Computer Science and Information Systems20(4):1311–1341 https://doi.org/10.2298/CSIS230225046Y

SRDF_QDAG: An Efficient End-to-End RDF Data Management when Graph Exploration Meets Spatial Processing

Houssameddine Yousfi[1,2], Amin Mesmoudi[3], Allel Hadjali[1], Houcine Matallah[2], and Seif-Eddine Benkabou[3]

[1] LIAS, ENSMA Engineering School
1 avenue Clément Ader, 86961 Futuroscope Chasseneuil Cedex, France
{houssameddine.yousfi, allel.hadjali}@ensma.fr
[2] LRIT, Science Faculty
University Abu Bekr Belkaid, Tlemcen, Algeria
{houssameddine.yousfi, houcine.matallah}@univ-tlemcen.dz
[3] LIAS, University of Poitiers
15 rue de l'Hôtel Dieu 86073 POITIERS Cedex 9, France
{amin.mesmoudi, seif.eddine.benkabou}@univ-poitiers.fr

Abstract. The popularity of RDF has led to the creation of several datasets (e.g., Yago, DBPedia) with different natures (graph, temporal, spatial). Different extensions have also been proposed for SPARQL language to provide appropriate processing. The best known is **GeoSparql**, that allows the integration of a set of spatial operators. In this paper, we propose new strategies to support such operators within a particular TripleStore, named RDF_QDAG, that relies on graph fragmentation and exploration and guarantees a good compromise between scalability and performance. Our proposal covers the different TripleStore components (Storage, evaluation, optimization). We evaluated our proposal using spatial queries with real RDF data, and we also compared performance with the latest version of a popular commercial TripleStore. The first results demonstrate the relevance of our proposal and how to achieve an average gain of performance of 28% by choosing the right evaluation strategies to use. Based on these results, we proposed to extend the RDF_QDAG optimizer to dynamically select the evaluation strategy to use depending on the query. Then, we show also that our proposal yields the best strategy for most queries.

Keywords: RDF, Graph Data, Spatial Data, TripleStore, Graph exploration, Optimization.

1. Introduction

Since Google popularized the use of the term Knowledge Graph to designate all knowledge used by its search engine, the number of this type of dataset has not stopped increasing. Knowledge Graphs (KG) are labeled and directed multi-graphs that encode information in the form of entities and relationships relevant to a specific domain or organization. KGs are effective tools for capturing and organizing a large amount of structured and multi-relational data that can be explored using query mechanisms. Given these characteristics, KGs become the backbone of the Web and existing information systems in different academic fields and industrial applications. Their power comes from their ability to extend the existing knowledge without affecting the previous ones.

Following this large gain in popularity of knowledge graphs, the need for a standard data representation format has become obvious. This is especially in the context of the semantic Web due to its vision of globally accessible and linked data on the internet. In order to meet that need, RDF (Resource Definition Framework) has been proposed as the main standard for the semantic Web. In RDF format, Data are represented logically by a graph-based structure. The advantage of such a representation relies on the fact that it is schema-less, making it flexible and easy to adopt in different application domains. Moreover, such flexibility makes RDF more suitable for fast changing data where normalization is not possible or impractical due to the frequent changes to the underlying schema.

RDF Data is structured using the concept of triples $< subject, predicate, object >$ where the object can be a literal of predetermined types (string, double, ...) or it can be the subject of another triple leading to a graph structure. In the context of RDF data, SPARQL has been proposed as a query language. Since RDF is a graph representation of data, the query is composed mainly of a sub-graph where some subjects, predicates, or objects are replaced with variables. This sub-graph is called a basic graph pattern (BGP). Answering a SPARQL query is equivalent to finding sub-graphs that match the query pattern. On top of the basic graph pattern (BGP) matching, it is possible to run filters on variables such as Boolean expressions and regular expressions. The standard W3C norm of SPARQL does not provide the possibility to express spatial filters. However, many extensions have been proposed to improve the expressiveness of SPARQL, making it able to express spatial filters. The most known extensions are GeoSparql [4] and stSparql [19].

Many attempts have been proposed to implement OGC GeoSPARQL [4] by the community. Such implementation is a hard task since it requires changes on many levels of the triples store (storage, indexing, evaluation engine and optimizer). The changes also depend on the type and architecture of the Triplestore. For example, strategies that work on a Triplestore based on a single table strategy (eg. 3-Store[14]) may not work on another based on property table (eg. Jena[36]).

Some of the existing Triplestores are capable of answering Spatial-RDF queries with variant capabilities. Most of them are based on the relational model, whereas, others are based on single table or fact strategies. Nevertheless, all the previously mentioned approaches suffer from a high number of joins, which leads to performance and scalability problems. Recently in [17], the RDF_QDAG Triplestore was proposed. It relies on graph fragmentation and exploration, making it able to offer a better performance and scalability compromise. In this work, we take advantage of such capabilities and expand the system to be able to handle spatial data without the loss of this trade-off. The proposed extensions are, to the best of our knowledge, the first that provide spatial-RDF data processing in a graph exploration system. We managed throw the proposed approaches to achieve an average gain of performance estimated at 28%.

In this paper, we discuss the extension of RDF_QDAG in order to add the support of spatial operators and filters proposed in GeoSPARQL. The contributions of this paper can be summarized as follows:

- Two evaluation approaches for spatial-RDF data (Spatial-First and BGP-First) are proposed.
- An existing spatial indexing approach [22] is adapted to be compatible with the graph exploration logic in RDF_QDAG triplestore. This indexing approach is used in the Spatial First strategies.
- The effect of the query evaluation strategies and the optimization techniques on the performance of RDF_QDAG, is studied in depth.
- An optimizer capable of selecting the best available evaluation strategy based on the query and statistics about the RDF and spatial data, is developed.
- Finally, an extensive experimental evaluation of the proposed approaches is conducted and compared with a well known and used commercial Triplestore.

The rest of this paper is organized as follows. First, we provide a comprehensive review of related work in section 2. Then, we state the basic concepts in section 3. After that, we explain the proposed query evaluation strategies with the help of running examples in section 4. In section 5, we address the optimization issue and establish the basis of an optimizer capable of choosing the best evaluation strategy based on the available statistics and metadata. To validate the proposed approaches, in section 6, we discuss the results of the experimental validation performed. Finally, we conclude the paper and list some future perspectives.

2. Related Work

In this section, we provide a critical review of main related work. We have divided this review in three parts: (i) work related to RDF data processing ; (ii) work related to Spatial data processing ; and (iii) work related to Spatial-RDF data processing.

2.1. RDF Data Processing

One can summarize the approaches dedicated to RDF data processing w.r.t. to their storage strategies of data. Four families of approaches can be distinguished (see Table 1 for a comparison):

1. The most intuitive way to store RDF data is by using a single big relational table that contains tree columns corresponding to the subject, predicate and object. This strategy is known as the single table strategy.
2. A second alternative storage option is the binary table. In this approach, for each property, the system stores a binary table containing the subject and the object. This approach is widely used for salable distributed systems [9].
3. The third approach is called ”Property table”. Indeed, subjects with common proprieties are grouped and stored in a large horizontal table. Each column in the table corresponds to a property.
4. In the fourth approach, RDF data is modeled and stored in its native graph form. Subjects and objects are considered as nodes, while properties are considered as labelled edges.

Table 1. Comparison of different storage strategies for RDF data, including examples of triplestores that utilize each strategy, their advantages and disadvantages.

Storage Strategy	**Triplestore Examples**	**Advantages**	**Disadvantages**
Single Table	Oracle, Sesame [8], 3-Store [14]	Intuitive	Large number of self joins needed for queries
Binary Table	SW-Store [3], C-store [35]	Suitable for distributed systems	Loss in performance with multiple properties, many tables needed for updates
Property Table	Jena [36], DB2RDF [5], 4store [15]	Efficient for queries with star patterns	Difficulties with chain queries, storage overhead due to null values, does not allow multiple values for the same property
Native Graph Form	Trinity [38], gStore, RDF_QDAG [17, 39]	Stores RDF data in its native form	N/A

As for the Triplestore RDF_QDAG [17], it stores RDF data in a graph form and it answers the queries using a graph exploration. For an efficient exploration, RDF_QDAG uses a combination of data partitioning and indexing techniques. First, the graph is partitioned in many fragments called Graph Fragments ($\mathcal{GF}$ for short). Each $\mathcal{GF}$ is then indexed using a clustered B+Tree. Similar to some existing works, RDF_QDAG keeps a separate dictionary of string values. The indices store only IDs rather than the strings. For more efficiency, RDF_QDAG makes use of two different orders SPO and OPS to store indices.

2.2. Spatial Data Processing

In this section, we focus on storing and indexing techniques of spatial data. Hereafter, the principle of each technique is presented (see also Table 2 for a comparison).

1. Grid files [29]: Partition the space into stripes alongside each dimension. The width of a strip can be variable and the number of stripes may differ for each dimension.
2. Kd-trees and kdb-trees [27][33]: Tree based structures that store data entries in the leafs. Each node in the Kd-tree splits the space along side one dimension (X for example) constructing two children. Each child node splits the space in the other dimension (Y in this case). We keep cycling throw dimensions on each layer until we reach the leafs.
3. Quad-trees [32][28]: They work as follows: each node recursively divide the space into four quadrants until each bucket (leaf node) has less objects then a given maximum capacity. During update operations, as soon as a bucket exceeds the given capacity threshold, a split operation is triggered.
4. R-trees [13, 18]: They rely on object grouping based on the construction of MBRs (Minimum Bounding Rectangles). To get a tree like structure, we keep grouping recursively MBRs inside each others to construct higher levels until we get one root for the tree. Properties are considered as labelled edges.

The processing of Big spatial data covers various domains and applications, including distributed computing frameworks, spatial data analysis and modeling, and spatial data quality. Notable contributions to this field include the work of Lee et al.[21], which provides an overview of the challenges and opportunities of processing Big spatial data. In addition, distributed computing frameworks such as SpatialHadoop[10] and GeoSpark[37] have been introduced for processing large-scale spatial data. More domain-specific approaches have also been proposed, such as Astroide[6], a scalable Spark-based processing engine for Big astronomical data, and the work of Papadopoulos et al.[25], which focuses on the challenges and opportunities of using Big data processing techniques for climate change research. Together, these works demonstrate the importance and impact of Big data processing for spatial data analysis and management.

Table 2. Storing and indexing techniques of spatial data.

Technique	**Advantages**	**Limits**
Grid files [29]	Good performance in certain applications.	Performance can be severely affected by a high number of dimensions. High cost of Balancing, Splitting or re-sizing a single cell Unsuited then for highly skewed data.
Kd-trees and kdb-trees [27][33]	Built efficiently in time complexity $O(n \log n)$. Offer efficient nearness search with time complexity $O(\log n)$	Hard to balance because the direction of split is different for each level
Quad-trees [32][28]	Good performance in nearness queries and KNN joins	Many empty nodes are stored in the form of chains (which can be solved by the external balanced regular (x-BR) trees). Unsuited for secondary storage due to low fan-out
R-trees [13, 18]	Lighter memory footprint. Rely on object grouping and not space partitioning	To answer a query, multiple sub-trees needs to be considered

2.3. Spatial-RDF data processing

In order to represent geographical linked data for the semantic Web, the Open Geospatial Consortium has proposed GeoSPARQL [4] as a norm that extends classic SPARQL. Many Triplestores have subsequently been subsequently extended to support the processing of this new standard. The spatial extension of RDF stores extensively depends on the storage model and the query evaluation engine.

For instance Strabon [20], an extension of Sesame [8], supports spatial data. It stores data in PostGIS. It implements a propriety table approach, where each table is indexed using SO and OS indices. Spatial data are saved on a separate relational table. This later is indexed using an R-tree [13]. The query optimizer extension of Strabon is simple. It relies on heuristics to push down spatial filters. Since Strabon is based on an old RDF store (i.e., Sesame), it lacks many optimization techniques used in modern Triplestores.

Table 3. Overview of different spatial extensions of RDF Triplestores.

Extension	**Underlying system**	**RDF Storage**	**Spatial storage**
Strabon [20]	Sesame [8]	Triple table in PostgreSQL	R-tree
Brodt et al. [7]	RDF-3X[24]	Heavy indexing	R-Tree
Geo-Store [34]	RDF-3X[24]	Heavy indexing	Grid file
Virtuoso [2]	RDBMS	N/A	N/A
Oracle	N/A	N/A	N/A
GraphDB [1]	N/A	N/A	N/A

Brodt et al. [7] extended RDF-3X [24] to support spatial data. In this work, the range selection operation is the only spatial operation supported leading to very limited extension. Moreover, the spatial filtering is also limited to either at the beginning of the query evaluation or at the end of the query. Geo-Store [34] is another spatial extension of RDF-3X. Geo-Store relies on a grid file to index the spatial data. The Hilbert space-filling curve is used to establish a global order for each cell on the grid. Each spatial object will be pared in the order of the cell that it resides in. An additional triple is added to the data graph in the following form: $< o, hasPosition, gridPosition >$. The additional triple leads to an additional join step while processing the queries. Note also that spatial RDF queries are also supported by many commercial systems, such as Oracle, Virtuoso [2], and GraphDB [1]. However, details about their internal design are inaccessible.

3. Background and Preliminaries

This section introduces the principle of Spatial DataBase Management Systems (SDBMS), some main formal definitions related to RDF graph management and an overview of the architecture of the RDF_QDAG Triplestore.

3.1. SDBMS Systems: A refresher

SDBMS systems are database systems that support spatial data types in their models, their query languages and provide efficient ways to process spatial operations [12]. The term spatial data types refers to every data object that contains coordinates in some multi-dimensional metric space. SDBMSs are considered as the basis of Geographic Information Systems (GIS) and many computer-assisted design systems (CADs).

Spatial information can be represented in two different ways: Raster or Vector [30]. Raster data refer to an array or a matrix where each cell (pixel) represents a rectangular region. Vectors represent object features in the form of geometric shapes. In this work, we focus on vector representation.

In addition to storing spatial data, SDBMs need to perform spatial operations to answer queries. Spatial queries can be categorized into nearness queries, region queries and join queries. Nearness queries allow to find objects close to a certain point. While region queries aim at finding objects that reside fully or partially in a certain region. Finally, the join queries are operations that allow to filter the Cartesian product of two datasets based on a given condition (i.e., Boolean expression). In the case where the condition implies a spatial operation, this join is called a spatial join.

In most geographic information systems, we rarely find only spatial data. We generally find other plane standard data types (i.e strings, doubles ...). These data types also need to be stored, queried and updated. A common way SDBMS fulfill this need is by using the relational model. The combination of the relational model with the spatial data has been well studied in the literature [11, 31]. However, combining spatial data with graph model has been little studied. The advantage of using a graph model is the ability to store data without passing by a schema, giving more flexibility for the information systems.

3.2. RDF graph formalisation

In this work, we propose a system capable of handling Spatial and graph data represented by means of RDF format. Data in RDF are represented using triples called SPO triples (subject, property, object). Subjects are identified by a Uniform Resource Identifier (URI). The property represents the relationship between the subject and the object. The object of a triple can be either the subject of another triple or just a simple data value called a literal. Following this format, we can represent data as a graph where the nodes are subjects/objects and the edges are the proprieties. Below, we provide some formal definitions on RDF graph necessary for the reading of the next sections.

Definition 1. *(RDF graph) An RDF graph is a four-tuple $G = \langle V, L_V, E, L_E \rangle$, where*

1. *V Is a collection of vertices that correspond to all subjects and objects in RDF data. The set V can be divided into V_l and V_e where V_l is the set of literal vertices and V_e is the set of entity vertices.*
2. *L_V is the set of vertex labels. The label of a vertex $u \in V_l$ is its literal value, and the label of a vertex $u \in V_e$ is its corresponding URI.*
3. *$E = \overrightarrow{u_1, u_2}$ is a collection of directed edges that connect the corresponding subject and objects.*
4. *L_E is a collection of edge labels. Given an edge $e \in E$, its edge label is its corresponding property.*

To query RDF data, we use SPARQL [26]. It is a query language that expresses queries using a basic graph pattern (BGP) containing variables. The answer to the query is the mappings of the variables where a sub graph from the data matches the graph pattern of the query. Filters can be added to the query in order to express some conditions on the graph elements. The following is a formal definition of a SPARQL query:

Definition 2. *(Query graph) A query graph is a five-tuple $Q = \langle V^Q, L_V^Q, E^Q, L_E^Q, FL \rangle$, where*

1. *$V^Q = V_e^Q \cup V_l^Q \cup V_p^Q$ is a collection of vertices that correspond to all subjects and objects in a SPARQL query, where V_p^Q is a collection of parameter vertices, and V_e^Q and V_l^Q are collections of entity vertices and literal vertices in the query graph Q respectively.*
2. *L_V^Q is a collection of vertex labels in Q. A vertex $v \in V_p^Q$ has no label, while that of a vertex $v \in V_l^Q$ is its literal value and that of a vertex $v \in V_e^Q$ is its corresponding URI.*

3. *E^Q is a collection of edges that correspond to properties in a SPARQL query. L_E^Q are the edge labels in E^Q.*
4. *FL are constraint filters, such as a wildcard constraint or a spatial constraint.*

When storing spatial data using the RDF format, spatial information is stored in the literals. As a consequence, to express spatial operations, it is necessary to use spatial functions in the filter part of the query. There are many extensions to the SPARQL language to support spatial filters. Here, we rely on the GeoSPARQL standard [4] defined by the Open Geospatial Consortium (OGC). It extends both RDF and SPARQL to express spatial information and queries. See also stSPARQL [19] for a similar set of features.

3.3. Architectural overview of RDF_QDAG

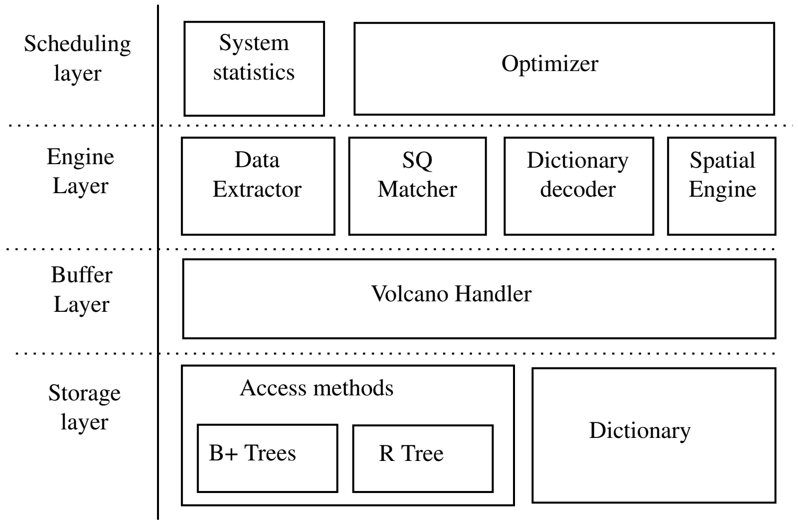


Fig. 1. Architectural overview of RDF_QDAG

RDF_QDAG [17] is composed of several layers. Each layer contains many components, as shown in figure 1. In this section, we present the overall architecture of the system and we detail the process of query evaluation.

Data storage The storage layer in RDF_QDAG is responsible for efficiently storing and accessing different types of data, mainly Graph and Spatial data. We note that data in RDF_QDAG support many native data types such as Strings, Integers, Doubles and more. To efficiently query all types of data, RDF_QDAG uses mainly three access methods B+tree, R-tree and a Dictionary.

The main storage of the graph data is the B+Tree. In order for the graph to be stored without losing the semantics that relies in the edges of the graph (Predicates in case of RDF), the graph should be partitioned into graph fragments while taking the connectivity between them into account. The ideal graph partitioning strategy is the one that maximizes inter-partition connectivity and minimizes intra-partition connectivity.

Each graph fragment is a grouping of Data Stars. Data stars extend the notion of tuple in the relational model. We define data stars formally as:

Definition 3. *(Data Star) Given a node x (named data star head) in a RDF graph G, a Data Star $DS(x)$ is the set of either triples sharing the same subject x, or the same object x. We name Forward Data Star and Backward Data Star the sets $\overrightarrow{DS}(x) = \{(x,p,o)|\exists_{p,o} : (x,p,o) \in G\}$ and $\overleftarrow{DS}(x) = \{(s,p,x)|\exists_{s,p} : (s,p,x) \in G\}$ respectively.*

If we compare the notion of data star with the notion of tuple, the primary key of a tuple corresponds to the head x of a data star $DS(x)$. RDF_QDAG groups similar data stars into sets called *Graph Fragments* using characteristic sets [23]

Each subject s in the graph G has a characteristic set defined as $\overrightarrow{cs}(s) = \{p|\exists_o : (s,p,o) \in G\}$. Similarly, for objects, $\overleftarrow{cs}(o) = \{p|\exists_s : (s,p,o) \in G\}$. A forward graph fragment $\overrightarrow{Gf}$ groups forward data stars with the same characteristic set. The backward graph fragments $\overleftarrow{Gf}$ are formed identically. The formal definition of this concept is given in definition 4

Definition 4. *(Graph Fragment) A Graph Fragment is a set of Data Stars. It is named a Forward Graph Fragment $\overrightarrow{Gf}$ if it groups Forward Data Stars such that:*

$$\overrightarrow{Gf} = \{\overrightarrow{DS}(x)|\forall_{i \neq j}\overrightarrow{cs}(x_i) = \overrightarrow{cs}(x_j)\}.$$

Likewise, a Backward Graph Fragment $\overleftarrow{Gf}$ is defined as

$$\overleftarrow{Gf} = \{\overleftarrow{DS}(x)|\forall_{i \neq j}\overleftarrow{cs}(x_i) = \overleftarrow{cs}(x_j)\}.$$

Once the graph is partitioned into graph fragments, each fragment is loaded into an index. The index used in the case of RDF_QDAG is a B+tree. The efficiency of this type of index is well studied for this type of graph data [17]. Also compression techniques are used to optimize the space usage for storage and the number of pages loaded into the buffers while evaluating queries.

In the context of optimizing space usage, Subject and Object of the graph that are of type String or URI are replaced with an ID. Otherwise, the size of the fragments will be significant. Especially since the values of subjects/objects may figure many times in the fragments. However, this technique necessitates a dictionary to store $< value, ID >$ combinations. Moreover, an additional encoding and decoding step is required to evaluate each query.

The third and the last access method is a spatial access method (namely R-tree) that we added in the context of this work as an extension of RDF_QDAG to support spatial queries. More details on spatial indexing are in section 4.

Scheduling Layer The main component of the scheduling layer is the optimizer. The optimizer has the role of selecting the best execution plan for a given query. This process is divided into two steps: (i) plan enumeration and (ii) cost estimation. The plan with the lowest estimated cost is the one chosen by the optimizer for evaluation.

The nature of the plan depends on the system design. In classical systems, a plan can be considered as a sequence of join operations on triple patterns. However, in RDF_QDAG, the notion of data star is proposed as an equivalent of tuple in the relational model. In a similar fashion, the notion of a star query is proposed. Indeed, triple patterns with the same Subject or Object are grouped together as a forward or a backward data star.

Definition 5. *(Query Star) Let Q be the SPARQL query graph. A Forward Query Star $\overrightarrow{QS}(x)$ is the set of triple patterns such that $\overrightarrow{QS}(x) = \{(x,p,o) | \exists_{p,o} : (x,p,o) \in Q\}$, x is named the head of the Query Star. Likewise, a Backward Query Star $\overleftarrow{QS}(x)$ is $\overleftarrow{QS}(x) = \{(s,p,x) | \exists_{s,p} : (s,p,x) \in Q\}$. We use $\overrightarrow{QS}, \overleftarrow{QS}$ to denote the set of forward and backward query stars and qs to denote indistinctly a forward and backward query star.*

An execution plan is an order function applied on a set of Query Stars and Filter Unites. The function denotes the order in which the mappings for each Query Star will be found and the order in which the filter unit will be evaluated

Definition 6. *(Execution Plan) . We denote by $\mathcal{P} = [QS_1, QS_2, Fu_1(p_1,p_2), ..., QS_n]$ the plan formed by executing QS_1, then QS_2, then evaluating the filter unit $Fu_1(p_1,p_2)$ which requires the mappings of p_1 and p_2 parameters.*

Engine Layer The engine layer is the layer responsible of evaluating the query. More precisely, it is responsible of evaluating the optimal plan provided by the optimizer.

The evaluation of a Query Star consists of finding matches between the variables of the Query Star and the nodes of the data graph. For each triple in the star, we seek the set of mappings, that satisfies it. Next, we merge the mappings related to the triples to build the Query Star matches.

Definition 7. *(Star Query Evaluation) The evaluation of a Query Star $QS(x)$ against the graph G is formally defined as follows:*

$$[\![QS(x)]\!]_G := \{[\![tp_1]\!]_G \bowtie [\![tp_2]\!]_G \bowtie ... \bowtie [\![tp_n]\!]_G | n = card(QS(x))\}$$

where:

$$[\![tp_i]\!]_G \bowtie [\![tp_j]\!]_G = \{\mu_l \cup \mu_r | \mu_l \in [\![tp_i]\!]_G \text{ and } \mu_r \in [\![tp_j]\!]_G, \mu_l \sim \mu_r \text{ and } \mu_l(tp_i) \neq \mu_r(tp_j)\}$$

We denote that a mapping μ is a function $V_p^Q \rightarrow V^G$. Given two mappings μ_1 and μ_2, $\mu_1 \sim \mu_2 \Rightarrow \mu_1(?x) = \mu_2(?x)$.

Based on the previous definitions, we can determine the evaluation of a query using the set of query stars, as follows:

Definition 8. *(Query Evaluation) Given a set of stars, $\{qs_1, qs_2, ..., qs_n\}$, that cover the query, $Triples_q(qs_1) \cup Triples_q(qs_2) \cup ... \cup Triples_q(qs_n) = Triplets(q)$, the evaluation of the BGP part of the query q using the set of query stars is defined as follows:*

$[\![q]\!]_G = \{\mu : \forall \mu \in [\![qs_1]\!]_G \bowtie [\![qs_2]\!]_G \bowtie ... \bowtie [\![qs_n]\!]_G\}$

We can also set the query BGP evaluation based on fragments, as follows:

$[\![q_g]\!]_G = \{\mu : \forall\mu \in \bigcup_{Gf \models qs_1} [\![qs_1]\!]_{Gf} \bowtie \bigcup_{Gf \models qs_2} [\![qs_2]\!]_{Gf} \bowtie ... \bowtie \bigcup_{Gf \models qs_n} [\![qs_n]\!]_{Gf}\}$
Where $Gf \models qs$ *iff* $cs(qs) \subset cs(Gf)$

The full evaluation of the query is the evaluation of the BGP part and the filters FL *and it is defined as follows*
$[\![q_g]\!]_G = \{\mu : \forall\mu \in [\![qs_1]\!]_G \bowtie [\![qs_2]\!]_G \bowtie ... \bowtie [\![qs_n]\!]_G | \mu \models FL\}$

```
SELECT ?p
WHERE {
?p <hasArea> ?a .
?p <isLocatedIn> ?l .
?l <hasGeometry> ?g .
};
```

Listing 1.1. Example of simple RDF query (Q_1)

An execution plan $\mathcal{P}$ is called an Acceptable Execution Plan if it fulfills the following conditions:

1. *Coverage:* All nodes and predicates of the given query are *covered* by the set of Query Stars of the plan.In the case of Query Q_1 the execution plan $[\overleftarrow{?l}, \overrightarrow{?p}]$ is not an acceptable plan since it does not cover the edge $< hasGeometry >$ and the variable $?g$.
2. *Instantiated head:* This condition guarantees that for a plan $\mathcal{P} = [QS_1, ... , QS_n]$, $\forall_{i>1} QS$, the head of the QS_i must be already instantiated. We use this condition to avoid to a Cartesian product when mappings are exchanged between two star queries. For example, in the case of Query Q_1 the execution plan $[\overleftarrow{?l}, \overleftarrow{?g}, \overrightarrow{?p}]$ is not an acceptable plan since the mapping of $?g$ is not yet available for the second $\overleftarrow{?g}$ to be evaluated. In this case the instantiated head condition is not satisfied.

The formal definition of an Acceptable Plan is given in Proposition 9.

Definition 9. *(Acceptable Plan)* $\mathcal{AP}$ *Let us consider* Q *as a given query,* $\overrightarrow{QS}$ *and* $\overleftarrow{QS}$ *as the sets of forward and backward graph star queries respectively,* T *has the set of triple patterns and the following functions:*

- *Tr:* $Q \cup \overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow T$ *It returns the set triple patterns of a query star or a query.*
- *Nd:* $\overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow V$ *It returns the nodes of a query star (subject or object).*
- *Head:* $\overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow V$ *a function that returns the head of a query star.*

An ***acceptable plan*** $\mathcal{AP}$ *is a tuple* $< X, f >$ *where* $X \subset \overrightarrow{QS} \cup \overleftarrow{QS}$ *and* $f : X \rightarrow \{1...|X|\}$ *is the query stars order function such that:*

1. $\bigcup_{QS \in X} Tr(QS) = Tr(Q)$
2. $\forall i \in \{2...|X|\}, Head(f^{-1}(i)) \in \bigcup_{j=1}^{i-1} Nd(f^{-1}(j))$

4. Query Evaluation Strategies

In this section, we present two evaluation strategies for Geo-SPARQL queries which are implemented in RDF_QDAG. In order to better illustrate those strategies, we show the processing of the example query Q2 on the dataset D1.

```
PREFIX gv: <http://geovocab.org/geometry#>
PREFIX ogis: <http://www.opengis.net/ont/geosparql#>

select ?g
where {
    ?o type "cultural".
    ?o gv:geometry ?p.
    ?p ogis:asWKT ?g.
    FILTER( bif:st_intersects(
        bif:st_geomfromtext( "POLYGON((7 43, 8 43,
        8 44, 7 44, 7 43))" ), ?g ) )
};
```

Listing 1.2. Example of spatial selection query (Q_2)

Table 4. Example of RDF triples (dataset D1).

Subject	Predicate	Object
Tennis Championship	hostedIn	Paris
Tennis Championship	type	Sports
Tennis Championship	geometry	G1
G1	asWKT	Point(2.34 48.85)
Festival of Lights	hostedIn	Lyon
Festival of Lights	type	Cultural
Festival of Lights	geometry	G2
G2	asWKT	Poit(4.846 45.75)
Film Festival	hostedIn	Cannes
Film Festival	Type	Cultural
Film Festival	geometry	G3
G3	asWKT	Point(7.012 43.55)

It is worth noticing that the existing formal framework for query plan and query evaluation do not take the filters into consideration. Previous contributions have focused on the graph matching aspect of the query evaluation. The filters were considered as an implementation detail. However, to introduce support for spatial filters, the existing formal definitions need to be extended to consider the spatial operators used in the filter clause of the query.

As we mentioned in definition 2, the Filter function FL is a truth function. We then express this function in a conjunctive normal form. We also introduce the concept of filter units as the operands of the mentioned conjunction.

Definition 10. *(Filter Unit)*

Let P be a subset of query parameters $P \in \mathcal{P}(V_p^Q)$. and qs_p the parameters of the star query qs. A filter is a truth function $FL : [\![q_g]\!]_G \rightarrow \{0,1\}$. Filter function can be expressed as a conjunction of operands. We name each operand a filter unit Fu.

$$FL = Fu_1 \wedge Fu_2 \wedge ... \wedge Fu_n$$

Using this concept of filter units Fu, one can see that the definition of an execution plan is extended. In the previous definition, the execution plan is a sequence of star query evaluation. While this is sufficient to perform the graph matching, it is not enough to consider the filters. In the new definition of the plan, we consider two types of operators: the classical star query evaluation and the new filter unit evaluation.

Definition 11. *(Execution Plan - extended definition) .Let $\mathcal{P}$ be a tuple $< X, f >$ where $X \subset \overrightarrow{QS} \cup \overleftarrow{QS} \cup FL$ and $f : X \rightarrow \{1...|X|\}$ is the query stars order function.*

We denote by $\mathcal{P} = [QS_1, QS_2, Fu_1(p_1, p_2), ..., QS_n]$ the plan formed by executing QS_1, then QS_2, then evaluating the filter unit $Fu_1(p_1, p_2)$ which requires the mappings parameters p_1 and p_2.

As mentioned before, to ensure a graph exploration logic, not all plans are acceptable. An execution plan is considered acceptable if, starting from the second star query, the head of the star is already instantiated. In a similar fashion, the position of the filter unit is critical. We can execute a filter unit only if the mappings for the parameters of the filter units are already available. On this principle, we extend the definition of an acceptable plan using the following condition:

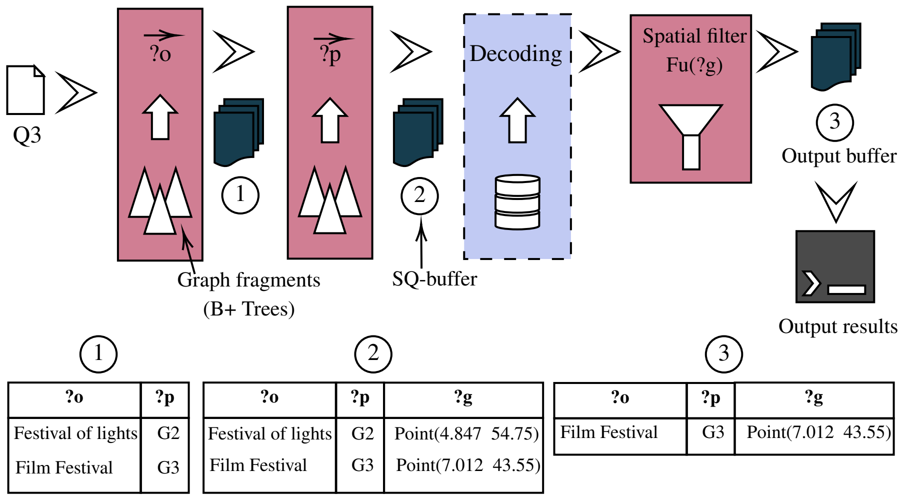
Definition 12. *(Instantiated filter unit parameter) Let consider the function Param: $FU \rightarrow V_p$ a function that returns the parameters of a Filter Unit. An acceptable plan $\mathcal{AP}$ is a tuple $< X, f >$ where $X \subset \overrightarrow{QS} \cup \overleftarrow{QS} \cup FL$ and $f : X \rightarrow \{1...|X|\}$ is the query stars order function such as $\forall i \in \{2...|X|\}, Param(f^{-1}(i)) \in \bigcup_{j=1}^{i-1} Nd(f^{-1}(j))$.*

To provide a better explanation of the concept of an acceptable plan, let's consider query Q_2, which contains the following star queries in the BGP: $\overleftarrow{?g}$, $\overrightarrow{?p}$, $\overleftarrow{?p}$, $\overrightarrow{?o}$, and $\overleftarrow{?o}$. The filter function consists of a single filter unit, $Fu(?g) = ?g \neg DC" POLYGON((-100...20))"$. There are many possible execution plans for this query, but not all of them are acceptable. For example, the plan $[\overrightarrow{?o}, \overleftarrow{?g}, Fu(?g)]$ is not acceptable because we need mappings of $?g$ to be able to evaluate $\overleftarrow{?g}$. The existence of such mappings is mandatory for all star queries except the first one. This is an example of a plan that does not satisfy the instantiated head condition explained in Section 3. Additionally, the plan $[\overrightarrow{?o}, Fu(?g), \overrightarrow{?p}]$ is not acceptable because, after evaluating $\overrightarrow{?o}$, the mappings of $?g$ are not yet available to process the spatial filter $Fu(?g)$. In this case, the condition of instantiated filter unit condition (definition 12) is not satisfied.

An example of an acceptable plan of the query $Q2$ is $[\overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$ or $[\overleftarrow{?g}, Fu(?g), \overleftarrow{?p}]$. To evaluate acceptable plans, two strategies are discussed: BGP-First strategy and Spatial-First strategy.

4.1. BGP-First strategy

This strategy consists of finding matches for the graph pattern first, before proceeding to run the filter on the results of the matching process. An example of a plan where this strategy can be considered is the following $AP_1 = [\overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$.



1

?o	?p
Festival of lights	G2
Film Festival	G3

2

?o	?p	?g
Festival of lights	G2	Point(4.847 54.75)
Film Festival	G3	Point(7.012 43.55)

3

?o	?p	?g
Film Festival	G3	Point(7.012 43.55)

Fig. 2. The execution of an BGP-First plan

The sequence of star queries and filter units listed in the logical execution plan does not consider implementation details. Therefore, we illustrate the full execution in figure 2. First, the graph matching part of the query is evaluated. Appropriate graph fragments are considered for evaluating each star query. Data in each fragment is stored in a B+tree in order to efficiently retrieve it from the disk. Once the information needed is retrieved, it is placed in a buffer, named SQ-buffer, so it can be used by the following operator in the plan.

The same logic is applied to spatial values. The true objects shapes can be significantly large depending on the geometry of the object (values describing Polygons are larger than values describing points for example) and on the resolution used to represent the object. To keep the size of the database low, and to maintain system performance, true shapes are stored in the dictionary.

Once the shapes are retrieved from the dictionary, the filter function FL is evaluated. In the case of $Q2$, the filter function is composed of a single filter unit $Fu(?g)$. This latter is evaluated in two steps (filter and refine). The Algorithm 1 is an example of an intersection filter without any loss of generalization to other region connection calculus operations. In the filter step, only MBRs of the shapes are considered (line 4) to significantly reduce the search space. The refining step considers the full geometry (line 11 and 12) hence, it is computationally expensive. However, it is necessary to eliminate false positives from the previous step.

Algorithm 1: Intersection Filter (L, Qb, Q)

```
Data: $M$: List of mappings;
$s$: Spatial object;
$use\_true\_shape$: flag to use true shape;
Result: $Q$: The set of mappings that intersect $s$
$Q \leftarrow \emptyset$;
for $m \in M$ do
    $(MBR(m), m) \leftarrow decode(m)$;
    if $MBR(m) \neg DC MBR(s)$ then
        add $m$ to $Q$;
        continue;
    if $m$ is a point then
        continue;
    if $use\_true\_shape = false$ then
        continue ;
    $GEO_m \leftarrow$ parseGeometry(m);
    if $GEO_m \neg DC s$ then
        add $m$ to $Q$;
end
return Q;
```

4.2. Spatial-First strategy

The BGP-First strategy presented above can answer spatial-RDF queries and can be easily integrated into the execution model of RDF_QDAG. However, it has some limitations that we discuss in this section. In this section, we introduce the second proposed strategy Spatial-First.

When we consider the same example query $Q2$ with the same dataset $D1$, one can observe that multiple valid plans can be run to answer the query. We can list a few of them as an example: $[\overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$, $[\overleftarrow{cultural}, \overrightarrow{?o}, \overrightarrow{?p}, Fu(?g)]$, $[\overleftarrow{?p}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?o}]$. All the listed plans have a common problem. Since the filter unit relies on the execution of the previous query stars, values of the geometry need to be obtained from the dictionary. As a result, it is impossible to use any spatial access method to speed up the spatial filter evaluation.

In the Spatial-First strategy, we try to take advantage of a spatial access method. To do so, we can only consider execution plans that start with the spatial filter. In the case of query $Q2$, the plan we consider is the following $[Fu(?g), \overleftarrow{?g}, \overleftarrow{?p}, \overrightarrow{?o}]$. As before, the spatial filter is run using two steps. However, this time, the filtering step can benefit from the spatial index.

The structure of the spatial index we use is an R-tree with some modifications for better integration with RDF_QDAG. The R-tree stores only object approximations in the form of MBRs with the necessary information to continue the graph exploration. This ensures the efficiency of the first step of the spatial filter by minimizing the number of pages. The page size in the index is 16 Kb. The structure of the pages is demonstrated in the figure 3.

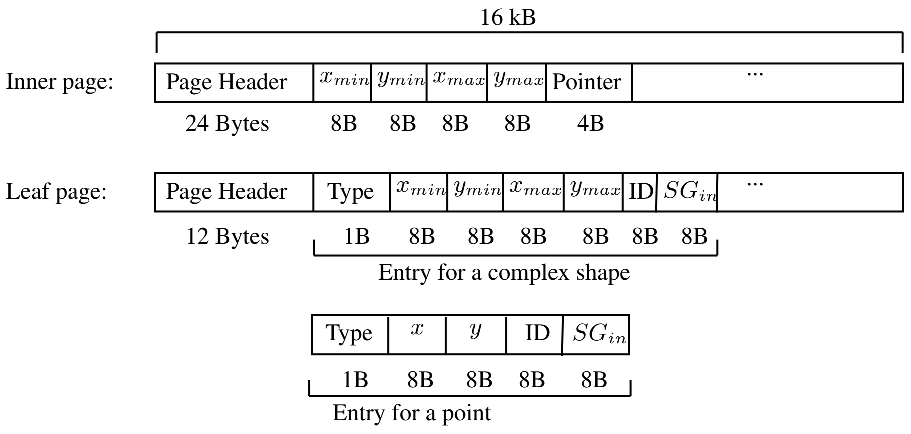


Fig. 3. The structure of index pages and entries

For inner pages, we save 24 bytes as page header. The rest is filled with inner entries where each entry is composed of an MBR (4 X 8 bytes) as a key and pointer to the appropriate page (4 bytes). An inner page can have up to 454 entries.

As for the leaf pages, we keep two types of entries: Points and MBRs. The MBRs are generally approximations of complex geometries. A point is represented by two coordinates (x, y) and an MBR is represented by four $(x_{min}, y_{min}, x_{max}, y_{max})$. On the leaf page, we save 12 bytes as page header, the rest is filled with leaf entries. For each entry, we store the object type in 1 byte, then we store the key, which is a point/MBR in 2*8/4*8 bytes, respectively, the object id in 8 bytes and the inward pointing fragment ID also in 8 bytes. The fragment ID is used to continue with the graph exploration. A leaf page can hold from 334 to 496 entries depending on the object types.

In the example shown in figure 4, only geometries $?g$ where $MBR(?g)\ \neg DC\ MBR(q)$ are returned after the exploration of the index. The next operator in the plan is standard graph exploration matchings.

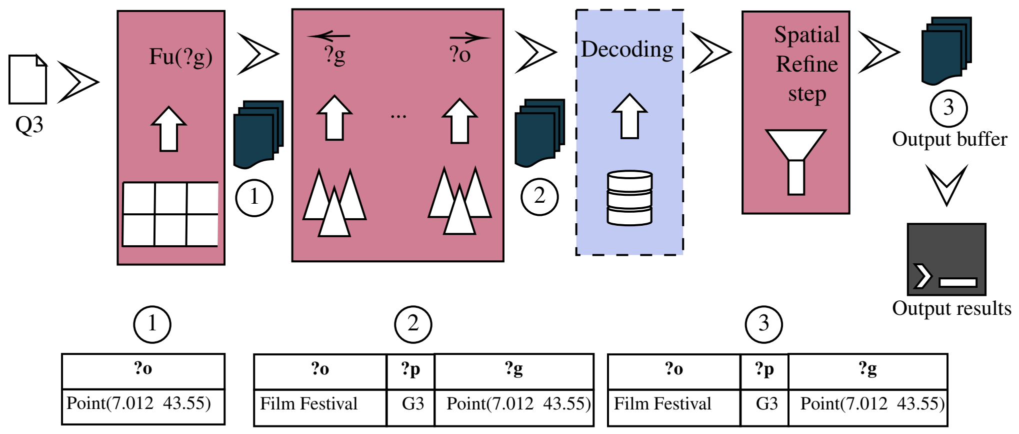
At the end of the evaluation, the decoding operation is performed to replace object IDs with the true value. The same is applied to spatial data where MBR approximation is replaced using the true geometries. Once the full shapes are available (true geometries) the refining step can be performed in the same way as in the BGP-First strategy.

5. Optimization Techniques

In this section, we present details about some optimization techniques that we propose to further improve execution time for both proposed strategies.

5.1. Query scheduling

A typical DBMS can answer the same query using different execution plans. All the plans provide the same results, however, the cost of execution for each plan is different. The



?o
Point(7.012 43.55)

?o	?p	?g
Film Festival	G3	Point(7.012 43.55)

?o	?p	?g
Film Festival	G3	Point(7.012 43.55)

Fig. 4. Execution of Spatial-First strategy

same logic applies for RDF_QDAG. In the case of the latter, an execution plan is a sequence of SQ and Filter units. Since the execution time can vary significantly from a plan to another, it is important to choose the best execution plan for a given query.

A traditional approach to select the best plan is to use two steps: plan enumeration and cost estimation. In the plan enumeration step, we list all the possible execution plans. However the number of execution plans can be very significant, so enumerating all the plans is either not possible or not efficient. Many DBMS use a heuristic approach to enumerate only the most promising plans. In the cost estimation step, we estimate the cost of executing each plan to select the plan with the lowest possible cost. This is generally done using dynamic programming since many plans share some parts between each other and it is not reasonable to recalculate the cost of the same plan segment multiple times.

RDF_QDAG uses the GOFast approach for the optimization [39]. In this approach, both the enumeration and estimation are performed in parallel. In order to do so, authors rely on a branch and bound algorithm. They start by constructing a tree where each node represents the accumulated cost of all previous operations and the edges represent plan operations. Naturally, the cost in the root is 0. The algorithm starts by estimating the cost of all possible first operations, then it expands on the operation with the lowest cost. GoFast continues on expanding the branch with the least cost until it gets a full execution plan.

Estimation in GoFast is based on the statistics collected for each graph fragment. The statistics also make it possible to reflect the interaction.

The existing GOFast optimization does not take into account the filters since the majority of the cost is caused by the graph exploration. However, this is not the case for the spatial filters since the cost of comparing complex shapes is high. On top of that, the use of an additional access method (R-tree) must be accounted for calculating the cost. Consequently, we extend the existing logic in order to take into account the cost of filter units. The cost of a plan is mainly the sum of cost of all star queries (both normal and spatial ones):

$$Cost(\mathcal{P}) = \sum_{qs \in \mathcal{P}} Cost(qs) \tag{1}$$

The estimation of the cost of star query is already part of RDF_QDAG system, however we changed it to be calculated in terms of triples not in terms of data stars. We opted for this change since the number of data star did not show (see appendix) a correlation with the choice of the best plan in the case of spatial queries contrary to the number of triples.

To estimate the full cost of the plan, for each part of the execution plan, two estimations needs to be done: estimation of the input and estimation of the number of results. This is necessary since the estimation of the cost of part of the plan depends on the number of results of the previous parts.

Estimation of the Number of Spatial Objects The estimation of the cost of a star query is already detailed in previous work [39], we will detail only the cost of the filter unit. In the case of an BGP-First plan, no spatial access method is used. In the case of Spatial-First plan, the cost of fu is the number of spatial objects that needs to be retrieved from the index:

$$Cost(fu) = SOC(Q) \tag{2}$$

$SOC(Q)$ is the number of spatial objects estimated using the spatial index. We can do this by taking advantage of the shallow depth of an R-tree. Indeed, since the fan-out is high, the depth is low (generally 4 to 5 layers maximum). In the estimation phase, we scan only the top layers of the R-Tree without loading the leaf layer. Naturally, we count only pointers where the attached key satisfies the filter. To calculate the number of objects ($SOC(Q)$) we simply multiply the number of leaf pages that satisfy the query by the average number of objects in a page. This assumption is based on the fact that most of the pages are close to 100% fill rate since the index is loaded using STR [22] and no updates are performed later. The only limitation of this estimation is the fact that not all objects in the leaf pages satisfy the query.

Estimation of spatial filter results .

The estimation of the number of results after the filter is necessary for the rest of the process. The number of objects $SOC(Q)$ can be considered as an estimation of the number of results since it is an estimation of objects where the MBR satisfies the spatial filter. However, to be able to continue calculating the cost with the GOFast approach for the rest of the plan, the total number of objects is insufficient. We need to calculate an estimation of the number of objects for each fragment.

The cost of a plan $\mathcal{P}$ is calculated in terms of the number of triples that need to be retrieved from the disk since the disk cost is the most important cost of the query. The cost of a particular plan is the sum of the cost of all star queries sq_i that compose the plan (equation 1). The cost of a star query is the number of triples retrieved from the relevant fragments fg_j:

$$Cost(qs) = \sum_{fg_j \in sq} Input\_Tr(fg_j, sq) \tag{3}$$

In the case of the first star query, no previous input is needed. As a consequence, the number of triples retrieved from a particular fragment fg_j is simply the number of triples in the fragment that satisfy the predicates of the star query:

$$Input\_Tr(fg_j, sq_1) = \#triples(fg_i, prd(sq_1)) \tag{4}$$

However for the rest of the star queries, the number of triples retrieved from a particular fragment fg_j is calculated using:

- $\#tripls(fg_j, pred(sq_i)$: the number of triples that satisfy the predicates of the star query sq_i
- $Input\_Ds(fg_j, sq_i)$: the number of data stars considered as in input
- $dist(fg_j)$: the number of data stars in the fragment fg_j

The formula for calculating the number of triples retrieved in case $i > 1$ is the following:

$$Input\_Tr(fg_i, sq_i) = \frac{\#triples(fg_i, pred(sqi)) * Input\_Ds(fg_)}{dist(fg_j)} \tag{5}$$

Detailed calculation of $Input_Ds(fg_j, sq_i)$ and $dist(fg_j)$ is found in Zouaghi et al[39] since we did not change it. As for the number of triples retrieved from a particular fragment it is the sum of all triples in the fragment where the predicate is the same as one of the star query predicates:

$$\#triples(fg_i, prd(sq_1)) = \sum_{P_j \in Pred(sq_i)} count(pj, fg) \tag{6}$$

In the case of Spatial-First plan, the number of triples is identical to the number of spatial objects estimated for each fragment :

$$Input\_tr(fg_i, SQ_1) = \#releventObject(fg_j/Q) \tag{7}$$

The number of spatial objects estimated for each fragment is estimated based on the selectivity of the spatial query as follows:

$$\#releventObject(fgj/Q) = size\_of(fg_j) * S\_select \tag{8}$$

Where $size\_of(fg_j)$ is the total number of triples in the fragmentfg_j and the spatial selectivity (S_select) is calculated as follows:

$$S\_select = \frac{SOC(Q)}{total\_spatial} \tag{9}$$

Where $total\_spatial$ is the total number of spatial objects stored in the index.

On top of the estimation of the number of relevant triples to read from disk, GoFast optimizer also relies on the number of results produced by each star query $outpu\_DS_{sqi}$ defined in [39] as follows:

$$output\_DS_{qs_i} = \{(Gf_j, p_i, k'')|p_i \in edges(qs_i) \wedge k'' = NDS_{p_i}\} \tag{10}$$

Where NDS_{p_i} is the number of data stars heads relevant to the predicate pi

However, we had to change the calculation of NDS_{p_i} to take into account the spatial filters. The new formula is the following:

$$NDS_{p_i} = \begin{cases} 1 & , if\ e.node\ is\ const \\ \frac{k'}{dist(Gf_j)} * dist\_NE(p_i, Gf_j) * S\_select & , otherwise \end{cases} \quad (11)$$

Where $dist\_NE(p_i, Gf_j)$ is the number of distinct nodes linked to the data star head in fg_j with respect to the predicate p_i.

With both estimations of the number of spatial objects and the spatial filter results, the GoFast optimizer can choose the best execution plan for Spatial-RDF queries.

5.2. Spatial pruning

Earlier, we proposed two execution strategies, "BGP-First" and "Spatial-First", of which only the latter can benefit from a spatial access method. The "BGP-first" strategy lacks spatial awareness at the beginning of the process, which means that it misses opportunities to reduce the search space based on spatial constraints. To address this issue, we propose a new optimization technique called "Spatial pruning".

As discussed in section 3, the initial RDF graph is partitioned into graph fragments $\mathcal{GF}$ for indexing and storage. When evaluating a query, only the necessary fragments are considered based on the characteristic sets of each fragment. However, when a query contains a spatial filter, many fragments that are considered due to their characteristic sets do not contribute to the final results. This is because the spatial filter in the query eliminates all the graph patterns produced by these fragments since they are connected to spatial objects that do not satisfy the filter.

To eliminate fragments that do not contribute to the results earlier in the process, we associate each graph fragment to an MBR such as all spatial objects connected to the fragment are situated inside this MBR. When processing the query, the optimizer do not choose fragment based on the graph part only, but also based on the spatial filter. If the MBR of a fragment ($MBR(Fg)$) satisfies the filter, it can contain the results. However, if the MBR does not satisfy the filter, it is immediately pruned and not considered while evaluating the query.

The proposed algorithm 2 operates on a set of star queries specified in a query plan. The algorithm iterates through each star query in the plan (line 3). For each star query, the relevant fragments are obtained based on the characteristic set (line 4). These fragments are then linked to the fragments of the previous star query using the function LinkToPreviousFragments() (line 5). If the current star query does not contain a spatial filter (line 6), the algorithm proceeds to the next star query (line 7). However, if the current star query contains a spatial filter (line 6), the algorithm loops through each fragment while testing the intersection of the fragment's Minimum Bounding Rectangle (MBR) with the query (line 9). If there is no intersection between the fragment's MBR and the query (line 10), the fragment and all fragments linked to it are removed from further consideration (line 11).

Algorithm 2: Spatial pruning

```
Data: $\mathcal{P}$: Execution Plan
      $\mathcal{GF}$: Set of graph fragments
      $SF : Gf_s \rightarrow MBR(Gf_s)$: List of spatial fragments MBRs
Result: $Gfs$ : qs $\rightarrow \mathcal{GF}_{Sq} | \mathcal{GF}_{Sq} \subseteq \mathcal{GF}$ :Set of fragment for each $Sq$
1  $Gfs \leftarrow []$;
2  $QS \leftarrow getQSList(P)$;
3  for $sq_i \in QS$ do
4      $CurrentFGs \leftarrow getCurrentFragments(sq_i)$;
5      $LinkToPreviousFragments(Gfs, CurrentGfs)$;
6      if $isSpatialFilter(qs_i) = false$ then
7          continue;
8      for $fg_i \in CurrentFGs$ do
9          $MBR_{fgi} \leftarrow SF.getMBR(fg_i)$;
10         if $MBR_{fgi} \neg DCs$ then
11             $Gfs.removeAllFGsConnectedTo(fg_i)$;
12     end
13 end
14 return $Gfs$;
```

6. Experimental evaluation

In this section we discuss several experimental results on the various approaches and optimisation techniques mentioned in the previous section. We also compare our proposed solution with a well-known commercial Triplestore Virtuoso.

6.1. Experimental setup and methodology

We perform several experiments on RDF_QDAG after integrating the approach and techniques proposed in this paper. RDF_QDAG is a project developed using Java and C++. The storage and access methods are developed using C++ and compiled using GCC version 7.5.0. The engine and optimizer are implemented using Java 11 and built using maven 3.8.6. For the run environment, we used Open JDK version 11.0.16. The system can be downloaded as a Docker image, which includes all necessary dependencies. The image is available on Docker Hub at https://hub.docker.com/r/qdag/rdf_qdag. In addition, a live demo of the system is provided on our project website at https://qdag.lias-lab.fr/.

All experiments were run on a machine equipped with Intel Xeon (Skylake, IBRS) @ 10x 2.295GHz and 64 GB of RAM and an SSD running Ubuntu 18.04 bionic with linux kernel x86_64 Linux 4.15.0-194-generic.

For the evaluation, we used the YAGO [16] knowledge base. YAGO is a real world data-set that contains more than 234 million facts on which 4 million are spatial objects.

All experiments are performed on a fresh install of the operating system. We clear page cache, dentry and inode cache before each query. Execution time is calculated from the submission of the query to the end of writing the results into an output file.

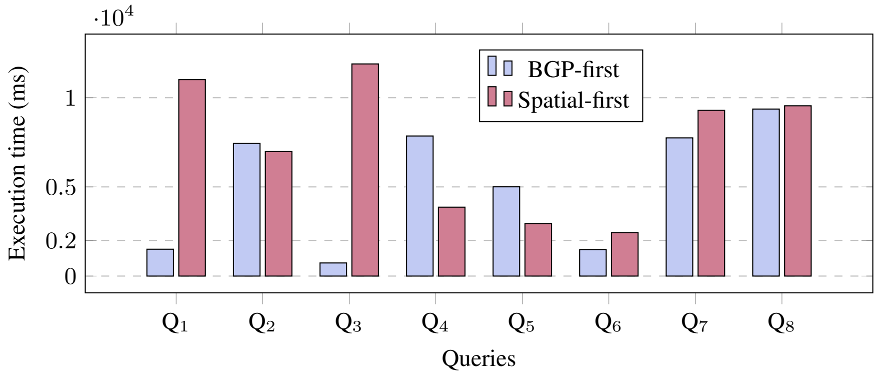


Fig. 5. Execution time (nanoseconds) of queries using both strategies BGP-first and Spatial-first.

Table 5. Execution time of queries on YAGO

Query	**Best BGP-First plan**			**Best Spatial-First plan**		
	Plan ID	**Execution time (ms)**	**# Triples**	**Plan ID**	**Execution Time (ms)**	**# Triples**
Q1	1	1506	**32503**	5	11014	463841
Q2	4	7442	238414	6	6981	**144671**
Q3	4	735	**4377**	5	11896	699942
Q4	3	7855	217526	2	3864	**91326**
Q5	4	5005	205310	6	2942	**60374**
Q6	1	1488	**10639**	3	2435	38092
Q7	3	7749	217526	2	9296	**56272**
Q8	1	9366	**369054**	3	9548	438063

6.2. Effect of evaluation strategies

To study the effect of evaluation strategies on the execution time, we ran several queries on the YAGO data-set.

The results of the execution time for queries using the BGP-First and Spatial-first strategies are shown in figure 5. Neither approach consistently outperforms the other, as demonstrated by the varying performance in queries Q_4, Q_2, and Q_5, where the Spatial-first approach is superior, and the remaining queries, in which BGP-first performs better.

To further investigate the factors contributing to the varying performance of each approach, we analyzed intermediary results in both the spatial and graph parts of the queries to extract the total number of triples loaded from the disk. The total number of triples is displayed in table 5. The results in the table show a clear correlation between the choice of the best execution strategy and the number of triples fetched from the disk. In each query, the strategy with the lowest number of triples is the best-performing one. This observation has motivated the improvements of the optimizer and the cost model proposed in section 5.

6.3. Effect of Scheduling

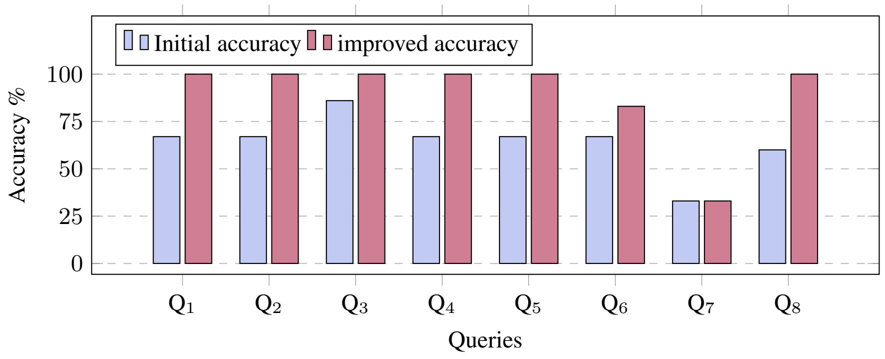


Fig. 6. Initial accuracy and the improved accuracy of the optimizer.

To select the best execution plan and execution strategy, we extended the GoFast optimizer to be able to estimate the cost and number of results of spatial filters. As far as experimental validation goes, we propose to compare the improved version of GoFast with the existing one. For that, we use the accuracy of the best plan prediction as a performance metric. The accuracy of the optimizer for a given query is calculated as follows:

$$A = \frac{\#plans - Rank\_plan}{\#plans - 1} \tag{12}$$

The primary function of an optimizer is to select the optimal execution plan for a given query. To accomplish this, the optimizer assigns a rank to each candidate plan based on an estimation of its cost. The accuracy of the optimizer is measured in terms of the rank of the true best plan. Specifically, the accuracy is calculated as the proportion of the true best plan's rank among all the candidate plans. A higher rank for the true best plan corresponds to a higher accuracy, with an accuracy of 100% indicating that the optimizer has successfully identified the true best plan as the top-ranked plan. Conversely, an accuracy of 0% would indicate that the optimizer ranked the true best plan as the worst among the candidates.

The figure 6 shows the initial accuracy of Go-Fast and the improved accuracy. As we can notice, the optimizer after the proposed improvements provides a better prediction of the best execution plan. It can find the actual best execution plan for the all of the test queries except $Q6$ and $Q7$. Moreover, even for the latter queries, it provides the same or better accuracy than the original optimizer. This is due to a better estimation of the cost of the spatial filters.

The accuracy of both approaches is plotted in the figure 6. However, more detailed results are in the Appendix where we list the results of estimation of each plan compared to the true cost. We will refer to values form the detailed tables to better explain the results. The accuracy on queries $Q6$ and $Q7$ demonstrates that there is still room for improvement for the optimizer. In $Q6$, the improved optimizer chooses the second best execution plan performing better then the old approach, which choose the third best plan. This is due to the error of estimation. The best plan for $Q6$ is the plan $\mathcal{P}1$ with a real cost of 10639, followed by the plan $\mathcal{P}7$ with a real cost of 7338. The results of the estimation proposed a cost of 9894 for $P1$ and 7338 for $P7$ leading to the choice of $P7$ as the best plan.

We can notice the same problem with the query $Q7$ where the cost of $P3$ is 217526 however it is estimated to be 194145. The gap between the real cost and the estimation is due to the number of objects eliminated with the refinement step in the spatial filter. In the refinement step true shapes are considered and in the case of $Q7$ many objects do not satisfy the spatial filter despite that there MBR approximations do satisfy the latter. On top of that, the number of acceptable plans is very low for $Q7$ (only four acceptable plans, meaning that each error is amplified when using the accuracy metric leading to 33% accuracy.

6.4. Effect of Encoding

As we mentioned in section 3, RDF_QDAG stores data in three types of files: spatial index, graph fragments and dictionary files. The description of a spatial object in a vector format can be long, for example the map of a state or a river. For efficiency, we store the full resolution shape definition in the dictionary. The full value will be replaced by an ID in the graph fragments and with an approximation (MBR) in the spatial index.

For the storage of the spatial object, we have mainly two options: The Well Known Text format (WKT) and the Well Known Binary format (WKB). RDF_QDAG is capable of outputting both representations, however, for the storage format, we experimented with both representations to determine the best encoding format for the system.

In Figure 7, we show the effect of the encoding format on the performance of the queries. We can clearly notice that the WKB encoding outperforms the WKT one for all queries. This is due to the different sizes of the two encoding formats. WKB is generally more compact than WKT, which leads to less I/O cost. On top of that, deserializing the WKB format is more efficient than parsing the WKT format. For RDF_QDAG system, if

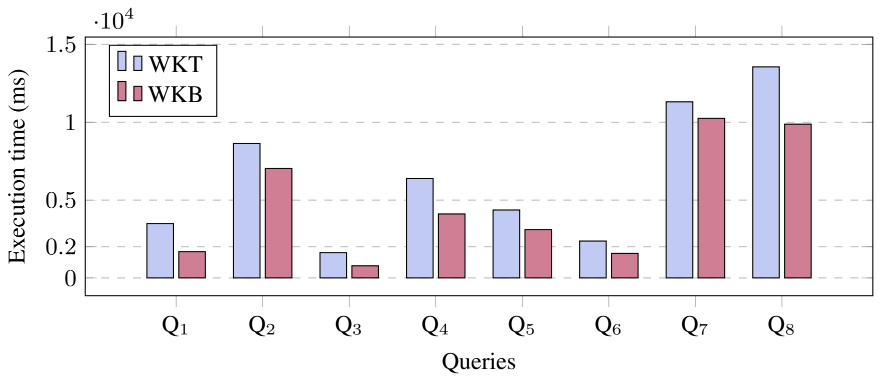


Fig. 7. Execution time (ms) of queries using WKT and WKB.

the user requests an output of the WKT format, it is more efficient to deserialize the WKB stored and convert it to WKT than to parse the WKT format.

6.5. Effect of Spatial Pruning

In figure 8, we compare the execution time of queries with and without spatial pruning. As demonstrated in the figure, the spatial pruning improves performance for most of the queries. This is due to the decreasing size of the search space. However, this is not the case of all queries, since the number of pruned fragments depends on the query and can vary form one to another. This is the case of query Q_2 where no fragment is pruned. More so, the overhead of evaluating the fragments for pruning can be negligible, as demonstrated with the same query (Q_2).

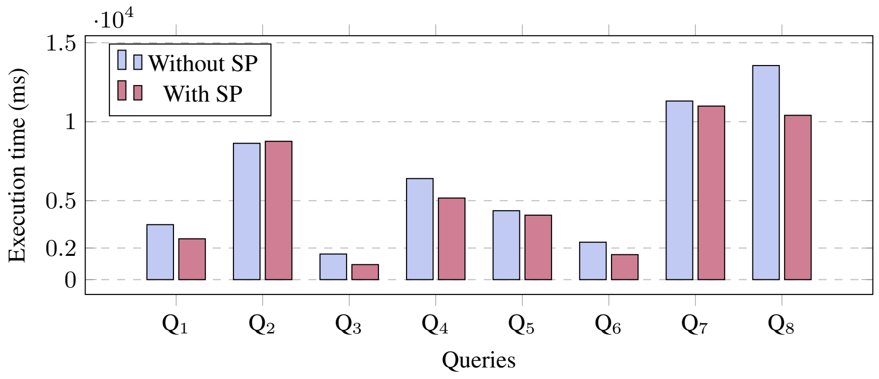


Fig. 8. Execution time (ms) of queries with and without Spatial Pruning.

6.6. Comparison against Virtuoso

After the optimization techniques applied to improve the performance of RDF_QDAG, we compare it with a commercial Triplestore Virtuoso. We choose Virtuoso since it is a stable and wildly used Triplestore. On top of that it is one of the few Triplestores capable of answering spatial-rdf queries since it support the GeoSPARQL norm proposed by the Open Geospatial Consortium. As for the other solutions (e.g., GraphDB and Strabon) we where unable to load the dataset due to stability issues in the mentioned systems.

The figure 9 depicts the execution times of queries run on both Virtuoso and RDF_QDAG. For RDF_QDAG, we plot the execution time of two different runs, one without any optimization technique used (WKT) the other one with the optimization techniques proposed and studied in previous sections (WKB+SP). We can notice that the WKT approach outperforms Virtuoso in some queries like Q_1 and Q_5. However, on most of the queries,Virtuoso still had better performace leading to a better total execution time of 47 seconds for Virtuoso compared to 52 seconds for WKT. On the other hand, after applying the proposed optimization techniques (WKT+SP), RDF_QDAG outperforms Virtuoso on all of the test queries without exception and has a better total execution time leading to an improvement of 28% on average.

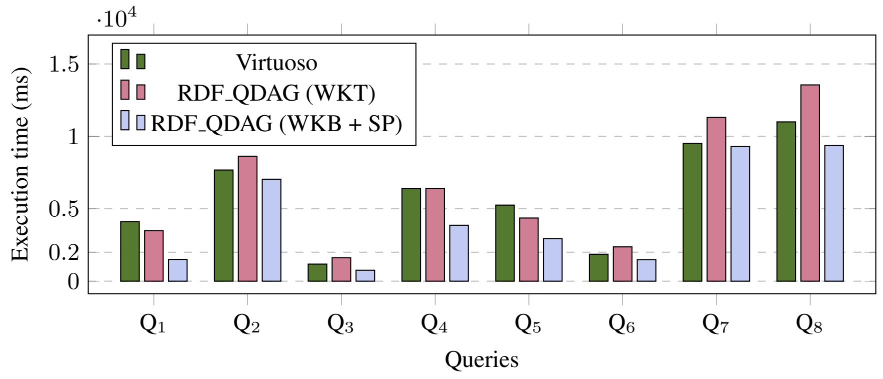


Fig. 9. Compression of execution time between Virtuoso and RDF_QDAG.

7. Conclusion

In this paper, we addressed the evaluation of spatial RDF queries issue in the setting of a graph exploration-based system, known as RDF_QDAG. To enhance the system's capability to answer such queries, we proposed an extension that integrates spatial awareness into the system's storage layer, evaluation engine and optimization process. More specifically, we proposed the use of an R-tree data structure, which is adapted to better fit the system, as well as the integration of the evaluation of spatial filters into the execution plans. Additionally, we introduced two evaluation strategies, namely, BGP-First and Spatial-First, for the execution engine. In terms of optimization, we presented a cost model that considers the cost of spatial operations in order to optimize the selection of execution plans.

Furthermore, we proposed a spatial pruning technique to further improve performance by reducing the search space.

On the other hand, we validated our proposed extension to RDF_QDAG through an experimental setup using a real-world dataset (i.e., YAGO). Our results indicated that the use of optimization techniques such as WKB encoding and spatial pruning improve the performance of the system. We also evaluated the proposed execution strategies of BGP-First and Spatial-First, and found that each strategy had advantages and limitations depending on the query being executed. To address this, we developed a cost model to determine the most suitable strategy for each query. Our results also indicated that the proposed cost model enables the system to better predict the best execution plan compared to the existing one.

In future work, we plan to continue improving the optimizer, particularly, for queries involving complex geometrical shapes in order to enhance its ability to predict the best execution plan. To achieve this, we plan to explore the use of machine learning techniques to integrate feedback from RDF_QDAG query evaluation. Additionally, we intend to extend the system by incorporating support for temporal constraints, enabling it to answer spatio-temporal queries. This could involve adapting the existing cost model, introducing new data structures and indices, and devising new evaluation strategies. The goal is to improve the efficiency and accuracy of spatio-temporal query processing.

References

1. Graphdb. https://graphdb.ontotext.com/, accessed: 2021-10-18
2. Virtuoso. https://virtuoso.openlinksw.com/, accessed: 2021-10-18
3. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Sw-store: a vertically partitioned dbms for semantic web data management. The VLDB Journal 18(2), 385–406 (2009)
4. Battle, R., Kolas, D.: Enabling the geospatial semantic web with parliament and geosparql. Semantic Web 3(4), 355–370 (2012)
5. Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient rdf store over a relational database. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 121–132 (2013)
6. Brahem, M., Zeitouni, K., Yeh, L.: Astroide: a unified astronomical big data processing engine over spark. IEEE Transactions on Big Data 6(3), 477–491 (2018)
7. Brodt, A., Nicklas, D., Mitschang, B.: Deep integration of spatial query processing into native rdf triple stores. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 33–42 (2010)
8. Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: An architecture for storing and querying rdf data and schema information (2001)
9. Chawla, T., Singh, G., Pilli, E.S., Govil, M.C.: Storage, partitioning, indexing and retrieval in big rdf frameworks: A survey. Computer Science Review 38, 100309 (2020)
10. Eldawy, A., Mokbel, M.F.: Spatialhadoop: A mapreduce framework for spatial data. In: 2015 IEEE 31st ICDE conference. pp. 1352–1363. IEEE (2015)
11. Ester, M., Kriegel, H.P., Sander, J.: Spatial data mining: A database approach. In: SSD. vol. 97, pp. 47–66. Citeseer (1997)
12. Güting, R.H.: An introduction to spatial database systems. The VLDB Journal—The Inter. Journal on Very Large Data Bases 3(4), 357–399 (1994)
13. Guttman, A.: R-trees: a dynamic index structure for spatial searching, vol. 14. ACM (1984)
14. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage. 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), Sanibel Island, Florida pp. 1–15 (2003)

15. Harris, S., Lamb, N., Shadbolt, N., et al.: 4store: The design and implementation of a clustered rdf store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). vol. 94 (2009)
16. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: Yago2: A spatially and temporally enhanced knowledge base from wikipedia. Artificial intelligence 194, 28–61 (2013)
17. Khelil, A., Mesmoudi, A., Galicia, J., Bellatreche, L., Hacid, M.S., Coquery, E.: Combining graph exploration and fragmentation for scalable rdf query processing. Information Systems Frontiers 23(1), 165–183 (2021)
18. Kim, K., Cha, S.K., Kwon, K.: Optimizing multidimensional index trees for main memory access. In: ACM SIGMOD Record. vol. 30, pp. 139–150. ACM (2001)
19. Koubarakis, M., Kyzirakos, K.: Modeling and querying metadata in the semantic sensor web: The model strdf and the query language stsparql. In: Extended Semantic Web Conference. pp. 425–439. Springer (2010)
20. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A semantic geospatial dbms. In: International Semantic Web Conference. pp. 295–311. Springer (2012)
21. Lee, J.G., Kang, M.: Geospatial big data: challenges and opportunities. Big Data Research 2(2), 74–81 (2015)
22. Leutenegger, S.T., Lopez, M.A., Edgington, J.: Str: A simple and efficient algorithm for r-tree packing. In: 13th ICDE conf. pp. 497–506. IEEE (1997)
23. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In: 2011 IEEE 27th International Conference on Data Engineering. pp. 984–994. IEEE (2011)
24. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. Proceedings of the VLDB Endowment 1(1), 647–659 (2008)
25. Papadopoulos, T., Balta, M.E.: Climate change and big data analytics: Challenges and opportunities. International Journal of Information Management 63, 102448 (2022)
26. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Transactions on Database Systems (TODS) 34(3), 1–45 (2009)
27. Robinson, J.T.: The kdb-tree: a search structure for large multidimensional dynamic indexes. In: Proc. of the 1981 ACM SIGMOD inter. conf. on Management of data. pp. 10–18. ACM (1981)
28. Roumelis, G., Vassilakopoulos, M., Corral, A.: Nearest neighbor algorithms using xbr-trees. In: 2011 15th Panhellenic Conference on Informatics. pp. 51–55. IEEE (2011)
29. Šidlauskas, D., Šaltenis, S., Christiansen, C.W., Johansen, J.M., Šaulys, D.: Trees or grids?: indexing moving objects in main memory. In: Proc. of the 17th ACM SIGSPATIAL inter. conf. on Advances in Geographic Info. Syst. pp. 236–245. ACM (2009)
30. Silberschatz, A., Korth, H.F., Sudarshan, S., et al.: Database system concepts, vol. 4. McGraw-Hill New York (1997)
31. Stolze, K.: Sql/mm spatial: The standard to manage spatial data in a relational database system. In: BTW 2003–Datenbanksysteme fur Business, Technologie und Web, Tagungsband der 10. BTW Konferenz. Gesellschaft für Informatik eV (2003)
32. Tang, M., Yu, Y., Aref, W., Mahmood, A., Malluhi, Q., Ouzzani, M.: In-memory distributed spatial query processing and optimization. Tech. rep., Purdue technical report (2016)
33. Wald, I., Havran, V.: On building fast kd-trees for ray tracing, and on doing that in o (n log n). In: 2006 IEEE Symposium on Interactive Ray Tracing. pp. 61–69. IEEE (2006)
34. Wang, C.J., Ku, W.S., Chen, H.: Geo-store: a spatially-augmented sparql query evaluation system. In: Proceedings of the 20th International Conference on Advances in Geographic Information Systems. pp. 562–565 (2012)
35. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment 1(1), 1008–1019 (2008)
36. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D., et al.: Efficient rdf storage and retrieval in jena2. In: SWDB. vol. 3, pp. 131–150. Citeseer (2003)

37. Yu, J., Wu, J., Sarwat, M.: Geospark: A cluster computing framework for processing large-scale spatial data. In: Proc. of the 23rd SIGSPATIAL Inter. Conf. on Advances in Geographic Info. Syst. p. 70. ACM (2015)
38. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale rdf data. Proceedings of the VLDB Endowment 6(4), 265–276 (2013)
39. Zouaghi, I., Mesmoudi, A., Galicia, J., Bellatreche, L., Aguili, T.: Gofast: Graph-based optimization for efficient and scalable query evaluation. Information Systems 99, 101738 (2021)

Houssameddine Yousfi is a PhD student in Computer science at the university of Tlemcen, Algeria and the National Engineering School for Mechanics and Aerotechnics (ISAE-ENSMA), Poitiers, France. He is a member of the Laboratory LRIT and the laboratory LIAS. The areas of his scientific interest focus on database management systems, spatial and graph data management and Massively Parallel Processing (MPP) frameworks.

Amin Mesmoudi is an associate professor at the University of Poitiers. He is also a member of the Data Engineering team at the LIAS laboratory. He holds a PhD from the Claude Bernard (Lyon 1) University. His research interests are related to large-scale data persistence, including partitioning, indexing, and compression, as well as data exploitation. He is particularly interested in designing new evaluation and optimization techniques to support Massively Parallel Processing (MPP) frameworks.

Allel Hadjali is currently Full Professor in Computer Science at the National Engineering School for Mechanics and Aerotechnics (ISAE-ENSMA), Poitiers, France. He is a member of the Data and Model Engineering research team of the Laboratory of Computer Science and Automatic Control for Systems (LIAS). His research interests are Massive Data Exploitation and Analysis, Extraction, Recommendation and Explainabiliy in Learning Machine Models. The complete list of his publications is available in http://www.lias-lab.fr/members/allelhadjali.

Houcine Matallah is currently the Head of the Computer Science Department at the University of Tlemcen. He is also a member of the faculty's council and scientific committee. As a member of the LRIT laboratory, his research focuses on database management systems, including NoSQL and New SQL systems, as well as the challenges of Big Data.

Seif-Eddine Benkabou received his M.Sc. and Ph.D. degrees from the University of Lyon, Villeurbanne, France, in 2014 and 2018, respectively. He is currently an Assistant Professor at the University of Poitiers, Poitiers, France. His main research area is unsupervised machine learning, with a focus on outlier detection from temporal data."

Received: February 25, 2023; Accepted: June 10, 2023.

Appendix

Appendix 1: Results of estimation of each plan for the different queries considered

For all of the following tables, the best execution plan is highlighted in bold.

Table 6. Results of estimation of Q_1

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?c},Fu(?g), \overleftarrow{?c}, \overrightarrow{?p}]$	4774913	4775175	7	7
1	**$[\overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$**	**5943.0**	**29657.0**	**3**	**1**
2	$[\overleftarrow{?c}, \overrightarrow{?c}, Fu(?g), \overrightarrow{?p}]$	5595	54502	1	2
3	$[\overleftarrow{?c}, \overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	5720	54627	2	3
4	$[\overleftarrow{?f}, \overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	286682	859297	5	4
5	$[Fu(?g),\overleftarrow{?g}, \overleftarrow{?c}, \overrightarrow{?p}]$	437395	446318	6	5
6	$[\overleftarrow{?n}, \overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	83750	857355	4	6

Table 7. Results of estimation of Q_2

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?c},Fu(?g), \overleftarrow{?c}, \overrightarrow{?p}]$	4775880.0	4777965.0	7	7
1	$[\overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	165958.0	493909.0	4	4
2	$[\overleftarrow{?a}, \overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	243796.0	1321696.0	5	6
3	$[\overleftarrow{?b}, \overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	389065.0	1154313.0	6	5
4	$[\overleftarrow{?c}, \overrightarrow{?c},Fu(?g), \overrightarrow{?p}]$	14421.0	192320.0	1	2
5	$[\overleftarrow{?c}, \overrightarrow{?p}, \overrightarrow{?c},Fu(?g)]$	16412.0	194311.0	2	3
6	**$[Fu(?g),\overleftarrow{?g}, \overleftarrow{?c}, \overrightarrow{?p}]$**	**104914.0**	**129598.0**	**3**	**1**

Table 8. Results of estimation of Q_3

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?a}, \overleftarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l},Fu(?g)]$	7231.0	7388.0	4	4
1	$[\overrightarrow{?l},Fu(?g), \overleftarrow{?l}, \overleftarrow{?w}, \overleftarrow{?a}]$	4774850.0	4774858.0	8	8
2	$[\overrightarrow{?p}, \overrightarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l},Fu(?g)]$	5447.0	5676.0	3	2
3	$[\overrightarrow{?w}, \overrightarrow{?l},Fu(?g), \overleftarrow{?w}, \overleftarrow{?a}]$	669919.0	1252615.0	7	7
4	**$[\overleftarrow{?a}, \overrightarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l},Fu(?g)]$**	**4231.0**	**5492.0**	**2**	**1**
5	$[Fu(?g),\overleftarrow{?g}, \overleftarrow{?l}, \overleftarrow{?w}, \overleftarrow{?a}]$	441052.0	702586.0	6	5
6	$[\overleftarrow{?l}, \overrightarrow{?l},Fu(?g), \overleftarrow{?w}, \overleftarrow{?a}]$	59251.0	1250718.0	5	6
7	$[\overleftarrow{?w}, \overleftarrow{?a}, \overrightarrow{?w}, \overrightarrow{?l},Fu(?g)]$	3245.0	7045.0	1	3

Table 9. Results of estimation of Q_4

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?e}, \overrightarrow{?l},Fu(?g)]$	201726.0	208424.0	3	3
1	$[\overrightarrow{?l},Fu(?g), \overleftarrow{?l}]$	4774844.0	4774844.0	4	4
2	**$[Fu(?g),\overleftarrow{?g}, \overleftarrow{?l}]$**	**74903.0**	**98141.0**	**2**	**1**
3	$[\overleftarrow{?l}, \overrightarrow{?l},Fu(?g)]$	17716.0	194145.0	1	2

Table 10. Results of estimation of Q_5

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?c}, \overleftarrow{?c}, \overrightarrow{?p}, Fu(?g)]$	4775880.0	4777965.0	7	7
1	$[\overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	165958.0	493909.0	4	4
2	$[\overleftarrow{?a}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	243796.0	1321696.0	5	5
3	$[\overleftarrow{?b}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	389065.0	1154313.0	6	6
4	$[\overleftarrow{?c}, \overrightarrow{?c}, \overrightarrow{?p}, Fu(?g)]$	14421.0	192320.0	1	2
5	$[\overleftarrow{?c}, \overrightarrow{?p}, Fu(?g), \overrightarrow{?c}]$	16412.0	194311.0	2	3
6	**$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?c}, \overrightarrow{?p}]$**	**51616.0**	**75701.0**	**3**	**1**

Table 11. Results of estimation of Q_6

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}, \overleftarrow{?u}, \overrightarrow{?p}]$	4774894.0	4774981.0	7	7
1	**$[\overrightarrow{?p}, \overrightarrow{?u}, \overrightarrow{?l}, Fu(?g)]$**	**5901.0**	**9894.0**	**3**	**2**
2	$[\overrightarrow{?u}, \overrightarrow{?l}, Fu(?g), \overleftarrow{?u}, \overrightarrow{?p}]$	669962.0	1252701.0	6	6
3	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}, \overleftarrow{?u}, \overrightarrow{?p}]$	30958.0	250625.0	5	4
4	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g), \overleftarrow{?u}, \overrightarrow{?p}]$	59295.0	1250841.0	4	5
5	$[\overleftarrow{?u}, \overrightarrow{?p}, \overrightarrow{?u}, \overrightarrow{?l}, Fu(?g)]$	5192.0	32596.0	2	3
6	$[\overleftarrow{?w}, \overrightarrow{?p}, \overrightarrow{?u}, \overrightarrow{?l}, Fu(?g)]$	3394.0	7338.0	1	1

Table 12. Results of estimation of Q_7

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?e}, \overrightarrow{?l}, Fu(?g)]$	201726.0	208424.0	2	2
1	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}]$	4774844.0	4774844.0	4	3
2	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}]$	270889.0	297661.0	3	4
3	**$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g)]$**	**17716.0**	**194145.0**	**1**	**1**

Table 13. Results of estimation of Q_8

Plan ID	Plan	# DS	# Triples	Initial position	New position
0	$[\overrightarrow{?l}, Fu(?g), \overleftarrow{?l}, \overrightarrow{?p}]$	4775053.0	4775546.0	6	6
1	**$[\overrightarrow{?p}, \overrightarrow{?l}, Fu(?g)]$**	**105462.0**	**339654.0**	**3**	**1**
2	$[\overleftarrow{?a}, \overrightarrow{?p}, \overrightarrow{?l}, Fu(?g)]$	134597.0	469582.0	4	3
3	$[Fu(?g), \overleftarrow{?g}, \overleftarrow{?l}, \overrightarrow{?p}]$	186351.0	420618.0	5	2
4	$[\overleftarrow{?l}, \overrightarrow{?l}, Fu(?g), \overrightarrow{?p}]$	59454.0	1251406.0	1	4
5	$[\overleftarrow{?l}, \overrightarrow{?p}, \overrightarrow{?l}, Fu(?g)]$	59857.0	1251809.0	2	5