Noname manuscript No.
(will be inserted by the editor)

Combining Graph Exploration and Fragmentation for Scalable RDF Query Processing

Abdallah Khelil · Amin Mesmoudi · Jorge Galicia · Ladjel Bellatreche

Received: date / Accepted: date

Abstract Data representation facilities offered by RDF (Resource Description Framework) have made it very popular. It is now considered as a standard in several fields (Web, Biology, ...). Indeed, by lightening the notion of schema, RDF allows a flexibility in the representation of data. This popularity has given birth to large datasets and has consequently led to the need for efficient processing of these data. In this paper, we propose a new approach allowing query processing on RDF data. We propose to combine RDF graph exploration with physical fragmentation of triples. Graph exploration makes possible to exploit the structure of the graph and its semantics while the fragmentation allows to group together the nodes of the graph having the same properties. Compared to the state of the art (i.e., gStore [20], RDF3X [15], Virtuoso [8]), our approach offers a compromise between efficient query processing and scalability. In this regard, we conducted an experimental study using real and synthetic datasets to validate our approach with respect to scalability and performance.

Keywords RDF · Graph exploration · Fragmentation · Scalability · Performance.

A. Khelil
LIAS/ISAE-ENSMA, 86960 Futuroscope, France
LAPECI/University Oran 1, Algeria
E-mail: abdallah.khelil@ensma.com

A. Mesmoudi
LIAS/University of Poitiers, 86960 Futuroscope, France
E-mail: amin.mesmoudi@univ-poitiers.fr

J. Galicia
LIAS/ISAE-ENSMA, 86960 Futuroscope, France
E-mail: jorge.galicia@ensma.com

L. Bellatreche
LIAS/ISAE-ENSMA, 86960 Futuroscope, France
E-mail: bellatreche@ensma.com

1 Introduction

Storage and data collection methods have been largely impacted by new applications based on modern measurement and observation instruments. In several areas, it is the Pay-as-you-go philosophy that is adopted during the production and storage of data. As a result, the traditional data management approach defining a schema (a structure) and then storing the data according to this schema has become very constraining.

Consequently, new methods of data representation have emerged. The Resource Description Framework (RDF) is one of the efforts led by the W3C to link data from the Web. It is based on the notion of Triples, which consists of representing each information in the form of a triplet $< Subject, Predicate, Object >$. This provides flexibility in data collection and has resulted in many large-scale datasets, for example Freebase[1] who has 2.5 billion triples [4] and DBpedia[2] with more than 170 million of triples [12]. The Linked Open Data Cloud (LOD) now connects over 10,000 datasets and currently has more than 150 billion triples[3]. The number of data sources has doubled in the last three years (2015-2018).

The development of the SPARQL language facilitated the exploitation of RDF data and required the development of novel approaches to query processing. Indeed, several studies have shown that conventional relational data management systems are not adapted to manage triplets. In particular, they are not able to guarantee the scalability and performance. The lack of an explicit schema in the RDF database and the number of junctions in a SPARQL query are the two main reasons for this limitation. In order to improve the per-

[1] http://www.freebase.com/.
[2] http://wiki.dbpedia.org
[3] http://lodstats.aksw.org/

formance of RDF data processing systems, two types of approaches are available in the literature. The first one (e.g., [15, 18]) takes advantage of traditional techniques of storage and query evaluation. The join is the main query evaluation operator used in this kind of approaches. In these relational-based approaches, RDF triples are stored as a big table of three attributes. Each SPARQL query is translated into an SQL query with many self-joins on this big table. In this type of approach, the number of self joins is very large which makes SQL queries difficult to evaluate and optimize. Extensive indexing has been widely used by this type of systems. This contributed to improve the performance of some queries. Another technique such as grouping of triplets by predicate (Property tables) has allowed to improve the performance of some queries. However both techniques are far from being optimal.

The second type of approaches (e.g., gStore [20]) considers the query evaluation problem as a graph matching problem, which allows to maintain the original representation of the RDF data. Unfortunately, this type of approach does not guarantee scalability and performance. Mainly because it is necessary to have an evaluation mechanism that controls the memory used. This aspect is not supported by the graph matching algorithms.

In this paper, we propose to take advantage of both types of approaches. We rely on indexing and fragmention to minimize the I/O's and on the exploration of the graph to take into account its structure. Unlike graph matching approaches our evaluation strategy uses the Volcano [9] model, which allows to control the memory use and to avoid any possible overflow. The rest of this paper is organized as follows. We discuss in Section 2 the related work. Then, in Section 3 we give an overview and formalize of our work. In Section 4, we present our experimental study and finally, in Section 5 we conclude and discuss some future research.

2 Background and Related Work

Relational Database Management Systems (RDBMSs) have been the preferred technology solution to manage the storage and retrieval of data for years. Modelizing the data as tables allows not only to organize raw datasets but to apply relational algebra operations to efficiently query the data. However, in order to create a relational database, the schema of the data needs to be established beforehand. Recently, the data collection strategies have largely improved in many domains like Biology, Astronomy, and certainly the Web. In these applications, the schema of the database is not fixed and it is unknown in advance, making the relational model not the most suitable solution. The data in these applications are usually modeled as a graph. Figure 1a illustrates a dataset (named G from this point) about a movie collected from the Web. The Resource Description Framework (RDF) is the W3C standard that models the data as graphs, decomposing the graph into triples with the form $< subject, predicate, object >$. The data is then queried generally with the SPARQL syntax based on graph pattern matching. An example query is shown in Figure 1b.

Still, most of the existent RDF processing systems adapted the traditional relational DBMSs to store and query the data modeled as graphs. In these systems, the data are physically stored as a table(s) and queries are translated from the graph query language (mostly SPARQL) into SQL queries. Traditional DBMSs do not guarantee good performances when treating these data and the logical flexibility of modeling the data as a graph is lost when triples are mapped to tables. Finding matches to complex queries implies the execution of very costly joins that could be avoided with graph exploration strategies if the data were modeled as graphs. Next, we give an overview of the state-of-the-art approaches modeling the data as tables.

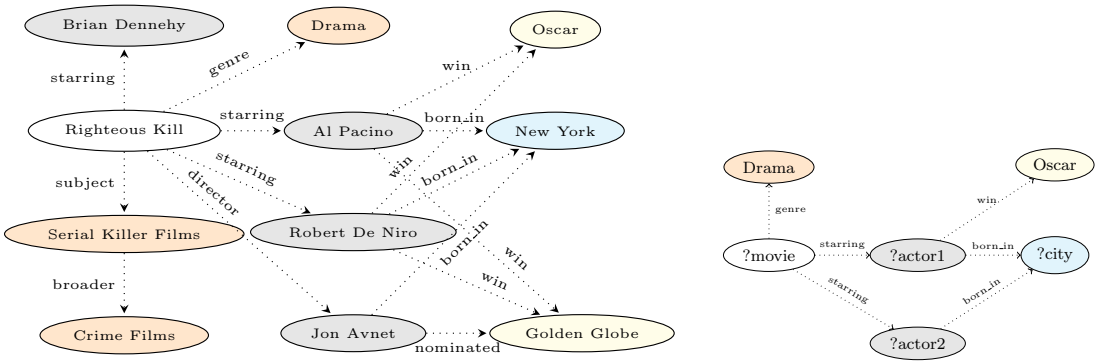
2.1 Relational-based approaches

2.1.1 Big table

This approache firstly introduced in [6] proposed to store and query triples stored on a single very large table of three attributes. The attributes correspond to the Subject Predicate and Object of a triple. An example of this approach used to store G is shown in Figure 2a. A graph SPARQL query is transformed into SQL using many self-joins to this table. The transformation of the query of Figure 1b is as follows:

```
SELECT T1.S as ?movie, T2.O as ?actor1,
T3.O as ?actor2, T4.O as ?city
FROM T as T1, T as T2, T as T3, T as T4, T as T5
WHERE { T1.O='Drama' AND T1.P='genre' AND
T1.S=T2.S AND T2.P='starring' AND
T1.S=T3.S AND T3.P='starring' AND
T4.S=T2.O AND T4.P='win' AND T4.O='Oscar' AND
T3.S=T2.O AND T4.P='born_in' AND
T5.S=T3.O AND T5.P='born_in' AND T4.O=T5.O       }
```

This approach has a major drawback and it is the number of generated joins. The number of joins corresponds to the number of triples in the query and to solve them it is necessary to scan the entire table many times implying a very high cost. To avoid the full table scan, some approaches use intensive indexing strategies as the ones described next.



(a) Data graph G (b) Query

Fig. 1: Example graph and query

2.1.2 Intensive indexing approaches

The second category of approaches rely on indexes. Systems applying this strategy are RDF-3X [15] and Hexastore [18]. In these systems, the strings are firstly mapped to id's in an index. Then, the triples are encoded and compressed before being indexed on separate B+-Tree indexes using different orders (e.g. SPO, OPS, POS). The data is replicated on each index but the gain on performance is considerable. Graph queries are translated into SQL and joining tables is still a mandatory step. However, the joins are performed very efficiently thanks to the use of the merge-joins instead of self-joins in a single table. The drawbacks of this approach are: 1) the use of additional disk space because data are replicated many times, and 2) the full scan of indexes(e.g SPO, OPS) is efficient but still expensive.

2.1.3 Property table

Another approach, called Property table, was proposed in the framework of Jena [13] and DB2-RDF [5]. This approach stores triples with related properties in the same table in such a way that queries involving the same subject (subject-subject queries) become single table scans. This approach is illustrated in Figure 2c. The tables of the Figure show the approach applied by the Jena2 system [19] which is able to store multi-valued properties creating tables for mutli-valued properties (tables ${}^*T_1$ and ${}^*T_2$ in the Figure). The major drawbacks of this approach are the unnecessary number of null values stored and the performance of the system to solve non subject-subject joins.

2.1.4 Binary tables

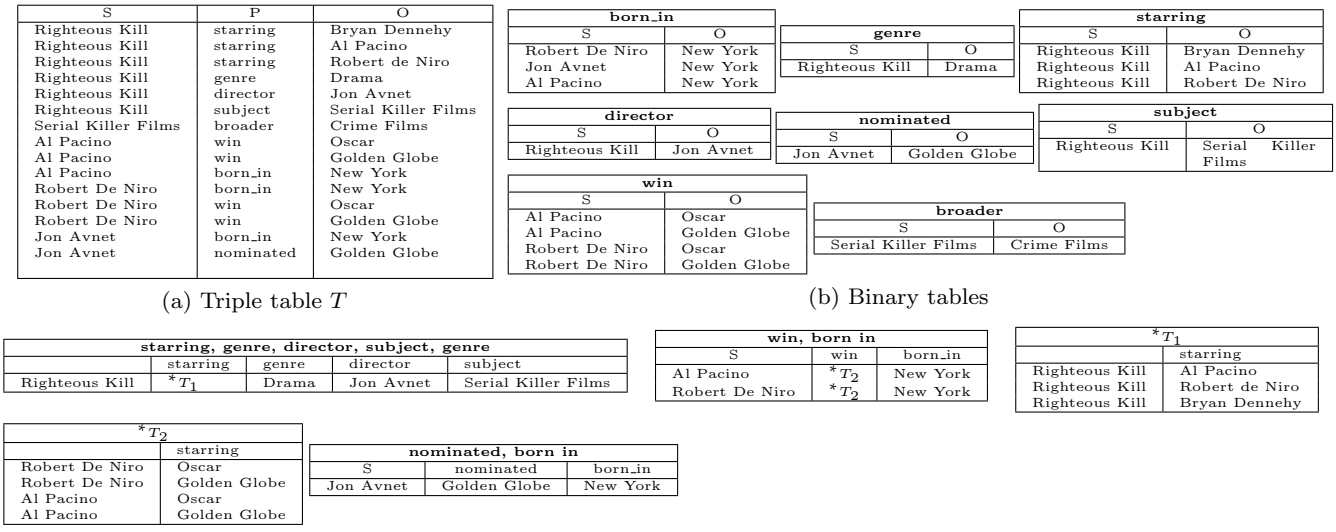
The approach based on Binary tables was firstly introduced in [1]. This approach builds a two-column table for each property containing both subject and object, ordered by subjects. This approach is also called vertical partitioned tables. The total number of two-column tables creates equals the number of unique properties in the data. Column-store systems could be used to speed up the evaluation of queries. This approach is illustrated in Figure 2b. Binary tables support multi-valued properties and only relevant data are stored (i.e. no need to manage `NULL` values contrary to Property Tables). This approach performs very efficiently for queries involving small number of predicates. Conversely, if this number increases the system becomes very inefficient due to join overhead.

2.2 Graph-based approaches

The systems under this approach do not rely on a table(s) to store and query the data. They represent the data on disk using a strategy specifically designed to store graphs and solve queries exploring the input graph. For example, the system gStore [20] represents the data with an adjacency that stores each node and its outgoing edges. The system relies on intensive indexes on this list to find the matches for a query pattern. This index, based on S-Tree [7], is used to speed up the evaluation of graph matching patterns with bit-wise operators. Unfortunately, this type of approach works mainly in memory and does not control the amount of memory used. This induces an overflow when processing certain queries in infrastructures with not much main memory available unlike join-based approaches.

2.3 In-memory based systems

More complex data structures used to process RDF data have been proposed in several in-memory processing systems. The system BitMat [3] for instance, is a



S	P	O
Righteous Kill	starring	Bryan Dennehy
Righteous Kill	starring	Al Pacino
Righteous Kill	starring	Robert de Niro
Righteous Kill	genre	Drama
Righteous Kill	director	Jon Avnet
Righteous Kill	subject	Serial Killer Films
Serial Killer Films	broader	Crime Films
Al Pacino	win	Oscar
Al Pacino	win	Golden Globe
Al Pacino	born_in	New York
Robert De Niro	born_in	New York
Robert De Niro	win	Oscar
Robert De Niro	win	Golden Globe
Jon Avnet	born_in	New York
Jon Avnet	nominated	Golden Globe

(a) Triple table T

born_in

S	O
Robert De Niro	New York
Jon Avnet	New York
Al Pacino	New York

genre

S	O
Righteous Kill	Drama

starring

S	O
Righteous Kill	Bryan Dennehy
Righteous Kill	Al Pacino
Righteous Kill	Robert De Niro

director

S	O
Righteous Kill	Jon Avnet

nominated

S	O
Jon Avnet	Golden Globe

subject

S	O
Righteous Kill	Serial Killer Films

win

S	O
Al Pacino	Oscar
Al Pacino	Golden Globe
Robert De Niro	Oscar
Robert De Niro	Golden Globe

broader

S	O
Serial Killer Films	Crime Films

(b) Binary tables

starring, genre, director, subject, genre

	starring	genre	director	subject
Righteous Kill	$*T_1$	Drama	Jon Avnet	Serial Killer Films

win, born in

S	win	born_in
Al Pacino	$*T_2$	New York
Robert De Niro	$*T_2$	New York

$*T_1$

	starring
Righteous Kill	Al Pacino
Righteous Kill	Robert de Niro
Righteous Kill	Bryan Dennehy

$*T_2$

	starring
Robert De Niro	Oscar
Robert De Niro	Golden Globe
Al Pacino	Oscar
Al Pacino	Golden Globe

nominated, born in

S	nominated	born_in
Jon Avnet	Golden Globe	New York

(c) Property tables

Fig. 2: Storage approaches

main memory based bit-matrix structure representing RDF triples. BRAHMS [10] maintains three indexes in main memory (subject → object-predicate, object → subject-predicate, and predicate → subject-object), GRIN [17] maintains similarly an in-memory graph-based RDF index. Even if the data structures offered by these models are less rigid than the ones proposed before, its scalability is compromised by the size of the main memory available.

In our work, we adopt a storage as graph. Our model scales since it is based in the Volcano execution model described later exchanging data between the hard-drive and main memory. We consider only queries with constant predicates, and we store only forward and backward edges as SPO and OPS. We then split SPO and OPS into many segments where each segment stores a set of subjects (in the case of SPO) or objects (in the case of OPS) that have the same edges (predicates). Each segment is then stored as a separated clustered B+Tree. We also offer an evaluation mechanism that allows to exploit different types of data structure. Our evaluation operations take into account the amount of available memory which allows to avoid any overflow.

3 Our approach

We start by presenting our formal framework for SPJ (Select-Project-Join) queries. We then explain how to extend this framework for other queries (*e.g.*, Group by, Order By). We give firstly an overview of the system's architecture and then detail each layer of the system.

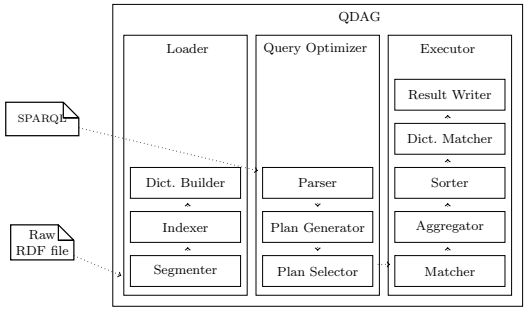


Fig. 3: System architecture

3.1 System Architecture

The architecture of our system is illustrated in Fig. 3. QDAG's architecture consists of three main modules: the Loader, the Query Optimizer and the Processor. In this section we detail the components and main objectives of each module.

3.1.1 Loader

This module is in charge of the data pre-processing and indexing. It receives as input an RDF file that is indexed and compressed to segments. This module is composed of three main units:

– *Segmenter*: This component is in charge of pre-processing the input RDF file. It creates firstly the data stars and then clusters and stores them in distinct segment files. The strings in the input file are indexed and the data is compressed with the graph storage format previously defined.

- *Indexer*: Receives as input the pre-processed data sent by the segmenter. This component creates the encoded binary files used by the system and generates the B+Tree indexes.
- *Dictionary Builder*: This component manages the creation of the string index.

3.1.2 Query optimizer

The query execution plan is generated in this module. Its main components transform the input SPARQL query. By the means of a cost model, this module selects the execution plan (expressed as a succession of star queries) to be sent to the execution layer of the system.

- *Parser*: this sub-module parses the initial SPARQL query and transforms it to a list of query stars (SQ_f or SQ_b).
- *Plan Generator*: using heuristics, valid execution plans expressed as a sequence of forward or backward star queries (SQ) are generated. Two execution plans for a specific query are illustrated in Figure 8.
- *Plan Selector*: based on statistics maintained per segments and on the selectivity of the query, the execution cost is estimated for each generated plan. The plan selector sends to the execution layer of the system the plan with the lowest estimated cost.

3.1.3 Executor:

This module is in charge of the execution of the plan generated by the query optimizer. The execution module pipelines the data for each star query of the plan in parallel applying the open-next-close interface or Volcano model [9].

- *Matcher*: the matcher receives the execution plan from the Plan Selector and triggers the execution. Explained more in detail in Section.
- *Aggregator*: this layer aggregates the data received from the matcher when the input query involves aggregations (i.e. `GROUP BY` clauses).
- *Sorter*: this layer is executed before sending the final results sorting them if an `ORDER BY` clause is present in the input query.
- *Dictionary matcher*: all the matches to queries are found using codified data. The dictionary of string is requested only if a string is useful to prune intermediate results (e.g. in a filter operation). This component communicates with the string index created by the Dictionary Builder and sends back to the user the decoded results.
- *Result Writer*: In this layer the final result is written. The interface could be instantiated as a Network, Console or File Result Writer.

3.2 Graph Storage

Several approaches have been proposed to efficiently manage data graphs. For example, RDF-3X [15], HexaStore [18] and Virtuoso [8] propose to store graphs in the form of SPO (Subject, Predicate, Object) and OPS (Object, Predicate, Subject). From a graph exploration point of view, each triple stored in the OPS schema corresponds to a node and its *incoming* edges whereas SPO stores each node along with its *forward* edges.

Most of existing approaches store either each node's forward edges (e.g. property table based approaches) or they store both, replicating forward and backward edges. The storage schema influences greatly its execution model since the query execution layer of the systems is designed for a specific storage configuration. A system storing the data applying the OPS schema would be more performant for some queries than another storing as SPO and vice-versa.

In our work we consider a graph-based data representation that uses both SPO and OPS structures. We did not consider other combinations, like POS for example, since our workload only considers queries with constant predicates. SPO and OPS are generally stored as a clustered B+Tree which allows triples retrieval in $log(n)$ disk accesses. Full scan is not necessary for selective queries.

Instead of storing two single files (one for SPO and another for OPS), the data on each are splitted grouping triples corresponding to the same entity. SPO and OPS store a lot of (heterogeneous) information. For example, one can find in the same graph information on movies and also information about stock transactions. However, queries rarely manipulate unrelated information. A query focus on a subset of the dataset characterized mainly by its predicates. Our storage model considers that data related to the same entity share the same set of predicates. The query execution begins with the identification of the characterized set(s) to be read according to the predicates of the query. We propose to split, in segments, the SPO and OPS structures with respect to the predicates that cover subjects in the case of SPO (or objects in the case of OPS). This splitting corresponds to the idea of characteristic sets [14], except that the characteristic sets are used only to collect statistics on data.

Predicate	ID
starring	a
genre	b
director	c
subject	d
broader	e
win	f
born_in	g
nominated	h

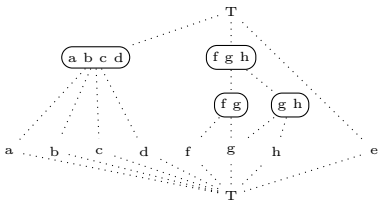


Fig. 4: SPO Lattice example

In our work, data are physically stored as segments.[4] In the case of a (non-selective) query where we have to perform a full scan, only relevant segments (which satisfy the query) will be scanned. The characteristic sets for the SPO configuration of the graph of Figure 1a are organized in a in a lattice as the one shown in Figure 4. The data are organized in 4 SPO segments ($abcd, fg, ghande$).

Definition 1 (Segment) A segment is a part of SPO (or OPS) that satisfies a set of predicates.

Let us consider a subset of the query of Figure 1b:

```
SELECT ?actor1 WHERE {
?actor1 win      Oscar
?actor1 born_in  ?city
?actor3 born_in  ?city   }
```

In order to find the matches to the query, the segments identified as fg and gh in Figure 4 need to be scanned. To efficiently find these relevant segments, we index the segment labels as a lattice [2]. Satisfaction operator of the lattice allows to find relevant segments. Each node of the lattice stores a set of predicates and a list of segments that satisfy those predicates.

The graph G of Figure 1a is stored using an SPO and OPS segments as illustrated in Figures 5a and 5b respectively.

3.2.1 Compression

As explained previously, data are fragmented into segments. Each segment is stored as a clustered B+Tree. For each Triple, our evaluation methods need to locate the segments hosting the subject and the object of the triple.

Each triple is extended as follows:
< $node_1$,sg,predicate,$node_2$,sg_{in},sg_{out} >
where:

- $node_1$: subject in the case of forward triple or object in the case of backward triple.
- sg: is the id of segment storing forward edges in the case where $node_1$ is a subject or backward edges where $node_1$ is an object.
- $node_2$: object in the case of forward triple or subject in the case of backward triple.
- sg_{in}: segment storing backward edges of $node_2$.
- sg_{out}: segment storing forward edges of $node_2$.

$node_1$ and $node_2$ are represented using 64-bit (8 bytes), whereas sg, sg_{in} and sg_{out} are represented using 32 bits (4 bytes). We use only one bit to represent the change of the predicate. Indeed, as data are sorted in the order subject, predicate and object and for all data stars of the segment we have the same set of predicates, we propose to use only one bit to indicate the change of the predicate.

Our approach to data compression is inspired from the one used in RDF-3X [15]. The number of bytes used to represent this information varies according to the number of bytes used to encode each element of the triple. For the $node_1$ we have 9 states. 0 to indicate that the $node_1$ does not change with respect to the previous triple. i $\in$ [1..8] to indicate the number we used to encode the new $node_1$. So, we need 4 bits to represent the 9 states.

For sg, sg_{in} and sg_{out}, we need 5 states. 0 to indicate that this information does not change or NULL. i $\in$ [1..4] to indicate the number we used to encode the new segment id. So, we need 3 bits to represent the 5 states. For $node_2$ we use i $\in$ [1..8] to indicate the number we used to encode the new $node_2$. So, we need 4 bits to represent the 8 states. For the predicate we need only 1 bit to indicate the change. We do not need to store the predicate.

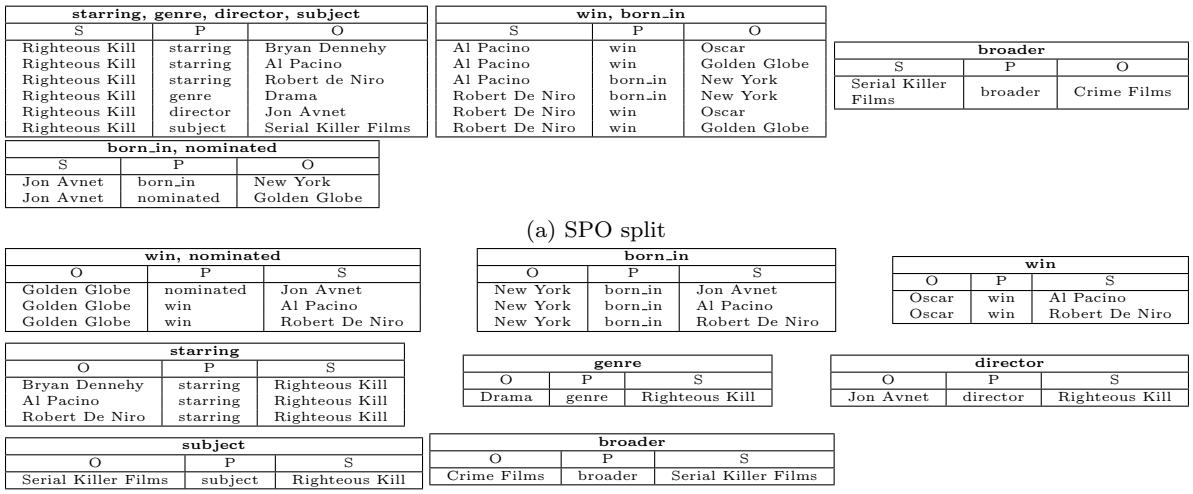
As shown in figure **??**, in total we need 18 bits. Those bits represent the indicator of the number of bytes used to encode each triple. Please note that only leaf pages are compressed.

3.2.2 B+Tree access

In this section, we discuss our strategy of bulk search of data stars using the B+Tree. Our evaluation operators use the B+tree structure to find relevant data stars. The naive algorithm consists in performing a search on B+Tree for every head we need. If we have n heads, then we need $n * log(n)$ time to find all needed data stars.

We propose another strategy. We first start by finding the first data star. During this operation, we memorize the non leaf keys we used. We pass them to the next data stars. We compare with the key previously memorized, we trigger a partial find if we violate the

[4] In the rest of this paper we use the word segment instead of characteristic sets to design the physical split of SPO or OPS.



starring, genre, director, subject

S	P	O
Righteous Kill	starring	Bryan Dennehy
Righteous Kill	starring	Al Pacino
Righteous Kill	starring	Robert de Niro
Righteous Kill	genre	Drama
Righteous Kill	director	Jon Avnet
Righteous Kill	subject	Serial Killer Films

win, born_in

S	P	O
Al Pacino	win	Oscar
Al Pacino	win	Golden Globe
Al Pacino	born_in	New York
Robert De Niro	born_in	New York
Robert De Niro	win	Oscar
Robert De Niro	win	Golden Globe

broader

S	P	O
Serial Killer Films	broader	Crime Films

born_in, nominated

S	P	O
Jon Avnet	born_in	New York
Jon Avnet	nominated	Golden Globe

(a) SPO split

win, nominated

O	P	S
Golden Globe	nominated	Jon Avnet
Golden Globe	win	Al Pacino
Golden Globe	win	Robert De Niro

born_in

O	P	S
New York	born_in	Jon Avnet
New York	born_in	Al Pacino
New York	born_in	Robert De Niro

win

O	P	S
Oscar	win	Al Pacino
Oscar	win	Robert De Niro

starring

O	P	S
Bryan Dennehy	starring	Righteous Kill
Al Pacino	starring	Righteous Kill
Robert De Niro	starring	Righteous Kill

genre

O	P	S
Drama	genre	Righteous Kill

director

O	P	S
Jon Avnet	director	Righteous Kill

subject

O	P	S
Serial Killer Films	subject	Righteous Kill

broader

O	P	S
Crime Films	broader	Serial Killer Films

(b) OPS split

Fig. 5: Data Splitting Example

Indicator	Predicate	$Node_1$	Segment	$Node_2$	Segment (in)	Segment (out)
17 bits	1 bit	0-8 bytes	0-4 bytes	0-8 bytes	0-4 bytes	0-4 bytes

Fig. 6: Compression diagram

memorized key. The input of our algorithm is a vector of data star heads and the output is a set of data stars.

3.3 Query Evaluation

In this section we describe the Query Optimizer and Executor layer of our architecture. First, we give the main concepts and definitions. Then we explain the execution model of the system based on the Volcano [9] architecture. Finally we illustrate the execution model with an example query.

3.3.1 Core definitions

Definition 2 (Triples) In our work, data are described using triples of the form (s, p, o), where s the subject, p is the predicate and o is the object.

Definition 3 (Graph Database) A graph Database is denoted as $G = \langle V_c, L_V, E, L_E \rangle$ where V_c is a collection of vertices which correspond to all subjects and objects in a data graph, L_V is a collection of vertex labels, E is a collection of directed edges that connect the corresponding subjects and objects, and L_E is a collection of edge labels. Given an edge $e \in E$, its edge label is its property .

A graph database with facts related to the "Righteous Kill" film is illustrated in Figure 1a.

Definition 4 (Graph query) A query graph is denoted as $Q = \langle V, L_V, E, L_E \rangle$, where $V = V_p \cup V_c$ is a collection of vertices that correspond to all subjects and objects in a Graph query, where V_p is a collection of parameter vertices, and V_c is as defined in the previous definition. As a naming convention, we distinguish variables from elements in V_c through a leading question mark symbol (*e.g.*, ?name, ?x), L_V is a collection of vertex labels. For a vertex $v \in V_p$, its vertex label is ϕ[5]. A vertex $v \in V_c$, E and L_E are defined in definition 3.

Definition 5 (Stars) We call data star the set of edges related to a given node in the data graph. If the edges are forward, we call this star "data forward star". We use the symbol: $SD_f(x)$, where $x \in V_c$, to denote the forward data star obtained from x. In this case, x is called the head of the star. If the triples are backward, we call this star backward data star. We use the symbol: $SD_b(x)$. For simplicity, we use $\overrightarrow{x}$ (respectively $\overleftarrow{x}$) to denote forward (respectively backward) star obtained using x. We apply the same principle to distinguish stars in a query. We use forward query star ($SQ_f(x)$) and backward query star ($SQ_b(x)$), where $x \in V_c \cup V_p$, to denote stars obtained from forward triples and backward triples respectively.

Forward data stars extend tuple definition. In relational model, we cannot represent more than one value

[5] ϕ is used to denote an empty element

for an attribute. In the case of one attribute as a primary key, we use it as a head of the data star, otherwise, we use the record id.

Proposition 1 *Given a set of predicates in a node and a set of predicates in a star SQ, a lattice node S satisfies ($\models$) the star SQ iff predicates(SQ) $\subseteq$predicates(S)*

From our query example we extract the stars shown in Table 1. We now explain how to evaluate a query by evaluating star queries.

Definition 6 (Mapping and Mapping universe) [16] A mapping is a partial function $\mu : V_p \rightarrow V_c$ from a subset of variables V_p to constant nodes V_c. The domain of a mapping μ, written dom(μ), is defined as the subset of V_p for which μ is defined. By M we denote the universe of all mappings.

We next define the main notion of compatibility between mappings. Informally speaking, two mappings are compatible if they do not contain contradicting variable bindings, *i.e.* if shared variables always map to the same value in both mappings:

Definition 7 (Compatibility of Mappings) Given two mappings μ1, μ2, we say μ1 is compatible with μ2 iff $\mu 1(?x) = \mu 2(?x)$ for all ?x $\in$ dom(μ1) $\cap$ dom(μ2). We write $\mu 1 \sim \mu 2$ if μ1 and μ2 are compatible, and $\mu 1 \not\sim \mu 2$ otherwise.

We denote by *vars* the function that returns variables of the element passed as a parameter. For instance, we denote by $vars(t)$ all variables in the triple pattern t, while $vars(SQ_f(x))$ denotes all variables of the query star $SQ_f(x)$.

We write μ(t) to denote the triple pattern obtained when replacing all variables ?x $\in vars(t)$ in t by μ(?x).

In the following, we use $[\![x]\!]_G$ to denote the process to find relevant mappings of x with respect to G.

Definition 8 (Triple Evaluation) We call $[\![t]\!]_G$ the process allowing to find the mapping related to a triple pattern t with respect to a graph G. $[\![t]\!]_G$ is formally defined as follows: $[\![t]\!]_G := \{\mu | dom(\mu) = vars(t)$ and $\mu(t) \in G\}$

Informally speaking, we try to find mappings such that when we replace the variable nodes by corresponding constants, the triple obtained is in the data graph.

In the following, we use triple evaluation to define query star evaluation.

Definition 9 (query star Evaluation) Informally speaking, the evaluation of query star allows to find mappings for variables in the query star with respect to data graph. For each triple t in the star, we try to find the set of mappings satisfying t. We then construct the mappings of a query star by joining mappings extracted from triples of the query star.

Formally, the evaluation of a query star $SQ(x)$ with respect to the data graph G is defined as follows:

$[\![SQ(x)]\!]_G := \{[\![t_1]\!]_G \bowtie [\![t_2]\!]_G \bowtie ... \bowtie [\![t_n]\!]_G | n = card(SQ(x)\}$

Where:

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

From this definition, we define the evaluation of the JOIN operator between two star queries. Indeed, JOIN will be used as basic operator of the query evaluation.

Definition 10 (Stars Join) Informally speaking, JOIN of two star queries allows to assembly the mappings obtained by evaluating two star queries. Formally, evaluation of a join between two star queries is defined as following:

$[\![SQ_i]\!]_G \bowtie [\![SQ_j]\!]_G = \{\mu_l \cup \mu_r | \mu_l \in [\![SQ_i]\!]_G, \mu_r \in [\![SQ_j]\!]_G$ and $\mu_l \sim \mu_r\}$

We take a mapping obtained from the left query star and another from the right query star, we check if the two mappings are compatible, the union of those mappings is a valid mapping for the JOIN of the two star queries.

Before presenting query evaluation using star queries, we will define the cover concept. Indeed, we use this concept to guarantee that a given set of star queries allow the correct evaluation of the query.

Definition 11 (Star cover) We denote by $Cover_q(SQ)$ the set of query triples shared with the query star.

$Cover_q(SQ) = \{t | t \in Triples(q) \cup Triples(SQ)\}$

From previous definitions, we can define query evaluation using a set of star queries as follows:

Proposition 2 *Given a set of stars $\{SQ_1, SQ_2,...,SQ_n\}$ that cover the query, i.e., $Cover_q(SQ_1) \cup Cover_q(SQ_2) \cup ... \cup Cover_q(SQ_n) = Triples(q)$, the evaluation of q using the set of star queries is defined as follows:*

$[\![q]\!]_G = [\![SQ_1]\!]_G \bowtie [\![SQ_2]\!]_G \bowtie ... \bowtie [\![SQ_n]\!]_G$

With respect to segments, we can define the query evalaution as follows:

$[\![q]\!]_G = \bigcup_{sg \models SQ_1} [\![SQ_1]\!]_{sg} \bowtie \bigcup_{sg \models SQ_2} [\![SQ_2]\!]_{sg} \bowtie ... \bowtie \bigcup_{sg \models SQ_n} [\![SQ_n]\!]_{sg}$

Note that one result of the evaluation of a triple, a query star and a query is a set of mappings. For the rest of this manuscript we denote by ω this set of mappings. A set of mappings, that represent all results obtained by evaluating a triple, a query star or a query is denoted by Ω (*i.e.*, $\Omega = \{\omega\}$)

Table 1: Example: query stars

SQ	Stars
$SQ_{1f}(?movie)$	{(?movie,genre,Drama),(?movie,starring,?actor1),(?movie,starring,?actor2)}
$SQ_{2f}(?actor1)$	{(?actor1,win,Oscar),(?actor1,born_in,?city)}
$SQ_{3f}(?actor2)$	{(?actor2,born_in,?city)}
$SQ_{1b}(Oscar)$	{(?actor1,win,Oscar)}
$SQ_{2b}(?city)$	{(?actor1,born_in,?city),(?actor2,,born_in,?city)}
$SQ_{3b}(?actor1)$	{(?movie,starring,?actor1)}
$SQ_{4b}(?actor2)$	{(?movie,starring,?actor2)}
$SQ_{5b}(Drama)$	{(?movie,genre,Drama)}

Given a query and the stars obtained from backward forward edges, we can easily show that the set of star queries allowing to evaluate the query is not unique.

We use the word "Plan" to refer to a set of star queries allowing to evaluate a given query. Formally, a plan is defined as follows:

Proposition 3 *A plan is a order function on a Set of Query Stars allowing to evaluate queries. We denote by* $p = [SQ_1, SQ_2, ..., SQ_n]$ *the plan formed by executing* SQ_1*, then* SQ_2*,..., and finally* SQ_n

The choice of the star queries and the order of evaluation allow to optimize the query evaluation process.

3.3.2 Volcano execution model

As it was previously mentioned, the execution core of the system is based on the Volcano iterator model which allows to control the amount of data loaded to main memory avoiding any overflow. Let us consider that the Plan Selector module choose the query execution plan $\{SQ_{1f}, SQ_{2f}, SQ_{3f}\}$. The execution model of our system is illustrated in Figure 7. The data organized and indexed as segments is also represented on the Figure.

The execution engine starts assigning a group of candidate segments to match each star query SQ of the plan based on the SQ's predicates. The sets of segments $\{1\}$, $\{4\}$, $\{3, 4\}$ are assigned respectively to SQ_{1f}, SQ_{2f} and SQ_{3f}. Then, the data of the assigned segments are loaded sequentially to a buffer created for each SQ named *input buffer* (e.g. buffers 1-3 of Fig. 7). Next, when the first input buffer is full (Buffer 1), the executor collects its data and finds the triples matching the star query (SQ_{1f} in our example). These matching triples are sent data to an *output buffer* (Buffer 5 on Fig. 7). Once the output buffer of SQ_{1f} is full, the executor triggers the execution of SQ_{2f} collecting the data from both buffers (i.e. Buffers 2 and 4 of Fig. 7). The results of this operation are again sent to an output buffer (Buffer 5) and the operation continues successively. The execution process stops when there is no more data to be scanned in any segment assigned to all the star queries.

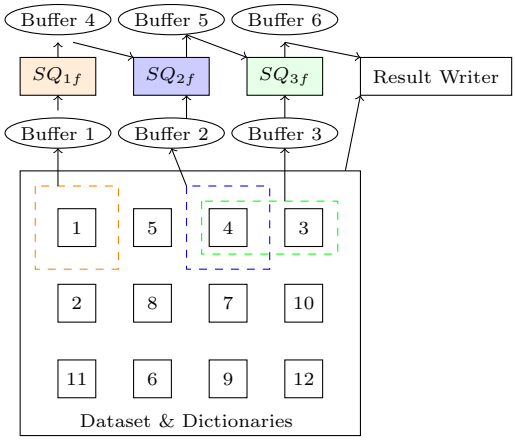


Fig. 7: QDAG execution model

3.3.3 Execution example

Let us illustrate the execution model of the system using the query shown in Figure 1b. The query is firstly parsed by in the Parser component of the system. Then, a series of execution plans (expressed as a serie of star-queries) are generated for the queries. The forward and backward star queries for each node are shown in Table 1. Two execution plans are illustrated in Figures 8a and 8b. Our search space will be discussed later. Indeed, with respect to a parallel model, only a subset of the search space will be considered. The Plan Selector component of the system uses statistics about the stored segments and estimates the cost to complete the execution plan.

The plan with the lowest cost is selected and sent to the Matcher component of the execution system. Let us suppose that the chosen execution plan is the one shown in Figure 8a. The execution layer for this plan is illustrated in Figure 7. The Matcher component in the execution layer checks if for the first star query (SQ_{1f}) finding some matches in the dictionary prunes the number of intermediate results. Otherwise, all the treatment is done with the codified data. For the example, the node *Drama* is looked in the dictionary index

Table 2: Experimental datasets

Dataset	Watdiv			LUBM			
Size	10M	100M	200M	10M	20M	40M	100M
Size (GB)	1.43	14.5	28.5	1.62	3.22	6.69	16.2
# Segments SPO	13,002	39,855	27,464	11	11	11	11
# Segments OPS	641	1,088	1,520	13	13	13	13

before finding the star-query matches. The execution was already described in last section when we described the Volcano execution model of our system. The final result is obtained joining the mappings obtained from the different star queries of the plan, we obtain the final result.

4 Experimental evaluation

We evaluated the performance of our system with two well known RDF benchmarks (Watdiv and LUBM). We assesed firstly the ability of the systems to load data greater than the available main memory. Then, we evaluate its performance in terms of

4.1 Experimental setup

Hardware: We conducted all experiments on a virtual machine with an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz CPU, 1TB hard disk and 32GB of RAM running Ubuntu server 16.04 LTS.
Software: The main components of QDAG (fragmentation, allocation and indexing modules) are implemented in Java. The core of the execution unit selecting the best execution plan for a given query was coded on C++.
Compared systems: Our system was compared with two state-of-the-art approaches that apply different execution paradigms. First a graph-based execution system gStore[6] and then a DBMS-based system Virtuoso[7].
Datasets: We evaluate and compare the performance of the systems using the Watdiv and LUBM benchmarks. We compare the execution time to solve queries with different configurations (Linear, Star, Snowflake and Complex). The list of queries is not here for space reasons but it is found in our technical report in [11]. Also, we compare the ability of the systems to deal with datasets of different sizes. We generated datasets with 10, 100 and 200 million triples for Watdiv and datasets of 10, 20, 40 and 100 million triples for LUBM. The size of each dataset is detailed in Table 2.

[6] https://github.com/pkumod/gStore
[7] https://github.com/openlink/virtuoso-opensource

4.2 Pre-processing evaluation

We tested first the ability of the systems to pre-process and load raw RDF datasets (in the N-Triples format). QDAG and Virtuoso were able to load all of the datasets for the Watdiv framework successfully. On the contrary, gStore was unable to load the dataset of 200 million triples. This is mainly due to the fact that gStore performs the pre-processing in main memory and is unable to load RDF graphs that do not fit into it. Virtuoso loads the dataset to a relational database, and QDAG creates files of segments (SPO and OPS). The number of SPO and OPS segments on each dataset is shown in Table 2. The number of segments SPO and OPS does not grow exponentially and their number remains reasonable to the size of the data. execution time.

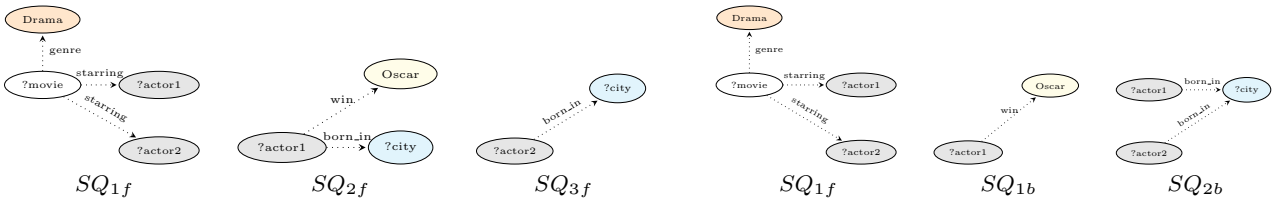
4.3 Query performance

4.3.1 Watdiv

The query performance for linear (L), star (S), snowflake (F) and complex (C) queries is shown in Figure 9. We plot using a logarithmic scale since the performance of QDAG is on average 300x better than gStore, leaving the original execution times would have led to unreadable graphs. The results for 10 million triples of Watdiv are shown in Figure 9a, even if QDAG obtains very similar performance than Virtuoso for star and linear queries, the execution model scales to more complex datasets and queries. This is proven with the complex and snowflakes queries in which our system is on average 1.6X times faster. The behaviour of the same queries in a 100 million dataset is very similar, QDAG is able to solve much more complex queries that are hardly transformed into SQL with a reasonable performance.

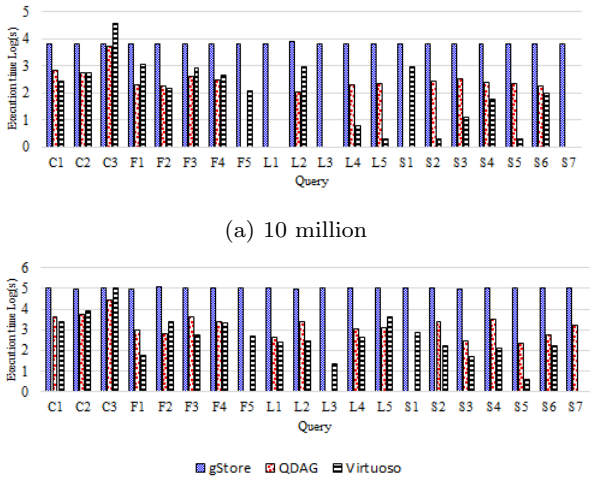
4.3.2 LUBM

Similarly to what was done for Watdiv, we evaluated the ability of the systems to load different sizes of datasets. gStore was unable to load the dataset of 100 million triples since it is greater than the available main memory. The query performance for the datasets of 10, 20 and 40 million triples are shown in Figure 10. In all cases, our system outperformed gStore solving queries on average more than 10x faster.



(a) Plan 1: $\{SQ_{1f}, SQ_{2f}, SQ_{3f}\}$ (b) Plan 2: $\{SQ_{1f}, SQ_{1b}, SQ_{2b}\}$

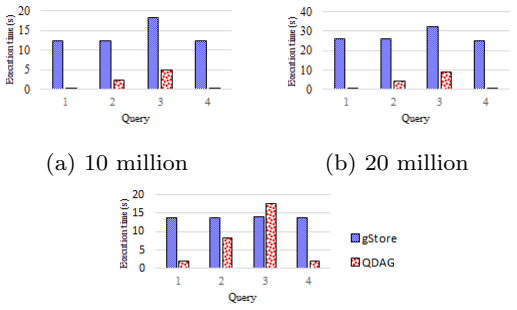
Fig. 8: Query execution plan examples



(a) 10 million

(b) 100 million

Fig. 9: Query performance for Watdiv benchmark



(a) 10 million (b) 20 million

(c) 40 million

Fig. 10: Query performance for LUBM benchmark

4.3.3 Yago2

4.3.4 DBLP

5 Conclusion

In this paper, we propose a new technique for evaluating queries on RDF data. Its main particularity is that it allows combining graph exploration and fragmentation – crucial issues in RDF data repositories. Our results are encouraging and showed that our proposal outperforms gStore system considered as the state of the art in the processing of RDF data.

This work opens several research directions. Currently, we are conducting intensive experiments to evaluate the scalability of our approach. Another direction consists in studying the ordering of the star queries study, by proposing adequate cost models. Finally, we plan to parallelize our approach. Since it already includes the fragmentation process, a new module that has to be developed concerns the management of the transfer of intermediate results between fragments.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.
2. H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1):115–146, 1989.
3. M. Atre, J. Srinivasan, and J. A. Hendler. Bitmat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC2008), Karlsruhe, Germany, October 28,*, 2008.
4. K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of ACM SIGMOD*, pages 1247–1250. AcM, 2008.
5. M. Briggs. Db2 nosql graph store what, why & overview, 2012.
6. R. Cyganiak. A relational algebra for sparql. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, page 35, 2005.
7. U. Deppisch. S-tree: a dynamic balanced signature index for office retrieval. In *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 77–87. ACM, 1986.
8. O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
9. G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

10. M. Janik and K. Kochut. BRAHMS: A workbench RDF store and high performance memory system for semantic association discovery. In *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC, Galway, Ireland, November 6-10, 2005, Proceedings*, pages 431–445, 2005.
11. A. Khelil. Combining graph exploration and fragmentation for scalable rdf query processing, technical report, 2019.
12. J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.
13. B. McBride. Jena: A semantic web toolkit. *IEEE Internet computing*, (6):55–59, 2002.
14. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE)*, pages 984–994, 2011.
15. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
16. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, volume 4273, pages 30–43. Springer, 2006.
17. O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, Vancouver, British Columbia, Canada*, pages 1465–1470, 2007.
18. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of VLDB*, 1(1):1008–1019, 2008.
19. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8*, pages 131–150, 2003.
20. L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.