
SELECT COUNT(?x)
WHERE {
?x :has_motors "4"  . }
```

(b) Example query 2

```
SELECT ?y ?z
WHERE {
?x :has_model :A340 .     #1
?x :has_length ?y .       #2
?x :has_constructor ?w .  #3
?w :has_location ?z .     #4
?z :has_name "Toulouse" . #5 }
```

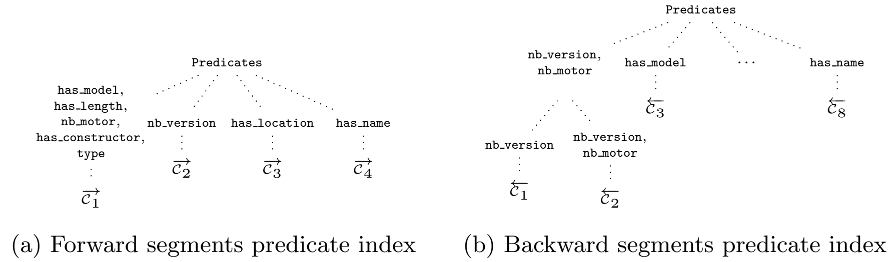
(c) Example query 3

Fig. 2: Example SPARQL queries

3.1 Creation of RDF groups of instances

To gather the triples of the same high-level entity we group in the first place the triples by its subjects. We obtain 8 groups of triples (`Airplane1, Airplane20, B747, A340, Airbus, Boeing, ToulouseFR` and `ChicagoIL`). We name each of these groups of triples a *forward entity* $\overrightarrow{E}$ representing instances of higher level entities (e.g. an Airplane, Constructor, Airplane Model). We observe that two instances of the same high-level entity share the same (or almost the same) set of predicates. In the example, the forward entities `Airplane1` and `Airplane20` share the same predicates `has_model, has_length, has_constructor, nb_motsors` and `type`. `Airplane1` and `Airplane20` belong to the same group that we named the i-th *forward segment* $\overrightarrow{\mathcal{C}_i}$ of G. A forward segment represent a high-level entity on the graph G. In Section **??** we prove that the set of forward segments is a partition set of the RDF dataset. The set of forward segments is represented as $\overrightarrow{C} = \{\mathcal{C}_i, ..., \mathcal{C}_m\}$. In the example, there are four forward segments representing the following logical entities: an airplane containing 10 triples, a location with 2 triples, an airplane constructor with 2 triples and an airplane model with 2 triples.

The organization of the RDF triples in forward segments is ideal when solving star-pattern BGP queries like the one shown in Figure 2a. In order to solve it, an index of predicates as the one illustrated on Figure 3a could be built to scan only the most relevant forward segments.



(a) Forward segments predicate index (b) Backward segments predicate index

Fig. 3: Example of predicates indexes

Still, an organization of the data in forward segments is not optimal when solving queries with a very reduced number of predicates like the one shown in Figure 2b. To solve the query all the triples of the forward segment must be read, even the triples with predicates other than `has_motors`. To solve the query, 10 triples in the airplane forward segment must be scanned when only 2 triples are relevant. The problem is quite similar to the one that motivated vertical partitions in relational databases in which only the pertinent attributes of a table are accessed. Partitioning a table by attributes is very performant when the query does not access many attributes of the relation. This inspired us to propose another organization model for the RDF data that we name *backward segments*.

3.2 Creation of RDF groups of attributes

The vertical partitions for RDF data are obtained gathering first the triples by its incoming edges. In other words, we group the triples by its object. We obtain 13 groups `"5", "4", "75.4", "70.1", Boeing, ChicagoIL, Airbus, ToulouseFR`, `"Toulouse", "Chicago", A340, B747` and `Airplane` . We named these groups *backward entities* $\overleftarrow{E}$. Similarly to what was done for the forward entities, we group the backward entities sharing the same(or almost the same) set of predicates. We named each of these sets the i-th *backward segment* $\overleftarrow{\mathcal{C}_i}$ of G. As it is proven in Section ??, the set of backward segments form a partition set of the RDF graph.

For the example graph of Figure 1 we obtain 8 backward segments: the group of names with the `has_name` predicate (2 triples), the group of constructors (2 triples), lengths (2 triples), locations (2 triples), models (2 triples) and types (2 triples). There is a group gathering the triples having only the `nb_version` predicate (1 triple), and another group gathering the triples with the predicates `nb_version` and `nb_motor` (3 triples). The predicates could be organized with a simple index structure like the one shown in Figure 3b.

The solution to the query of Figure 2b is found scanning only the backward segment storing triples with the predicate `has_motors`. The execution engine checks the predicate's index of Figure3b and scans only the backward segment

$\overleftarrow{\mathcal{C}_2}$ that has only 3 triples. Comparing to the result obtained when the data are organized as *forward entities* and assuming that the cost to explore the predicate's index is the same, the execution process on the backward segments is more efficient. The process saves for this case the resources used to scan 7 triples. However, as it is the case for vertical partitions in the relational model, the performance can be degraded in SPARQL queries joining many single patterns.

3.3 Integration of the partitions by attributes and by instances

The integration of horizontal and vertical partitions in the relational model has been explored by many researchers in the past [2, 20]. Meanwhile, many massive processing systems propose replication strategies not only to recover and support fault tolerance but to improve the response time of queries. The Hadoop distributed framework for example, stores the data with a default replication factor of 3. In our approach, similar to the Fractured Mirrors [20] in relational databases, we consider a system that stores two copies of the data. One copy organizes the data as forward segments and another as backward segments. This configuration is useful especially when the workload is unknown in the initial partitioning stage.

We assume that the execution engine of the system is able to consider simultaneously forward and backward segments. The execution engine should be able to decide when each segment should be scanned according to a query. Let us consider the query of Figure 2c that contains 5 single patterns. The single patterns 1-3 form a star-query pattern, the patterns 3-4 and 4-5 form two path-query patterns. The execution engine proposes a plan in which, for instance, the matches to the query patterns 1-3 are found using the forward segment $\overrightarrow{\mathcal{C}_1}$. Then to find matches for pattern 4 the intermediate results could be joined with the forward pattern $\overrightarrow{\mathcal{C}_3}$. Finally, to find matches from patterns 1 to 5, the results could be joined with the backward segment $\overleftarrow{\mathcal{C}_8}$.

Forward and backward segments could be used as *fragments* to be allocated in a distributed or parallel system. The allocation strategies are discussed in the following section.

3.4 Allocation of segments

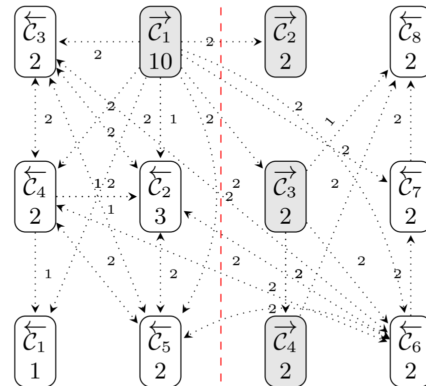
When we are dealing with distributed or parallel systems, the segments must be allocated into the processing workers. We assume that the size of a segment is smaller than the maximum capacity of a site. We treat the problem of big segments in the next section. Using a straightforward strategy like a round-robin for instance, may not produce an optimal performance. Firstly because the size of the segments is not uniform, therefore the data may be unequally distributed causing data skewness in some nodes. Secondly, queries like the one presented on Figure 2c may be affected by the cost to send intermediate results through the network. The network costs are the bottleneck of distributed systems and should be minimized to improve the performance. We consider an allocation strategy

in which the segments connected with a path pattern should be allocated in the same site. For example in the execution plan detailed previously, the segments $\overrightarrow{\mathcal{C}_1}, \overrightarrow{\mathcal{C}_3}$ and $\overleftarrow{\mathcal{C}_8}$ should be located in the same machine to avoid unnecessary network costs. The allocation problem is formalized in Section 4.3.

Allocation heuristics: graph partitioning The segments could be represented in a weighted directed graph $\mathcal{G}$ as it is illustrated in Figure 4b. As it is shown in Section 4.4, the allocation problem for the segments is an NP-Hard problem. Graph partitioning heuristics (e.g. METIS [13]) could be used to find good approximate solutions. In the graph of Figure 4b, the colored nodes correspond to the forward segments. The red dashed line separates the graph into two hypothetical partitions. An edge joining two nodes in different partitions(e.g. the edge between $\overrightarrow{\mathcal{C}_1}$ and $\overrightarrow{\mathcal{C}_2}$) represents a necessary network communication for a query joining both segments. The node's weights represent the total number of triples on each segment. The weights are represented inside the nodes of the graph of Figure 4b. The predicates on each segment are shown on the table of Figure 4a. The edge's weights represent the number of triples that should be transferred between two segments when they are joined. The weights are calculated according to the cases shown on Table 1. Not all the edges between segments are represented in the graph of Figure 4b for readability. Let us consider for example the weight of the edge between the forward segments $\overrightarrow{\mathcal{C}_1}$ and $\overrightarrow{\mathcal{C}_2}$. The weight in this case is equal to 2 since according to Table 1, the objects(`A340` and `B747`) of two triples in $\overrightarrow{\mathcal{C}_1}$ are the subject of two triples in $\overrightarrow{\mathcal{C}_2}$. The same reasoning is used to create the other edges and its weights.

Predicates	C
Forward	
has_model,has_length, nb_motors, type, has_constructor	$\overrightarrow{\mathcal{C}_1}$
nb_version	$\overrightarrow{\mathcal{C}_2}$
has_location	$\overrightarrow{\mathcal{C}_3}$
has_name	$\overrightarrow{\mathcal{C}_4}$
Backward	
nb_version	$\overleftarrow{\mathcal{C}_1}$
nb_version, nb_motor	$\overleftarrow{\mathcal{C}_2}$
has_type	$\overleftarrow{\mathcal{C}_3}$
has_model	$\overleftarrow{\mathcal{C}_4}$
has_length	$\overleftarrow{\mathcal{C}_5}$
has_constructor	$\overleftarrow{\mathcal{C}_6}$
has_location	$\overleftarrow{\mathcal{C}_7}$
has_name	$\overleftarrow{\mathcal{C}_8}$

(a) Predicates by segment



(b) Extract of segment graph $\mathcal{G}$

Fig. 4: Allocation example of graph G

Table 1: Edge's weights in segment graph $\mathcal{G}$

Case	Edge	$Weight_{Edge}$ = # triples t_i such that $t_i \in C_i, t_j \in C_j$ and:
$F^a F$	$(\overrightarrow{\mathcal{C}_i}, \overrightarrow{\mathcal{C}_j})$	$object(t_i) = subject(t_j)$
$B^b B_A$	$(\overleftarrow{\mathcal{C}_i}, \overleftarrow{\mathcal{C}_j})$	$subject(t_i) = subject(t_j)$
BB_B	$(\overleftarrow{\mathcal{C}_i}, \overleftarrow{\mathcal{C}_j})$	$object(t_i) = subject(t_j)$
FB_A, BF_A	$(\overrightarrow{\mathcal{C}_i}, \overleftarrow{\mathcal{C}_j})$	$subject(t_i) = subject(t_j)$
FB_B, BF_B	$(\overrightarrow{\mathcal{C}_i}, \overleftarrow{\mathcal{C}_j})$	$object(t_i) = subject(t_j)$

[a] Forward segment
[b] Backward segment

Allocation of big segments When the size of a forward or backward segment is bigger than the maximum available space for a site, the entity needs to be repartitioned. A segment is formed by sets of forward or backward entities. To partition within a segment, we apply a function that sub-partitions the segments in such a way that all the triples of a backward or forward entity belong to the same partition. In other words, the repartitioning function should not divide the forward or backward entities in a segment when it creates sub-partitions.

Let us consider that the size of an entity $\mathcal{C}$ is bigger than the available space for a site. To sub-partition the entity, we apply a function that maps the forward or backward entities to a partition according to the values of a (some) predicate(s). If the segment $\mathcal{C}$ is formed by the predicates $p_1..p_k$, the conditions of a repartition function are predicates such as: i)$p_1 \geq value$ and ii)$p_1 < value$, distributing the entities of the segment in two groups. In order to choose the best sub-partitioning function, statistics about the entity must be available (e.g. number of distinct values per predicate, histograms of values for a predicate).

For example, let us consider the forward entity $\overrightarrow{\mathcal{C}_1}$ of G with 10 triples. Supposing that a site cannot hold more than 6 triples, $\overrightarrow{\mathcal{C}_1}$ cannot be allocated to a single site. To repartition the forward segment, we apply a function, for instance $\sigma_{\overrightarrow{\mathcal{C}_1}.has_length<73}(\overrightarrow{\mathcal{C}_1})$, whose predicates are set based on segment's statistics. The original segment is split into two partitions: $\overrightarrow{\mathcal{C}_{1A}}$ (5 triples) with all the facts of the instance `<:Airplane20>` and $\overrightarrow{\mathcal{C}_{1B}}$ (5 triples) with all the facts of the instance `<:Airplane1>`. All the edges of $\overrightarrow{\mathcal{C}_1}$ are maintained in $\overrightarrow{\mathcal{C}_{1A}}$ and $\overrightarrow{\mathcal{C}_{1B}}$. Finally, the partitioning algorithm is performed to decide the allocation of segments considering the recently created sub-partitions.

3.5 Use of a declarative language to describe a partition

The creation of forward and backward segments before partitioning a raw RDF file contributes to integrate a logical dimension to the purely physical partitioning process. The forward and backward segments are described by its predicate's set. As it was mentioned previously, partitioning big segments is done based on

these predicates. A declarative language could be then used to describe the creation and partitioning processes for both types of segments. The declarations are done similarly to the declarations of tables with the DDL (data declaration language) in SQL.

Both types of entities are declared as follows:

```
CREATE {FORWARD,BACKWARD,BOTH}_SEGMENTS
     FROM [raw rdf file path] WITH [max_size]
```

The `CREATE` command is taken from the SQL DDL. The instruction `{FORWARD, BACKWARD}_SEGMENTS` indicate the creation of only forward or backward entities respectively. To create both types of segments the instruction: `BOTH_SEGMENTS` must be used. The `[raw rdf file path]` is the path to the raw file to be loaded into the processing system. Finally the parameter `WITH [max_size]` states the maximum allowed size for a segment. At the end of both instructions, the forward and backward entities should be created and a list of the segments, its predicates and statistics should be available to the user. This file creates an `id` for each segment used mainly when the segment is repartitioned. At the end of the instruction the forward and backward entities that need to be repartitioned, because their size is bigger than the max_size parameter, should also be indicated.

To repartition a segment, the following syntax is used:

```
CREATE [{FORWARD,BACKWARD}_SEGMENT [id]
     PARTITION BY ([predicate function])
```

This instruction overwrites a segment (identified with `id`) and creates the partitions according to the predicate's function. The predicate's function model is inspired as well from the DDL of the relational model:

```
(
PARTITION [partition_name_1] [predicate] {LESS,MORE} THAN [VALUE],
...
PARTITION [partition_name_m] [predicate] {LESS,MORE} THAN [VALUE].
)
```

The following expression declares the repartition of the segment Airplane (we assumed that its ID is 1) used as example at the previous section:

```
CREATE FORWARD_SEGMENT 1
    PARTITION BY (
      PARTITION part1 has_length LESS THAN  (73),
      PARTITION part2 has_length VALUES LESS THAN < (maxvalue))
```

The maxvalue is a reserved word indicating the maximum value of the object with the predicate `has_length`.

To allocate the segments to the processing sites, the declaration is done with the following syntax:

```
ALLOCATE {FORWARD,BACKWARD,BOTH}_ENTITIES BY [partition function]
   WITH [nb_partitions]
```

As it was done at the creation stage of segments, we can allocate individually forward and backward edges with the instruction `ALLOCATE {FORWARD,BACKWARD}_ENTITIES`. To allocate both segments the keyword `BOTH_ENTITIES` should be used. The partitioning strategy to be used by the system is indicated in the `[partition function]` instruction. The available partitioning strategies are:

- `ROUND_ROBIN`
- `METIS`

The `nb_partitions` parameter indicates the number of sites in which the data should be distributed. If one segment must be moved to another partition by implementation constraints, the declaration is:

```
ALLOCATE {FORWARD,BACKWARD}_SEGMENT [id]
     TO SITE ([site id])
```

In the following section we give the formal definitions for the entities, segments and partitioning algorithms applied throughout the example developed on this section.

4 Formal definitions

In this section we define the forward and backward segments for an input RDF dataset that we represent as G. The data in G is stored in either of the RDF syntaxes (e.g. RDF/XML, N-Triples, Turtle). We start defining the forward entities that compose forward segments. Then, we define a forward segment and prove that the set of all forward segments constitutes a partition of G using the completeness, reconstruction and disjointedness rules. Next, we formalize the backward entities and segments proving that they also form a partition of G. Later, we define the allocation problem in which the forward and backward segments are considered as fragments. Finally, we show the formal mapping of the allocation problem to a graph partitioning problem.

4.1 Forward segments construction

We start defining the functions $s : (s, p, o) \to s$ and $o : (s, p, o) \to o$ returning the subject and object of an RDF triple respectively. They are applied in some of the definitions of this section.

Definition 1. ***Forward entity*** *A forward entity denoted as $\overrightarrow{E}$ is a subset of G in which the triples share the same subject. Formally $\overrightarrow{E} \subseteq G$, such that $\overrightarrow{E} = \{(s, p, o) | \forall_{i \neq j}(s_i = s_j)\}$.*

The forward entity creation algorithm is described in Alg. 1. Since the algorithms to create forward and backward entities are very similar, we use the same algorithm to describe both procedures. In this case, we consider that the symbol $\overleftrightarrow{E}$ in Alg. 1 represents actually $\overrightarrow{E}$. For each triple t in the graph, we verify if there is already a forward entity that contains triples with the same subject on the forward entity set(Step 2). If it is the case, the triple is added to this set (Step 3), otherwise a new forward entity containing the triple t is created (Step 5).

Algorithm 1 Entities creation

```
    Input: RDF dataset G
    Output: Set of segments ⟷E = {⟷E_1, ..., ⟷E_p}
1: for each t in G do
2:    if ∃q ∈ ⟷E_j | (if →E_j → s(q) = s(t) else ←E_j → o(q) = o(t)) then
3:       ⟷E_j = ⟷E_j ∪ {t}
4:    else
5:       ⟷E_k = {t}
6:       ⟷E = ⟷E ∪ ⟷E_k
7:    end if
8: end for
9: return ⟷E
```

Theorem 1. *A triple belongs to one and only one forward entity. Let t be a triple $t \in G$, if $t \in \overrightarrow{E}_i \Rightarrow \forall_{i \neq j}(t \notin \overrightarrow{E}_j)$.*

Proof. To prove by contradiction. Let us assume that a triple $t \in \overrightarrow{E}_i$ and $t \in \overrightarrow{E}_j$. Let us consider two not equal triples t_1, t_2, both different from t, such that $t_1 \in \overrightarrow{E}_i$ and $t_2 \in \overrightarrow{E}_j$. Based on the forward entities definition, $s(t_1) = s(t)$ and $s(t_2) = s(t)$. Using the transitivity of the equality, $s(t_1) = s(t_2)$ which is impossible if $\overrightarrow{E_i} \neq \overrightarrow{E_j}$.

Definition 2. ***Forward segment*** *A forward segment is a set of forward entities $\overrightarrow{E}$ that share the same predicates (or almost the same according to a threshold τ). A forward entity $\overrightarrow{E}_i$ belongs to one and only one forward segment $\overrightarrow{\mathcal{C}}_i$. Formally $\overrightarrow{\mathcal{C}} = \{\overrightarrow{E} | \forall_{i \neq j}\big(Sim(p(\overrightarrow{E}_i), p(\overrightarrow{E}_j)) \geq \tau\big) \Rightarrow \overrightarrow{E}_i \in \overrightarrow{E} \wedge \overrightarrow{E}_j \in \overrightarrow{E}\}$.*

The function $p(\overrightarrow{E}_i)$ returns the set of distinct predicates in the forward entity $\overrightarrow{E}_i$. The similarity function $Sim(p_1, p_2)$ returns the similarity between two sets of predicates p_1 and p_2. This function could be, for example, a Jaccard similarity between both sets as:

$$\frac{|p_1 \cap p_2|}{Max(|p_1|, |p_2|)}$$

To assign the entity $\overrightarrow{E}_i$ to a segment, the pairwise similarity is calculated between the entity $\overrightarrow{E}_i$ with all the entities that have already been assigned to a

segment. Let us consider two already assigned entities $\overrightarrow{E}_j \in \overrightarrow{\mathcal{C}_j}$ and $\overrightarrow{E}_k \in \overrightarrow{\mathcal{C}_k}$. When calculating the similarity one can encounter the following cases:

- If $Sim\big(p(\overrightarrow{E}_i), p(\overrightarrow{E}_j)\big) > Sim\big(p(\overrightarrow{E}_i), p(\overrightarrow{E}_k)\big) > \tau$ then $E_i \in \overrightarrow{\mathcal{C}_j}$.
- If both scores have the same value, $Sim\big(p(\overrightarrow{E}_i), p(\overrightarrow{E}_j\big) = Sim\big(p(\overrightarrow{E}_i), p(\overrightarrow{E}_k)\big) > \tau$, then $\overrightarrow{E}_i$ is randomly assigned to either of the sets $\overrightarrow{\mathcal{C}_j}$ or $\overrightarrow{\mathcal{C}_k}$.
- If the forward entity has not a similarity score greater than the threshold when comparing with all the forward entities of the graph G, formally expressed as $\forall_{i \neq j}(Sim(p(\overrightarrow{E}_i), p(\overrightarrow{E}_j)) < \tau)$, then the forward entity $\overrightarrow{E}$ form a forward segment on its own ($\overrightarrow{\mathcal{C}} = \overrightarrow{E}_i$).

The algorithm to generate the forward segments is shown in Alg. 2. Similarly to the notation in Alg. 1, the symbols $\overleftrightarrow{E}$ and $\overleftrightarrow{C}$ correspond actually to $\overrightarrow{E}$ and $\overrightarrow{C}$ respectively when creating forward segments. For each forward entity we calculate the pairwise similarity between the forward entity and the already assigned forward entities (Step 2). The entity $\overrightarrow{E}_i$ is assigned to the segment according to the cases described previously. The similarity between each distinct set of predicates sets could be pre-calculated to improve the similarity search efficiency. When the similarity is greater or equal than the threshold (Steps 4-6), the forward entity is added to the corresponding forward segment. Otherwise the forward segment is created on its own (Steps 8,12).

Algorithm 2 Segment creation

Input: Set of entities $\overleftrightarrow{E} = \{\overleftrightarrow{E}_1, ..., \overleftrightarrow{E}_p\}$, threshold similarity τ
Output: Set of segments $\overleftrightarrow{C} = \{\overleftrightarrow{C}_1, ..., \overleftrightarrow{C}_p\}$
1: **for each** $\overleftrightarrow{E}_i$ in $\overleftrightarrow{E}$ **do**
2: **if** $maximum\Big(\forall_{i \neq j}\big(Sim(p(\overleftrightarrow{E_i}), p(\overleftrightarrow{E_j}))\big)\Big) \geq \tau$ **then**
3: **if** $\exists_{\overleftrightarrow{C}_k \in \overleftrightarrow{C}}(\overleftrightarrow{E_j} \in \overleftrightarrow{C}_k)$ **then**
4: $\overleftrightarrow{C} = \overleftrightarrow{C} - \{\overleftrightarrow{C}_k\}$
5: $\overleftrightarrow{C}_k = \overleftrightarrow{C}_k \cup \{\overleftrightarrow{E_i}\}$
6: $\overleftrightarrow{C} = \overleftrightarrow{C} \cup \{\overleftrightarrow{C}_k\}$
7: **else**
8: $\overleftrightarrow{C}_k = \{\overleftrightarrow{E_i}\}$
9: $\overleftrightarrow{C} = \overleftrightarrow{C} \cup \{\overleftrightarrow{C}_k\}$
10: **end if**
11: **else**
12: $\overleftrightarrow{C}_k = \{\overleftrightarrow{E_i}\}$
13: $\overleftrightarrow{C} = \overleftrightarrow{C} \cup \{\overleftrightarrow{C}_k\}$
14: **end if**
15: **end for**
16: **return** $\overleftrightarrow{C}$

Theorem 2. *The set $\overrightarrow{C} = \{\overrightarrow{\mathcal{C}}_1, ..., \overrightarrow{\mathcal{C}}_l\}$ of all forward segments for the graph G is a correct[5] partition set of the graph G.*

Proof. We will show that the three correctness fragmentation rules are enforced.

- *Completeness*: If $t \in G \Rightarrow \exists \overrightarrow{E}_i$ such that $t \in \overrightarrow{E}_i$, $\overrightarrow{E}_i \in \overrightarrow{\mathcal{C}}_i$, and $\overrightarrow{\mathcal{C}}_i \in \overrightarrow{C}$. By contradiction, if $\forall \overrightarrow{E}_i, t \notin \overrightarrow{E}_i$ then the triple's t subject $s(t)$ does not equal any of the subjects of the forward entities $\overrightarrow{E}$. Since the forward entities $\overrightarrow{E}$ are built by grouping first all triples of the graph G by subject, the triple t's subject is not equal to the subject of any triple in G, therefore $t \notin G$.
- *Reconstruction*: It is possible to define an operator ∇ such that $G = \nabla \overrightarrow{\mathcal{C}_i}, \forall \overrightarrow{\mathcal{C}_i} \in \overrightarrow{C}$. For the forward entity classes, the operator ∇ equals the union operator $\cup$. In other words, $\bigcup_{i=1}^{l} \overrightarrow{\mathcal{C}_i} = G$. By contradiction, if $\exists t \in \overrightarrow{\mathcal{C}_i}$ such that $t \notin G$, then the subject of any triple in $\overrightarrow{\mathcal{C}_i}$ does not belong to G. This is impossible because by definition, the forward entities are created grouping all triples from the initial graph G by subject. This would be possible only if $t \notin \overrightarrow{\mathcal{C}_i}$ and therefore the set of forward entities would not be complete.
- *Disjointness*: $\forall_{i \neq j} (\overrightarrow{\mathcal{C}_i} \cap \overrightarrow{\mathcal{C}_j} = \emptyset)$. By contradiction, if $\exists (\overrightarrow{\mathcal{C}_i} \cap \overrightarrow{\mathcal{C}_j} = \{t\})$ then $t \in \overrightarrow{E_i}$ and $t \in \overrightarrow{E_j}$ $(\overrightarrow{E_i} \in \overrightarrow{\mathcal{C}_i}, \overrightarrow{E_j} \in \overrightarrow{\mathcal{C}_j})$, which is impossible unless the same triple had two different subjects.

4.2 Backward segments construction

Definition 3. ***Backward entity*** *A backward entity denoted as $\overleftarrow{E}$ is a subset of the original RDF graph G in which the triples share the same object. Formally $\overleftarrow{E} \subseteq G$ such that $\overleftarrow{E} = \{(s, p, o) | \forall_{i \neq j} (o_i = o_j)\}$. A backward entity $\overleftarrow{E}_i$ belongs to one and only one backward segment $\overleftarrow{\mathcal{C}}_i$.*

The backward entity creation algorithm is described in the Alg. 1. In this case we consider that the symbols $\overleftrightarrow{E}$ and $\overleftrightarrow{C}$ correspond actually to $\overrightarrow{E}$ and $\overrightarrow{C}$ respectively. For each triple t in the graph, we verify if there is already a backward entity that contains triples with the same object on the backward entity set(Step 2). If so the triple is added to this set (Step 3), otherwise we a new backward entity containing the triple t is created (Step 5).

Theorem 3. *A triple belongs to one and only one backward entity. Let t be a triple $t \in G$, if $t \in \overleftarrow{E}_i \Rightarrow \forall_{i \neq j} (t \notin \overleftarrow{E}_j)$.*

Proof. To prove by contradiction. Let us assume that a triple $t \in \overleftarrow{E}_i$ and $t \in \overleftarrow{E}_j$. Let us consider two not equal triples t_1, t_2, both different from t, such that $t_1 \in \overleftarrow{E}_i$ and $t_2 \in \overleftarrow{E}_j$. Based on the backward entity definition, $o(t_1) = o(t)$ and $o(t_2) = o(t)$. Using the transitivity property of the equality, $o(t_1) = o(t_2)$ which is impossible if $\overleftarrow{E_i} \neq \overleftarrow{E_j}$.

[5] According to the correctness fragmentation rules in [18]

Definition 4. ***Backward segment*** *A backward segment* $\overleftarrow{\mathcal{C}}$ *is a set of backward entities* $\overleftarrow{E}$ *that share the same predicates (or almost the same according to a threshold* τ*). Formally* $\overleftarrow{\mathcal{C}} = \{\overleftarrow{E} | Sim(p(\overleftarrow{E}_i), p(\overleftarrow{E}_j)) \geq \tau\}$.

The similarity is calculated applying the same rules for the forward segments in Definition 2. The similarity function could be, as it is the case for forward entities, a Jaccard similarity between the predicates.

The algorithm to generate the backward segments is shown in Alg. 2. The backward segment creation is created in the same way as the forward segments. The steps are described in Alg. 1, the symbols $\overleftrightarrow{E}$ and $\overleftrightarrow{C}$ correspond actually to $\overleftarrow{E}$ and $\overleftarrow{C}$ respectively when creating backward segments.

Theorem 4. *The set* $\overleftarrow{C} = \{\overleftarrow{\mathcal{C}}_1, ..., \overleftarrow{\mathcal{C}}_m\}$ *of all backward segments for the graph* G *is a correct*[6] *partition set of the graph* G.

Proof. We will show that the three correctness fragmentation rules are enforced.

- *Completeness*: If $t \in G \Rightarrow \exists \overleftarrow{E}_i$ such that $t \in \overleftarrow{E}_i$, $\overleftarrow{E}_i \in \overleftarrow{\mathcal{C}}_i$, and $\overleftarrow{\mathcal{C}}_i \in \overleftarrow{C}$. By contradiction, if $\forall \overleftarrow{E}, t \notin \overleftarrow{E}$ then the triple's t object $o(t)$ does not equal any of the objects of the backward entities $\overleftarrow{E}$. Since the backward entities $\overleftarrow{E}$ are built by grouping first all triples of the graph G by object, the triple t's object is not equal to the object of any triple in G, therefore $t \notin G$.
- *Reconstruction*: It is possible to define an operator ∇ such that $G = \nabla \overleftarrow{\mathcal{C}_i}, \forall \overleftarrow{\mathcal{C}_i} \in \overleftarrow{C}$. For the backward entity classes, the operator ∇ equals the union operator $\cup$. In other words, $\bigcup_{i=1}^{m} \overleftarrow{\mathcal{C}_i} = G$. By contradiction, if $\exists t \in \overleftarrow{\mathcal{C}_i}$ such that $t \notin G$, then the object of any triple in $\overleftarrow{\mathcal{C}_i}$ does not belong to G which is impossible because by definition the backward entities are created grouping all triples from the initial graph G by object. This would be possible only if $t \notin \overleftarrow{\mathcal{C}_i}$ and therefore the set of backward entities would not be complete.
- *Disjointness*: $\forall_{i \neq j} (\overleftarrow{\mathcal{C}_i} \cap \overleftarrow{\mathcal{C}_j} = \emptyset)$. By contradiction, if $\exists (\overleftarrow{\mathcal{C}_i} \cap \overleftarrow{\mathcal{C}_j} = \{t\})$ then $t \in \overleftarrow{E_i}$ and $t \in \overleftarrow{E_j}$ ($\overleftarrow{E_i} \in \overleftarrow{\mathcal{C}_i}, \overleftarrow{E_j} \in \overleftarrow{\mathcal{C}_j}$), which is impossible unless the same triple had two different objects.

We have proven that the sets of forward and backward segments ($\overrightarrow{C} = \{\overrightarrow{\mathcal{C}}_1, ..., \overrightarrow{\mathcal{C}}_l\}$, $\overleftarrow{C} = \{\overleftarrow{\mathcal{C}}_1, ..., \overleftarrow{\mathcal{C}}_m\}$ respectively) induce *correct* partitions (i.e. complete, disjoint and rebuildable) of the original RDF dataset G. The elements on each set correspond to *fragments* of G that are adopted as distribution units during the *allocation* step. In the next section we describe the allocation strategies for the sets of fragments $\overrightarrow{C}, \overleftarrow{C}$.

4.3 Allocation problem

In this section, we state the allocation problem for RDF distributed systems and specify our assumptions. The allocation is performed for a set of segments into a set of sites.

[6] According to the correctness fragmentation rules in [18]

Inputs Let $V = \{V_1, ..., V_p\}$ be the set of forward and backward segments $V = \{\overrightarrow{\mathcal{C}_1}, ..., \overrightarrow{\mathcal{C}_l}\} \cup \{\overleftarrow{\mathcal{C}_1} ... \overleftarrow{\mathcal{C}_m}\}$ with cardinality $p = l + m$. The **sites** are represented by the set $S = \{S_1, ..., S_q\}$. We assume that the size of a single segment is always inferior to the maximum capacity of a site $|S_i| \geq |V_j|$. An **imbalance factor** denoted by ϵ is used to avoid high imbalance of the size of the partitions.

Preliminary functions The functions defined in this section are used in the definition of the objective function and restrictions of the allocation problem. Their definitions are stated in Table 2.

Table 2: Preliminary functions

Function	Returns
$W_V : V \to \mathbb{N}^+$	Number of triples per segment.
$W_F : V \times V \to \mathbb{N}$	Number of triples *shared* by two segments V_1 and V_2. Its calculation depends on whether the relation is for example between a forward and backward segment or between two forward segments. The cases are detailed on Table 1 .
$x_{iS_k} : V \to \{0, 1\}$	The assignation function returns 1 if and only if $V_i \in S_k$, otherwise it returns zero.

Problem definition The partitioning problem consists then in finding an allocation function $X_{iS_k} : V \to S$ that minimizes the total number of *shared* triples by two segments. Given a set of segments V, sites S and an imbalance factor ϵ,

$$minimize \sum_{\substack{i,j \in \{1,...,p\} \\ k \in \{1,...,q\}}} \Big((x_{iS_k} \oplus x_{jS_k}) \cdot W_E(V_i, V_j) \Big) \tag{1}$$

Subject to:

$$\forall_{k \in \{1..q\}} \left| \sum_{i=1}^{p} \left(x_{iS_k} \cdot W_V(V_i) \right) - \sum_{j=1}^{p} \left(x_{jS_k} \cdot W_V(V_j) \right) \right|_{i \neq j} \leq \epsilon \tag{2}$$

$$\forall_{i \in \{1..p\}} \Big(\sum_{k=1}^{q} x_{iS_k} = 1 \Big) \tag{3}$$

Complexity NP-Hard still to be determined.

4.4 Allocation as a graph partitioning problem

The allocation of segments in set of sites is an *NP-Hard?* problem as it was proven in last section. Likewise, the data partitioning problem has been proven to be NP-Complete [7, 22, 15]. Finding exact solutions is consequently computationally unfeasible and therefore heuristics producing sub-optimal results are the most convenient strategies.

In this line, we mapped the segments to a directed weighted graph, transforming the allocation problem into a graph partitioning problem. The graph partitioning problem has been proved to be as well a very complex and computationally expensive problem. However, many efficient heuristics have been developed (e.g. METIS [13]).

Let us map the set of entity classes $\mathcal{C}$ into a directed weighted graph represented by the quadruple $\mathcal{G} = (V, E, W_V, W_E)$. V correspond to the set of nodes composed by the union of forward and backward segments. The node weights W_V correspond to the number of triples on each segment. E represent the set of edges and W_E its weights. An edge is added between two nodes when at least one of its triples meets the characteristics described in Table 1. The weight of an edge represent the number of triples to be transferred between two segments when they are joined with a path pattern.

Given a graph $\mathcal{G}$ and a given a number $p \in \mathbb{N}^+$ indicating the number of partitions, the graph partitioning problem asks for blocks of nodes $\mathcal{V}_1, ..., \mathcal{V}_p$ such that $\mathcal{V}_1 \cup ... \cup \mathcal{V}_p = V$ and $\mathcal{V}_i \cap \mathcal{V}_j \forall i \neq j$. As it was defined before, a balance parameter ϵ is used to regulate the balance size between each partition.

Objective function We seek a partition that minimizes the *total cuts*. In other words, the objective computes the sum of the weight of the cut edges. A cut edge is an edge formed by two nodes belonging to two different partitions.

$$minimize \sum_{i<j} w(E_{ij}) \tag{4}$$

Subject to:

$$|w(E_i) - w(E_j)|_{\forall i \neq j} \leq \epsilon \tag{5}$$

The graph of segments needs to be distributed across several nodes and com-

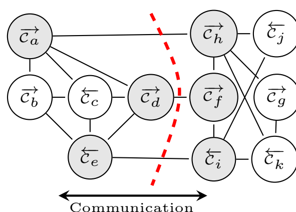


Fig. 5: Graph partitioning example

munication between sites takes place for nodes that have non-local edges. We assume that the number of cuts is proportional to the units of communications as shown in Figure 5 in which the filled nodes located in different partitions imply network costs when both segments are joined (e.g. the nodes $\overrightarrow{\mathcal{C}_a}$ and $\overrightarrow{\mathcal{C}_h}$).

5 Related Work

In this section we describe briefly the principal storage models used by RDF systems. Then we classify the RDF partitioning approaches and we cite the studies comparing the performance between partitioning strategies.

5.1 RDF data storage models

The data storage model of several RDF systems is based on the relational model. The triples of these systems are stored on disk using one of the following strategies: i) *Triple table*: a single table with three columns (subject, predicate, object) is created. The most popular systems using this strategy are RDF-3X[17] and Hexastore[27]. ii) *Property table:* a single table whose dimensions are determined by the number of subjects and distinct predicates is created. This strategy is applied by the Jena2[28] system. iii) *Vertical partition:* this strategy creates a two-column table (subject, object) per predicate. The strategy is applied by SW-Store[1], HadoopRDF[5] and Jena-HBase[14]. More sophisticated storage models are used by the systems G-Store[31], Trinity.RDF[29] and H2RDF[19] in which the data are stored using graph-representation models like adjacency lists.

5.2 Partitioning RDF techniques

Distributed[7] RDF systems are classified in two major groups. The first group concerns *Federated systems* (e.g. HiBISCuS [24], SPLENDID [8]) in which SPARQL queries run over multiple endpoints in different physical locations. Federated systems are out of the scope of our research but we mention them for the completeness. The second group gathers systems in which the RDF data are distributed among different data nodes being part of a single RDF storage solution. The systems in this category are called *clustered storage systems* and its partitioning strategies are classified as:

1. *Hash partitioning:* the allocation of triples is performed according to a hash value computed on the subject(or predicate) of a triple modulo the number of computer nodes. The strategy ensures that the triples of the same subject(or predicate) are located in the same partition. The strategy is used in systems like Virtuoso[6], Trinity.RDF[29] and HadoopRDF[5].
2. *Hierarchical hash:* also called semantic-hash partitioning, the strategy builds a path hierarchy based on the subject's IRIs. It is applied in SHAPE[16].

[7] Distributed or Parallel

3. *Minimal edge-cut:* this strategy solves the partitioning problem as a k-way graph partitioning using heuristic packages like METIS[13]. The strategy aims to minimize the number of edges between vertices of different partitions. This strategy is applied in the EAGRE[30] system. A variant of this strategy, named *n-hop guarantee*, replicates certain data to reduce the number of exchanged intermediate results. This strategy is applied in H-RDF-3X[10]. Another variant uses the workload to assign weights to the edges and perform a graph partitioning heuristic on a weighted graph. This strategy is applied in the systems WARP[9] and [4].
4. *Round-robin*: This technique assigns an x number of triples to partition 1, then an x number to partition 2 and so on. The strategy is applied in [23].
5. *Other approaches:* the previous strategies are built specifically to deal with RDF. Several other strategies work on top of distributed platforms like Hadoop or key-value stores. In these systems, the storage back-end is in charge of distributing the triples. For example, when the graph is directly stored into the HDFS (Hadoop Distributed File System), the data are split into blocks of fixed size that are hash-distributed among the workers. This is the case of systems like S2RDF[26](with Spark), SHARD[21] and PigSparql[25].

Recently, the studies [12, 3] evaluated and compared the impact of certain data placement strategies in distributed RDF stores. Some efforts have also been made to build a system to benchmark partitioning strategies. The system Koral[11], for example, allows the integration of different RDF graph partitioning techniques to investigate their behavior.

6 Conclusion

In this paper, we added a logical dimension to the partitioning process of RDF graphs. The logic dimension allows to declare the partitions with a declarative language. The allocation methods show that the exchanges of intermediate results are minimized using a min-cut algorithm. Our on-going projects include the consideration of the query workload in the declaration of partitions and the considering of changes on the dataset (updates).

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: Sw-store: a vertically partitioned DBMS for semantic web data management. VLDB J. **18**(2), 385–406 (2009). https://doi.org/10.1007/s00778-008-0125-y, https://doi.org/10.1007/s00778-008-0125-y
2. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Weikum, G., König, A.C., Deßloch, S. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004. pp. 359–370. ACM (2004). https://doi.org/10.1145/1007568.1007609, http://doi.acm.org/10.1145/1007568.1007609

3. Akhter, A., Ngomo, A.N., Saleem, M.: An empirical evaluation of RDF graph partitioning techniques. In: Faron-Zucker, C., Ghidini, C., Napoli, A., Toussaint, Y. (eds.) Knowledge Engineering and Knowledge Management - 21st International Conference, EKAW 2018, Nancy, France, November 12-16, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11313, pp. 3–18. Springer (2018). https://doi.org/10.1007/978-3-030-03667-6_1, https://doi.org/10.1007/978-3-030-03667-6_1
4. Al-Ghezi, A.I.A., Wiese, L.: Adaptive workload-based partitioning and replication for RDF graphs. In: Hartmann, S., Ma, H., Hameurlain, A., Pernul, G., Wagner, R.R. (eds.) Database and Expert Systems Applications - 29th International Conference, DEXA 2018, Regensburg, Germany, September 3-6, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11030, pp. 250–258. Springer (2018). https://doi.org/10.1007/978-3-319-98812-2_21, https://doi.org/10.1007/978-3-319-98812-2_21
5. Du, J., Wang, H., Ni, Y., Yu, Y.: Hadooprdf: A scalable semantic data analytical engine. In: Huang, D., Ma, J., Jo, K., Gromiha, M.M. (eds.) Intelligent Computing Theories and Applications - 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7390, pp. 633–641. Springer (2012). https://doi.org/10.1007/978-3-642-31576-3_80, https://doi.org/10.1007/978-3-642-31576-3_80
6. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Auer, S., Bizer, C., Müller, C., Zhdanova, A.V. (eds.) The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), September 26-28, 2007, Leipzig, Germany. LNI, vol. 113, pp. 59–68. GI (2007), http://subs.emis.de/LNI/Proceedings/Proceedings113/article1851.html
7. Eswaran, K.P.: Placement of records in a file and file allocation in a computer. In: IFIP Congress. pp. 304–307 (1974)
8. Görlitz, O., Staab, S.: SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In: Hartig, O., Harth, A., Sequeda, J.F. (eds.) Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011. CEUR Workshop Proceedings, vol. 782. CEUR-WS.org (2011), http://ceur-ws.org/Vol-782/GoerlitzAndStaab_COLD2011.pdf
9. Hose, K., Schenkel, R.: WARP: workload-aware replication and partitioning for RDF. In: Chan, C.Y., Lu, J., Nørvåg, K., Tanin, E. (eds.) Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013. pp. 1–6. IEEE Computer Society (2013). https://doi.org/10.1109/ICDEW.2013.6547414, https://doi.org/10.1109/ICDEW.2013.6547414
10. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. PVLDB **4**(11), 1123–1134 (2011), http://www.vldb.org/pvldb/vol4/p1123-huang.pdf
11. Janke, D., Staab, S., Thimm, M.: Koral: A glass box profiling system for individual components of distributed RDF stores. In: Usbeck, R., Ngomo, A.N., Kim, J., Choi, K., Cimiano, P., Fundulaki, I., Krithara, A. (eds.) Joint Proceedings of BLINK2017: 2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21st - to - 22nd, 2017. CEUR Workshop Proceedings, vol. 1932. CEUR-WS.org (2017), http://ceur-ws.org/Vol-1932/paper-05.pdf

12. Janke, D., Staab, S., Thimm, M.: Impact analysis of data placement strategies on query efforts in distributed RDF stores. J. Web Semant. **50**, 21–48 (2018). https://doi.org/10.1016/j.websem.2018.02.002, https://doi.org/10.1016/j.websem.2018.02.002
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Scientific Computing **20**(1), 359–392 (1998). https://doi.org/10.1137/S1064827595287997, https://doi.org/10.1137/S1064827595287997
14. Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.M., Castagna, P.: Jena-hbase: A distributed, scalable and effcient RDF triple store. In: Glimm, B., Huynh, D. (eds.) Proceedings of the ISWC 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012. CEUR Workshop Proceedings, vol. 914. CEUR-WS.org (2012), http://ceur-ws.org/Vol-914/paper_14.pdf
15. Lam, K., Yu, C.T.: An approximation algorithm for a file-allocation problem in a hierarchical distributed system. In: Chen, P.P., Sprowls, R.C. (eds.) Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data, Santa Monica, California, USA, May 14-16, 1980. pp. 125–132. ACM Press (1980). https://doi.org/10.1145/582250.582270, https://doi.org/10.1145/582250.582270
16. Lee, K., Liu, L.: Scaling queries over big RDF graphs with semantic hash partitioning. PVLDB **6**(14), 1894–1905 (2013). https://doi.org/10.14778/2556549.2556571, http://www.vldb.org/pvldb/vol6/p1894-lee.pdf
17. Neumann, T., Weikum, G.: RDF-3X: a risc-style engine for RDF. PVLDB **1**(1), 647–659 (2008). https://doi.org/10.14778/1453856.1453927, http://www.vldb.org/pvldb/1/1453927.pdf
18. Özsu, M.T., Valduriez, P.: Principles of distributed database systems. Springer Science & Business Media (2011)
19. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H2RDF+: high-performance distributed joins over large-scale RDF graphs. In: Hu, X., Lin, T.Y., Raghavan, V.V., Wah, B.W., Baeza-Yates, R.A., Fox, G.C., Shahabi, C., Smith, M., Yang, Q., Ghani, R., Fan, W., Lempel, R., Nambiar, R. (eds.) Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA. pp. 255–263. IEEE (2013). https://doi.org/10.1109/BigData.2013.6691582, https://doi.org/10.1109/BigData.2013.6691582
20. Ramamurthy, R., DeWitt, D.J., Su, Q.: A case for fractured mirrors. In: VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China. pp. 430–441. Morgan Kaufmann (2002), http://www.vldb.org/conf/2002/S12P03.pdf
21. Rohloff, K., Schantz, R.E.: Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In: Kosar, T. (ed.) DIDC'11, Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, San Jose, CA, USA, June 8, 2011. pp. 35–44. ACM (2011). https://doi.org/10.1145/1996014.1996021, http://doi.acm.org/10.1145/1996014.1996021
22. Saccà, D., Wiederhold, G.: Database partitioning in a cluster of processors. ACM Trans. Database Syst. **10**(1), 29–56 (1985). https://doi.org/10.1145/3148.3161, https://doi.org/10.1145/3148.3161
23. Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., Ngomo, A.N.: A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web **7**(5), 493–518 (2016). https://doi.org/10.3233/SW-150186, https://doi.org/10.3233/SW-150186

24. Saleem, M., Ngomo, A.N.: Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8465, pp. 176–191. Springer (2014). https://doi.org/10.1007/978-3-319-07443-6_13, https://doi.org/10.1007/978-3-319-07443-6_13
25. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: Pigsparql: mapping SPARQL to pig latin. In: Virgilio, R.D., Giunchiglia, F., Tanca, L. (eds.) Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, Athens, Greece, June 12, 2011. p. 4. ACM (2011). https://doi.org/10.1145/1999299.1999303, http://doi.acm.org/10.1145/1999299.1999303
26. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. PVLDB **9**(10), 804–815 (2016). https://doi.org/10.14778/2977797.2977806, http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf
27. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1), 1008–1019 (2008). https://doi.org/10.14778/1453856.1453965, http://www.vldb.org/pvldb/1/1453965.pdf
28. Wilkinson, K.: Jena property table implementation (2006)
29. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. PVLDB **6**(4), 265–276 (2013). https://doi.org/10.14778/2535570.2488333, http://www.vldb.org/pvldb/vol6/p265-zeng.pdf
30. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In: Jensen, C.S., Jermaine, C.M., Zhou, X. (eds.) 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013. pp. 565–576. IEEE Computer Society (2013). https://doi.org/10.1109/ICDE.2013.6544856, https://doi.org/10.1109/ICDE.2013.6544856
31. Zou, L., Özsu, M.T., Chen, L., Shen, X., Huang, R., Zhao, D.: gstore: a graph-based SPARQL query engine. VLDB J. **23**(4), 565–590 (2014). https://doi.org/10.1007/s00778-013-0337-7, https://doi.org/10.1007/s00778-013-0337-7

7 Appendix

7.1 Queries from experiments

Query 1: Linear

```
SELECT ?v0 ?v2
WHERE {
?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic179> .
?v0 <http://schema.org/caption> ?v2 .
}
```

Query 2: Linear

```
SELECT ?v0 ?v2
WHERE {
?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic183> .
?v0 <http://schema.org/caption> ?v2 .
}
```

Query 3: Linear

```
SELECT ?v0 ?v2
WHERE {
?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic84> .
?v0 <http://schema.org/caption> ?v2 .
}
```

Query 4: Linear

```
SELECT ?v0 ?v2
WHERE {
?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic84> .
?v0 <http://schema.org/caption> ?v2 .
}
```

Query 5: Linear

```
SELECT ?v0 ?v1
WHERE {
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> <http://db.uwaterloo.ca/~galuc/wsdbm/Website28> .
}
```

Query 6: Star

```
SELECT ?v0 ?v2 ?v3 ?v4
WHERE {
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory5> .
?v0 <http://schema.org/caption> ?v2 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v3 .
?v0 <http://schema.org/publisher> ?v4 .
}
```

Query 7: Star

```
SELECT ?v0 ?v2 ?v3
WHERE {
?v0 <http://xmlns.com/foaf/age> <http://db.uwaterloo.ca/~galuc/wsdbm/AgeGroup2> .
?v0 <http://xmlns.com/foaf/familyName> ?v2 .
?v3 <http://purl.org/ontology/mo/artist> ?v0 .
?v0 <http://schema.org/nationality> <http://db.uwaterloo.ca/~galuc/wsdbm/Country1> .
}
```

Query 8: Star

```
SELECT ?v0 ?v1 ?v2
WHERE {
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v1 .
?v0 <http://schema.org/text> ?v2 .
<http://db.uwaterloo.ca/~galuc/wsdbm/User506> <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v0 .
}
```

Query 9: Star

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
WHERE {
?v0 <http://purl.org/goodrelations/includes> ?v1 .
<http://db.uwaterloo.ca/~galuc/wsdbm/Retailer10> <http://purl.org/goodrelations/offers> ?v0 .
?v0 <http://purl.org/goodrelations/price> ?v3 .
?v0 <http://purl.org/goodrelations/serialNumber> ?v4 .
?v0 <http://purl.org/goodrelations/validFrom> ?v5 .
?v0 <http://purl.org/goodrelations/validThrough> ?v6 .
?v0 <http://schema.org/eligibleQuantity> ?v7 .
?v0 <http://schema.org/eligibleRegion> ?v8 .
?v0 <http://schema.org/priceValidUntil> ?v9 .
}
```

Query 10: Star

```
SELECT ?v0 ?v2 ?v3 ?v4
WHERE {
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory12> .
?v0 <http://schema.org/caption> ?v2 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v3 .
?v0 <http://schema.org/publisher> ?v4 .
}
```

Query 11: Snowflake

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6
WHERE {
?v0 <http://purl.org/goodrelations/includes> ?v1 .
<http://db.uwaterloo.ca/~galuc/wsdbm/Retailer3> <http://purl.org/goodrelations/offers> ?v0 .
?v0 <http://purl.org/goodrelations/price> ?v3 .
?v0 <http://purl.org/goodrelations/validThrough> ?v4 .
?v1 <http://ogp.me/ns#title> ?v5 .
?v1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v6 .
}
```

Query 12: Snowflake

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6
WHERE {
?v0 <http://purl.org/goodrelations/includes> ?v1 .
<http://db.uwaterloo.ca/~galuc/wsdbm/Retailer11> <http://purl.org/goodrelations/offers> ?v0 .
?v0 <http://purl.org/goodrelations/price> ?v3 .
?v0 <http://purl.org/goodrelations/validThrough> ?v4 .
?v1 <http://ogp.me/ns#title> ?v5 .
?v1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v6 .
}
```

Query 13: Snowflake

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5
WHERE {
?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic61> .
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v2 .
?v3 <http://schema.org/trailer> ?v4 .
?v3 <http://schema.org/keywords> ?v5 .
?v3 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v0 .
?v3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory2> .
}
```

Query 14: Snowflake

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5
WHERE {
?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic232> .
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v2 .
?v3 <http://schema.org/trailer> ?v4 .
?v3 <http://schema.org/keywords> ?v5 .
?v3 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v0 .
?v3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory2> .
}
```

Query 15: Snowflake

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7
WHERE {
?v0 <http://xmlns.com/foaf/homepage> ?v1 .
?v0 <http://ogp.me/ns#title> ?v2 .
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v3 .
?v0 <http://schema.org/caption> ?v4 .
?v0 <http://schema.org/description> ?v5 .
?v1 <http://schema.org/url> ?v6 .
?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/hits> ?v7 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre24> .
}
```

Query 16: Snowflake

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7
WHERE {
?v0 <http://xmlns.com/foaf/homepage> ?v1 .
?v0 <http://ogp.me/ns#title> ?v2 .
?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v3 .
?v0 <http://schema.org/caption> ?v4 .
?v0 <http://schema.org/description> ?v5 .
?v1 <http://schema.org/url> ?v6 .
?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/hits> ?v7 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre59> .
}
```

Query 17: Complex

```
SELECT ?v0
WHERE {
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v2 .
?v0 <http://purl.org/dc/terms/Location> ?v3 .
?v0 <http://xmlns.com/foaf/age> ?v4 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v5 .
?v0 <http://xmlns.com/foaf/givenName> ?v6 .
}
```

Query 18: Complex

```
SELECT ?v0 ?v4 ?v6 ?v7
WHERE {
?v0 <http://schema.org/caption> ?v1 .
?v0 <http://schema.org/text> ?v2 .
?v0 <http://schema.org/contentRating> ?v3 .
?v0 <http://purl.org/stuff/rev#hasReview> ?v4 .
?v4 <http://purl.org/stuff/rev#title> ?v5 .
?v4 <http://purl.org/stuff/rev#reviewer> ?v6 .
?v7 <http://schema.org/actor> ?v6 .
?v7 <http://schema.org/language> ?v8 .
}
```

Query 19: Complex

```
SELECT ?v0
WHERE {
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v2 .
?v0 <http://purl.org/dc/terms/Location> ?v3 .
?v0 <http://xmlns.com/foaf/age> ?v4 .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v5 .
?v0 <http://xmlns.com/foaf/givenName> ?v6 .
}
```

Query 20: Complex

```
SELECT ?v0 ?v4 ?v6 ?v7
WHERE {
?v0 <http://schema.org/caption> ?v1 .
?v0 <http://schema.org/text> ?v2 .
?v0 <http://schema.org/contentRating> ?v3 .
?v0 <http://purl.org/stuff/rev#hasReview> ?v4 .
?v4 <http://purl.org/stuff/rev#title> ?v5 .
?v4 <http://purl.org/stuff/rev#reviewer> ?v6 .
?v7 <http://schema.org/actor> ?v6 .
?v7 <http://schema.org/language> ?v8 .
}
```