Query Optimization for Large Scale Clustered RDF Data

Ishaq Zouaghi
LIAS/ISAE-ENSMA
Chasseneuil-du-Poitou, France
LR-Sys'Com-ENIT/UTM
Tunis, Tunisia
ishaq.zouaghi@ensma.fr

Amin Mesmoudi
University of Poitiers
Poitiers, France
amin.mesmoudi@univ-poitiers.fr

Jorge Galicia
LIAS/ISAE-ENSMA
Chasseneuil-du-Poitou, France
jorge.galicia@ensma.fr

Ladjel Bellatreche
LIAS/ISAE-ENSMA
Chasseneuil-du-Poitou, France
bellatreche@ensma.fr

Taoufik Aguili
LR-Sys'Com-ENIT/UTM
Tunis, Tunisia
taoufik.aguili@enit.utm.tn

ABSTRACT

The popularity of the Resource Description Framework (RDF) and SPARQL has thrust the development of high-performance systems to manage data represented with this model. Former approaches adapted the well-established relational model applying its storage, query processing, and optimization strategies. However, the borrowed techniques from the relational model are not universally applicable in the RDF context. First, the schema-free nature of RDF induces intensive joins overheads. Also, optimization strategies trying to find the optimal join order rely on error-prone statistics unable to capture all the correlations among triples. Graph-based approaches keep the graph structure of RDF representing the data directly as a graph. Their execution model leans on graph exploration operators to find subgraph matches to a query. Even if they have shown to outperform relational-based systems in complex queries, they are barely scalable and optimization techniques are completely system dependent. In this paper, we propose optimization strategies for graph-based RDF management systems. We intend to take the strengths of relational databases and propose logical structures generically depicting graph-based query execution. First, we define novel statistics collected for clusters of triples to better capture the dependencies found in the original graph. Second, we redefine an execution plan based on these logical structures. Finally, we introduce an algorithm for selecting the optimal execution plan based on a customized cost model.

KEYWORDS

Optimization, RDF, SPARQL, Cardinality Estimation, Cost Model.

1 INTRODUCTION

The versatility of the Resource Description Framework (RDF) has contributed to its rapid expansion not only as a standard data model in the semantic Web but also as the preferred representation for data from diverse domains (e.g. genetics, biology). The RDF model uses triples consisting of a subject, a predicate and an object $< s, p, o >$ to represent data and SPARQL as its query language. Currently, public RDF data sets (known as knowledge bases) with billions of triples are extensive sources of information (e.g. DBPedia[1], Bio2RDF[2]) popularly queried and aggregated. As the volume of available RDF data grows, the need for high performance RDF data management systems becomes more noticeable.

To cope with the proliferation of RDF datasets, early works used the well-established relational model as backend, storing RDF triples directly into tables (single-table [3, 10], vertical partitioning [1]). In these approaches, a SPARQL query, generally specified as a sequence of triple patterns (TPs), is mapped to a SQL statement. Although, considerable research efforts were dedicated to these relational-based strategies, they rapidly suffered from many intensive join overheads induced by the schema-free nature of RDF. Some optimization strategies borrowed from the relational model strive to reduce the evaluation cost by finding an optimal execution order of a query expressed as a sequence of TPs. For example, deciding the optimal join order for the TPs of the query shown in Figure 2 (e.g. $[(tp_1 \bowtie_{?f} tp_2) \bowtie_{?f} tp_3]$)

Furthermore, these strategies are not universally applicable in the RDF context. Firstly, because they rely on statistics that are very error-prone since they are gathered on the entire collection of data (contrary to the relational model where statistics are calculated per table entity). Capturing statistics for datasets without an explicit schema is not a simple task. Former approaches collected statistics at the predicate level and assumed independence between triples. However, as shown later by [6, 11] these assumption led to considerable underestimations since RDF triples are highly correlated. Capturing these correlations may prompt to exponentially huge statistics whose maintenance is very complex. Although, some heuristics have been introduced [20], they are still not sufficient to estimate the cardinalities of complex queries involving several joins. Moreover, these query optimization strategies do not tackle the leading issue which is the intensive joins product of the unsuited direct mapping of RDF to tables. Even with an optimal join order, the join operation would still be the bottleneck at query runtime especially for complex queries (which are more and more frequent in SPARQL).

Graph-based processing systems (e.g. [22]) keep the graph structure of RDF representing the data directly as a graph. In these systems, the graph essence of RDF is maintained and query processing is turned into a subgraph matching problem. They outperform relational-based systems when solving complex queries [9, 22]. However, as shown in [2, 9] they are less scalable since their processing is mostly based on main memory. There have not been *generic* optimization strategies specifically built for these systems. There is not a single logical layer enclosing their execution model (as a graph exploration) and their data organization; without a common scheme optimization strategies will still be completely system dependent. Even though the relational-based

[1] https://wiki.dbpedia.org/
[2] https://bio2rdf.org/

optimization strategies were not ideal, they offered some comfort to the designer since they allowed to organize the execution regardless of how the data were stored on disk (i.e. if the data are stored in a single table, binary tables).

In this paper, we propose optimization strategies for graph-based RDF management systems. Our strategies fit both centralized and distributed approaches. We took the strengths of the logical execution modeling from the relational databases and propose first logical structures to portray the query execution based on the exploration of the query and input graphs. We redefine an execution plan based on these structures and present an algorithm that generates and picks based on a cost model the optimal execution plan for a given query. Our cost model relies on statistics, but in contrast to former approaches estimating statistics on the global graph we collect them for clusters of triples (that we named graph fragments Gf) logically connected in the original graph. Our cost model considers the interactions between graph fragments to estimate the network and disk costs of a given query plan.

The contributions of the paper are summarized as follows:

(1) We formalize a logical model describing the query execution of RDF systems based on graph exploration methods.
(2) We present the essential statistics collected for each graph fragment.
(3) We detail what is to the best of our knowledge the first cost model to compare execution plans based on the disk and network costs in graph-based systems.
(4) We present a study of the problem allowing to choose the optimal execution plan. We prove its complexity and proffer a branch and bound like algorithm to efficiently explore and select the optimal execution plan in terms of our logical structures.

The rest of the paper is organized as follows. First, Section 2 presents the state of the art of the optimization strategies proposed for RDF systems. Then, Section 3, introduces the preliminary definitions used to describe the query execution based on graph exploration. Section 4 presents the cost model allowing to compare the equivalent plans and introduces an algorithm to find the best execution plan. Next, section 5 presents the experimental study. Finally, Section 6 summarizes the work and gives insights on future researches.

2 RELATED WORK

In this section, we summarize the most relevant optimization strategies adopted by triple stores in the state of the art. We classify them according to whether the strategy is applied *before* or *during* the query execution (*BQE* and *DQE* respectively). In the first category we consider approaches of organization, indexing and distribution of data; all of them have been implemented by different systems with the aim of finding the best query performance. The second category depicts optimization strategies at query runtime.

2.1 BQE strategies

2.1.1 Data organization. The earliest RDF processing systems adapted the prominent relational model. The naïve approach embraced by Sesame [3] stores the data in a single table of three columns (subject, predicate, object). Its major drawback is the processing of self-joins that turns quite expensive when SPARQL queries become more complex. The property table approach (Jena2[10]) reduces the number of self-joins storing the data in a wider table whose dimensions correspond to the number of distinct subjects and predicates. The overheads of this approach are the great number of null values and the treatment of multi-valued properties. The vertical partitioning approach proposed in SW-Store[1] overcomes this drawback storing the data in n binary tables, where n is the number of distinct predicates. Still, overheads exists when many predicates are involved in a single query. Most recent approaches have tried to find the implicit schema of the data in an RDF dataset. These approaches use the characteristic sets [11] to distinguish entities and store the data of similar entities together (e.g. EAGRE [21], [15]). Other approaches maintain the graph structure of RDF data representing the data as adjacency lists (e.g. gStore [22]). The main disadvantage of this approaches is related to scalability to large RDF graphs [9].

2.2 DQE strategies

Before describing optimization strategies applied during query execution, let us classify SPARQL evaluation approaches. In centralized systems, the evaluation is either join-based (e.g. [10]) or graph-matching based (e.g. [14]). The first approaches comprise all systems translating each single graph pattern into SQL and combining the results on each iteration using the join operation (on a single or multiple tables). Indexing the data enhance the performance since the joins are performed as merge-joins (e.g. [12]). Graph matching approaches on the other side break a SPARQL query into subgraphs, and to avoid invalid intermediate results since at every iteration only valid subgraph bindings are kept.

The static query optimization strategies use maintained statistics about the data to determine an optimal query plan. The estimation of the cardinality is the base measure used to evaluate and compare execution plans for a specific query.

Cardinality Estimation in RDBMS. It has been longly seen as a key component in the query optimization and it is a well established field in the the relational database world[8]. It is usually solved by using various summarization techniques such as one-dimensional synopsis (e.g. one-dimensional histograms[17]) Even if the cardinality estimation used in the relational model would seem useful for the semantic Web, its estimation has been less successful due to the heterogeneous, string-oriented nature and to the fact that queries in RDF contain many self-joins [16].

Cardinality Estimation in SPARQL. Currently, several studies have investigated the cardinality estimation issues for SPARQL queries. A line of work uses very simplistic models based on RDF-specific statistical synopses including counters of frequent predicate-sequences in paths of the data graph[13]. Similarly, other approaches use one-dimensional histograms and pre-compute the number of occurrences of all predicates pairs to estimate the triple pattern and joined triple patterns selectivities[19]. The first approach was implemented in RDF-3X and the second in Jena ARQ optimizer. The drawback of these approaches is that the formulas assume statistical independence between tuples, which produce large estimation errors[6]. The second line of work introduced a specific kind of summary based on a schema-level synopsis for RDF data while preserving as much of its structure as possible[18]. Finally, the third line of approaches collected statistics for tuple groups based on characteristic sets[6, 18], or by summarizing the graph into large entities[7].

Contrarily to existing techniques, the set of strategies proposed in this paper are independent of the physical storage and query evaluation models of any system. Our assumptions are only that the data are logically clustered based on the predicates and that the main query evaluation operator is based on a graph exploration strategy. We propose a set of techniques allowing to find the best way to explore the data graph in order to evaluate a SPARQL query. We rely on a novel cost model that takes into account the correlation between predicates and nodes. Our proposal is not only adapted to centralized systems but also to parallel systems that rely on graph exploration as query evaluation technique. In this kind of systems, our cost model will consider the interactions between fragments to estimate the disk and network costs.

3 PRELIMINARIES

3.1 RDF and SPARQL

The Resource Description Framework (RDF) has been widely accepted as the data model for the Linked Open Data and the semantic Web. The model uses *triples* consisting of a subject, a predicate and an object $< s, p, o >$ as its main abstract structure. The model provides flexibility without explicitly enforcing a schema. A collection of interlinked RDF triples could be represented as a graph as shown in Figure 1. The graph of the example contains data related to air traffic control. The RDF graph is formally defined in Definition 3.1.

Definition 3.1. (RDF Graph) An RDF graph is denoted as $G = \langle V_c, L_V, E, L_E \rangle$ where V_c is a collection of vertices corresponding to all subjects and objects, 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 .

SPARQL is the most popular query language for RDF. A simple SPARQL query consists of a query form (e.g. SELECT in Figure 2), a Basic Graph Pattern (BGP) and a set of SPARQL operations (e.g. FILTER). A Basic Graph Pattern is composed of triple patterns (TPs). TPs are expressed in a triple form and they are composed of at least one of S, P, O being a variable. An example query with three TPs and its graph representation is shown in Figure 2. In our work, we consider only SPARQL queries with bounded predicates. A SPARQL query can also be represented as a graph as described in Definition 3.2.

Definition 3.2. (Graph Query) A Graph Query is denoted as $Q = (V, L_V, E, L_E)$, where $V = V_p \cup V_c$ is the union of the sets of variable and bounded vertices. L_V is the set of vertex labels, the labels of variable vertices are distinguished with a leading question mark symbol. E and L_E represent the set directed edges between vertices and its labels respectively.

3.2 Overview of Query Evaluation

In this section, we discuss the main definitions allowing to model the logical organization of data in clusters keeping its graph structure. Then, we detail the query evaluation operators based on graph exploration on the clustered data.

3.2.1 Graph storage. In contrast to several of the approaches mentioned in Section 2.1.1 in which the graph structure of the loaded RDF data is broken, we strive to preserve it. The storage model groups RDF data first such that implicit structures within the data are automatically discovered. Data are firstly grouped

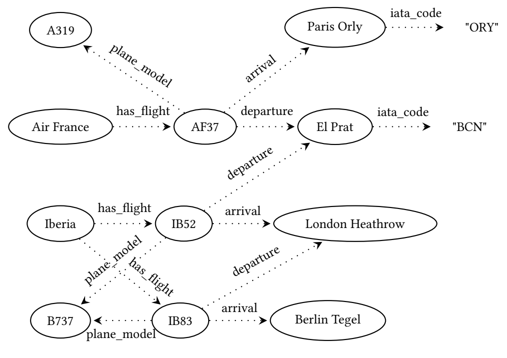
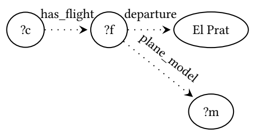


Figure 1: RDF Graph G



```
SELECT ?c ?f ?m WHERE {
?c <:has_flight>  ?f .          #tp1
?f <:departure>   <El Prat> .   #tp2
?f <:plane_model> ?m .          #tp3
}
```

Figure 2: SPARQL query Q

into Data Stars (formally defined in Definition 3.3). Data stars allow identifying the triples related to a specific instance grouping the data by subject (or object).

Definition 3.3. (Data Star) Given a node x (named data star head) in a RDF graph G, a Data Star $DS(x)$ is the set of triples related to node x with a direct edge. We name *Forward Data Star* and *Backward Data Star* to 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.

Data Stars extend the notion of a record in the relational database model. The primary key of a $DS(x)$ corresponds to its head x. Records are grouped in tables in a RDBMS, following this logic we group records describing similar entities into sets named *Graph Fragments.*

When building the Graph Fragments, ontologies could be applied since they intend to provide an overall schema of the data stored in an RDF graph. However, several studies show that there is still a very partial use of ontology classes and sometimes subjects share triples with properties coming from different ontological sources [15]. Consequently, we decided to group Data Stars in Graph Fragments based on the combination of properties characterizing an entity using Characteristic Sets [11].

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 the objects we define $\overleftarrow{cs}(o) = \{p | \exists_s : (s, p, o) \in G\}$. A Forward Graph Fragment $\overrightarrow{Gf}$ groups Forward Data Stars having the same characteristic set. Backward Graph Fragment $\overleftarrow{Gf}$ are formed similarly. Its formal definition is given in Definition 3.4.

Definition 3.4. (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 $\overleftarrow{Gf} = \{\overleftarrow{DS}(x) | \forall_{i \neq j} \overleftarrow{cs}(x_i) = \overleftarrow{cs}(x_j)\}$.

It is shown that indexing and compressing the data of fragments in B+Trees improves significantly the performance at query runtime[9, 12]. In the rest of the paper we assume that the data are indexed using this structure, however the estimations and the cost model are easily generalized to other data structures. Additionally, the data could be stored as Forward Graph Fragments, Backward Graph Fragments or using both structures. In the definitions of the following section, we assume that both types of fragments are available, yet this is not a mandatory condition.

3.2.2 Query execution. In this section we formalize the logical structures used to describe the query evaluation. As previously mentioned, a SPARQL query can also be represented as a directed graph whose nodes are either variables (e.g. ?f, ?m in Figure 2) or bounded values (e.g. `<El Prat>`). Let us first recall how a SPARQL query is evaluated in most of the state-of-the-art systems. Traditionally, a SPARQL query is evaluated in a TP by TP manner [4]. A query execution plan is then seen as a join of TPs on a variable. For example, an execution plan for the query of Figure 2 composed of three single triple patterns is:

$$tp_1 \bowtie_{?f} tp_2 \bowtie_{?f} tp_3$$

This representation can become quite complex when several TPs are involved in the query. Optimization strategies for these approaches seek to find the optimal execution order of triple patterns according to pre-computed statistics.

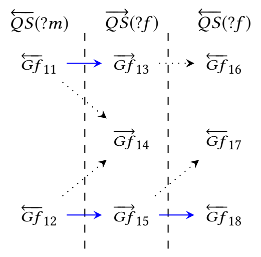
To shorten the logical query plan, TPs can be grouped if they share a common variable on the subject (or object). We name these structures Forward Query Stars and Backward Query Stars if they group the triples on subject or object respectively. Furthermore, as it will be shown later, with these structures the query execution can be easier tracked following a graph exploration approach. Both structures are formally described in Definition 3.5.

Definition 3.5. (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 graph stars and qs to denote indistinctly a forward and backward query star.

The execution of a query can be expressed as a join of Query Stars. Since we consider two copies of the data, (one copy stored as $\overrightarrow{Gf}$ and another as $\overleftarrow{Gf}$), the execution plans consider both types of Query Stars. An execution plan is composed of a sequence of joined Query Stars as shown in Definition 3.6.

Definition 3.6. (Execution Plan) An execution plan is an order function applied on a set of Query Stars. The function denotes the order in which the mappings for each Query Star will be found. We denote by $\mathcal{P} = [QS_1, QS_2, ..., QS_n]$ the plan formed by executing QS_1, then QS_2,..., and finally QS_n.

Let us consider for instance that the optimal query plan for the query of Figure 2 is $\mathcal{P}_1 = [\overleftarrow{QS}(?m)_1, \overrightarrow{QS}(?f)_2, \overleftarrow{QS}(?f)_3]$. The execution engine starts processing $\overleftarrow{QS}(?m)_1$ by loading all the *backward* fragments whose characteristic set contains all the predicates (in this case only the `plane_model` predicate) of $\overleftarrow{QS}(?m)_1$. Assuming that the backward fragments $\overleftarrow{Gf}_{11}$ and $\overleftarrow{Gf}_{12}$ are the only backward fragments matching the predicates of the query star, the execution scans both fragments and finds the mappings



$\overleftarrow{QS}(?m)$
$\overrightarrow{QS}(?f)$
$\overleftarrow{QS}(?f)$
$\overleftarrow{Gf}_{11}$
$\overrightarrow{Gf}_{13}$
$\overleftarrow{Gf}_{16}$
$\overrightarrow{Gf}_{14}$
$\overleftarrow{Gf}_{17}$
$\overleftarrow{Gf}_{12}$
$\overrightarrow{Gf}_{15}$
$\overleftarrow{Gf}_{18}$

Figure 3: Execution Plan

to the variables `?m` and `?f`. These mappings are sent to the forward graph fragments whose characteristics match the predicates of the second Query Star $\overrightarrow{QS}(?f)_2$. In this way, only the pertinent mappings with respect to $\overleftarrow{QS}(?m)_1$, $\overrightarrow{QS}(?f)_2$ are kept. The process continues similarly for the following query stars.

It is evident that an Execution Plan represents a way to explore the graph. Indeed, finding the optimal execution plan conveys to find the best way to explore the data graph. In the next section we detail the optimization strategy followed to find the optimal query plans in terms of Query Stars. We define an acceptable query plan and then depict the algorithm used to generate them. Then we present the cost model in terms of disk and network cost applied to decide on the optimal plan.

4 GRAPH-BASED QUERY OPTIMIZATION

In this section, we present our cost-based optimization strategy which allows comparing execution plans (based on Query Stars). Firstly, we define an acceptable plan and we detail the statistics allowing to evaluate the cost of an execution plan based on the disk and network cost. Next we define the problem of finding an optimal plan and we describe a branch and bound based algorithm used to generate the list of candidate execution plans.

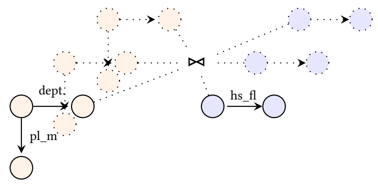
4.1 Acceptable Execution Plan $\mathcal{AP}$

In the last section, we defined an Execution Plan $\mathcal{P}$ as an order function applied to a set of Query Stars. An execution plan $\mathcal{P}$ is called an Acceptable Execution Plan if it fulfills the following conditions:

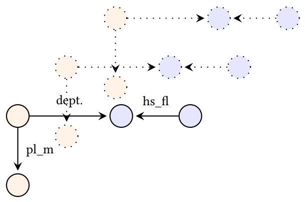
(1) *Coverage:* All nodes and predicates of the given query are *covered* by the set of Query Stars of the plan. For example, for the query of Figure 2, the execution plan $[\overleftarrow{QS}(?m), \overrightarrow{QS}(?f)]$ is not a valid plan since the node `?c` and the edge `has_flight` are not covered by the plan.
(2) *Instantiated head:* This condition guarantees that for a plan $\mathcal{P} = [SQ_1, ..., SQ_n]$, $\forall_{i>1} SQ$, the head of the SQ_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, the plan shown in Figure 4a is not acceptable since the head of the second query star $(\overrightarrow{QS}(?c)$ is not instantiated before finding the mappings for this query star. A plan in which the head has been instantiated is shown in Figure 4b.

The formal definition of an Acceptable Plan is given in Proposition 4.1.

Proposition 4.1. *(Acceptable Plan) $\mathcal{AP}$ Let us consider Q as a given query, $\overrightarrow{QS}$ and $\overleftarrow{QS}$ as the sets of forward and backward*



(a) $[\overrightarrow{QS}(?f), \overrightarrow{QS}(?c)]$



(b) $[\overrightarrow{QS}(?f), \overleftarrow{QS}(?f)]$

Figure 4: Head instantiation examples

graph star queries respectively, T as 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 graph.*
- *Nd: $q \cup \overrightarrow{QS} \cup \overleftarrow{QS} \rightarrow V$ It returns the nodes of a graph.*
- *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 order function of query stars such that:*

(1) $\bigcup_{QS \in X} Tr(QS) = Tr(q)$

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

4.2 Cost model

In this section, we present a novel cost model used to compare execution plans $\mathcal{P}$. As for distributed databases, the cost is expressed with respect to the estimated total time. The total time is the sum of all time components (CPU, I/O, Communication), however, the I/Os and the communication costs are generally the dominant factors. Our cost estimation considers both the disk and communication costs for each query star on the plan as shown in Equation 1. The parameters $T_{I/0}$ and T_{TR} are the time of a disk I/0 and the time to transmit a data unit from one site to another respectively. The estimated number of of I/O's and network packets transmitted are represented by $\mathcal{DC}$ and $\mathcal{NC}$ and their calculation is described in the next sections.

$$Total\_Cost(\mathcal{P}) = \sum_{qs \in \mathcal{P}} T_{I/0} * \mathcal{DC}(qs) + T_{TR} * \mathcal{NC}(qs) \quad (1)$$

Both the disk and network costs are estimated based on statistical information about each graph fragment Gf (Definition 3.4). For the disk cost, the statistics allow estimating the number of Data Stars loaded on each fragment that contains potential query matches. Likewise, for the network cost, the number of intermediate results exchanged between fragments at different sites is estimated based on the same statistics. In the next section, we describe the statistical data collected for each graph fragment.

4.2.1 *Fragment statistics.*

We rely on statistical data are collected for each Graph Fragment Gf_k. The statistical data collected

Table 1: Collected Statistics

	Statistic	Description
(a)	$dist(Gf_k)$	# of data stars in Gf_k.
(b)	$count(p_i, Gf_k)$	# of edges with predicate p_i in Gf_k.
(c)	$dist\_NE(p_i, Gf_k)$	# of distinct nodes linked to the data star heads in Gf_k with respect to p_i.
(d)	$SF(Gf_k, Gf_j, p_i)$	Proportion of the # of nodes in Gf_k pointing to Gf_j with respect to p_i.

for a graph fragment Gf_k (forward or backward) whose characteristic set is $cs = \{p_1, ..., p_m\}$, considering that $p_i \in cs$ are summarized in Table 1. Some examples of statistics for the example graph of Figure 1 are given in Figure 5.

Let us consider the statistics shown in Figure 5. The statistics for the backward graph fragment $\overleftarrow{Gf}_4$ are shown in Figure 5a. The $dist(\overleftarrow{Gf}_4)$ is 3 since the fragment contains three data stars whose heads are `AF37`, `IB62` and `IB83`. Both $count(p_i, Gf_k)$ and $dist\_NE(p_i, Gf_k)$ are calculated only for the predicate `has_flight` (identified as 1 in the Figure) since it is the only predicate in the fragment's characteristic set. There are three edges in the fragment having `has_flight` as a predicate, therefore $count(p_i, Gf_k) = 3$. For this fragment, $dist\_NE(p_i, Gf_k) = 2$ since there are two distinct nodes (`Air France` and `Iberia`) linked to the data star heads of $\overleftarrow{Gf}_4$.

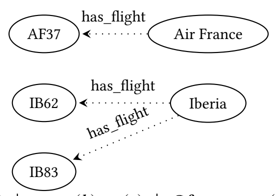
The *selectivity factor* $SF(Gf_k, Gf_j, p)$ between two graph fragments with respect to a predicate is the ratio of the number of edges in a node pointing to the data star's head located in another fragment. For example, in Figure 5c, the selectivity factor $SF(\overleftarrow{Gf}_5, \overleftarrow{Gf}_4, 2)$ (2 is the id of the predicate `arrival`) is 2/2 since out of the 2 edges of the predicate `arrival` in $\overleftarrow{Gf}_5$, 2 nodes (`AF37` and `IB83`) are the heads of data stars located in $\overleftarrow{Gf}_4$.

For simplicity and to keep the same notation, the selectivity of two graph fragments sharing the same head is represented as $SF(Gf_k, Gf_j, -1)$. For example, the selectivity factor $SF(\overrightarrow{Gf}_3, \overleftarrow{Gf}_5, -1)$ in Figure 5b is 1/2 since out of the two heads in $\overrightarrow{Gf}_3$, one of them is a head of a data star in the graph fragment $\overleftarrow{Gf}_5$ (shown in Figure 5c).

4.2.2 *Disk cost.*

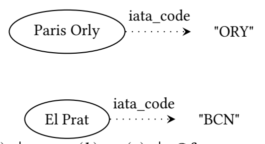
We present in this section the different formulations that allow estimating the number of pages targeted by a plan. We assume that the data on each fragment are stored as a clustered B+Tree as done in [9, 12]. Our disk cost is related to this data structure. However if the fragments were stored using another data structure, only the parameters of the function calculating the number pf disk pages ($\mathcal{N_{DP}}$) would change. The disk cost for a single query star is given in Equation 2. In this equation, the $\mathcal{NP}$ function allows estimating the number of pages targeted by a query star in a fragment and, the tuple (Gf_j, k_j) represents the estimation of the number of data stars k_j in the fragment Gf_j involved in the evaluation of such a query. In the following sections, we detail the function $\mathcal{NP}(Gf, k)$ and we present the formulations to estimate the number of data stars read on each graph fragment (k).

$$\mathcal{DC}(qs_i) = \sum_{(Gf_j, k_j) \in \, input_{qs_i}} \mathcal{N_{DP}}(Gf_j, k_j) \quad (2)$$



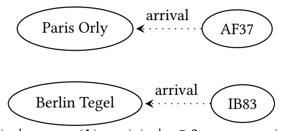
(a)	p_i	(b)	(c)	Gf_j	p_i	(d)
3	1	3	2	1	1	2/3
				2	-1	3/3

(a) $\overleftarrow{Gf}_4$



(a)	p_i	(b)	(c)	Gf_j	p_i	(d)
2	5	2	2	5	-1	1/2
				6	-1	1/2
				8	5	2/2

(b) $\overrightarrow{Gf}_3$



(a)	p_i	(b)	(c)	Gf_j	p_i	(d)
2	2	2	2	2	2	2/2
				3	-1	1/2
				4	2	2/2

(c) $\overleftarrow{Gf}_5$

Figure 5: Subset of statistics for G of Figure 1

The number of pages per fragment. Let us now detail the function $\mathcal{NP}$ estimating the number of pages read from the disk given a graph fragment Gf_j and a number of data stars k_j as inputs:

$$\mathcal{N_{DP}}(Gf_j, \mathbf{k}) = \sum_{i=1}^{N_{Gfj}} r_f(i)$$

where,

$$r_f(i) = \begin{cases} \frac{\mathbf{k}}{N_{Gf_j}} * B_{Gf_j} & if\ i = H_{Gf_j} \\ r_i * r_f(i+1) & otherwise \end{cases}$$

The parameters and properties in the previous function are defined as follows:

(1) B_{Gf_j}: number of disk pages in the last level.
(2) H_{Gf_j}: number of levels.
(3) r_i: reduction factor for the i^{th} level. Given two levels "i" and "$i+1$", and "Z" and L as the number of pages for the level "i" and "$i+1$" respectively, r_i is defined as by Z/L.
(4) N_{Gf_j}: number of data stars in the fragment Gf (i.e., number of keys in the leaf level of the B+Tree).
(5) $r_f(i)$: is the number of pages to be manipulated at the i^{th} tree level.

As it is shown in last equation, the number of pages $\mathcal{NP}$ is the sum of the number of pages manipulated at all levels.

The number of data stars k. In this part we explain the procedure to obtain the number of data stars per each star query (represented as k). To better understand the procedure, let us recall the graph exploration execution model illustrated in Figure 3. For a given plan, the execution is done finding the mappings of one query star after another. This execution model guides the estimation of data stars per query stars. We start calculating the number of data stars for the first query star (*Input*). Then we estimate the number of data stars that we get after executing this query star(*Valid Input*). Next, we estimate the number of data stars sent to the next query star for each predicate (*Output*). Finally, using the selectivity factor and the output, we calculate the input of the next query star. This procedure continues until the last query star.

Let us introduce the estimation of the *Input* of the first query star. We distinguish two cases:

(i) If the head of the star query is a variable:

$$input\_DS_{qs_1} = \{(Gf_j, k) | Gf_j \models qs_1 \wedge k = dist(GF_j)\}$$

(ii) If the head of the star query is a constant:

$$input\_DS_{qs_1} = \{(Gf_j, 1) \mid Head(qs_1) \in GF_j \wedge Gf_j \models qs_1\}$$

The input of a query star is expressed as a set of tuples (Gf_j, k_j). Gf_j is a Fragment satisfying the predicates of qs_1 and k_j is the number of data stars targeted by this query.

The *Valid* input data stars is calculated for each $(Gf_j, k) \in input\_DS_{qs_i}$ and it is expressed as follows:

$$valid\_DS_{qs_i} = \{(Gf_j, k') \mid k' = \lceil k * \min_{e \in Edges(qs_i)} f(e) \rceil\}$$

Where

$$f(e) = \begin{cases} \frac{1}{dist\_NE(e.label, Gf_j)} & , if\ e.node\ is\ constant \\ 1 & , otherwise \end{cases}$$

In the $f(e)$ function, we calculate a reduction factor for each predicate ($e.label$) to consider an estimation of the number of data stars found in the fragment after the execution of the star query.

For each tuple $(Gf_j, k') \in valid\_DS_{qs_i}$, we calculate the $output\_DS_{qs_i}$ expressed as a set of triplets (Gf_j, p_i, k'') where Gf_j is the fragment from the input, p_i is a predicate of the qs_i and k'' is the number of distinct $edge.node$ related to p_i with respect to the valid input. The *Output* is calculated as follows:

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

Where

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

NDS_{p_i} is the number of data stars head related to each predicate.

After computing the output of the first query star, We can now compute the input of the second query star. For each star order greater than one in the plan, the input is computed as follows:

$$input\_DS_{qs_i} = \{(Gf_j, k) \ / \ Gf_j \models qs_i \wedge k = \lceil NDS \rceil\}$$

To calculate the *Input* we of qs_i if the head of the query star is a variable we consider two cases:

(i) Neighbor star queries: In this case, the last evaluated query star is a *neighbor* of the current query star, therefore in this case we can use the selectivity factors. The data stars heads targeted in the fragment "Gf_j" are computed as follows:

$$NDS = \sum_{(Gf_k, p, k'') \in output_{qs_{i-1}}} k'' * SF(Gf_k, Gf_j, p)$$

Where p is the edge between qs_i and qs_{i-1}.

(ii) Same head star queries: In this case, the current query star has the same head of the last evaluated query star so there is no link between the star queries. In this case, when $Head(qs_i)$ is restricted, the number of data stars is equal to the product

between the selectivity of the query star head in the fragment that satisfies qs_i.

$$NDS = \sum_{(Gf_k,k') \in valid\_input_{qs_{i-1}}} k' * SF(Gf_k, Gf_j, -1)$$

If the head of the query star is a bounded value, the input is calculated similarly to the first query star:

$$input\_DS_{qs_i} = \{(Gf_j, 1) \mid Head(qs_i) \in GF_j \land Gf_j \models qs_i\}$$

4.2.3 Net cost. We present in this section the formulations to estimate the number of network packets exchanged between fragments in different machines. Unlike the Disk Cost in which we estimate the number of data stars loaded by each graph fragment in a query star, the Network Cost aims to estimate the number of *mappings* sent from one graph fragment to another.

As a recall, a mapping is a binding between the nodes of the query star and the corresponding values in the input graph. For example, the mappings of the first query star of Figure 3 are the bindings for variables ?m and ?f in the input graph (e.g. ?m $\longrightarrow$ B737, ?f $\longrightarrow$ IB83). As illustrated in Figure 3, these mappings are sent to the graph fragments of the following query stars. The total network cost is equivalent to the sum of mappings exchanged between fragments located in different sites (illustrated with dotted arrows in Figure 3).

The network cost is shown in Equation 3 as the sum of exchanged packages between graph fragments. The function $\mathcal{N}_{\mathcal{NP}}$ returns the number of packages exchanged between two fragments given a number of mappings M and its size S.

$$\mathcal{NC}(qs_i) = \sum_{\substack{(Gf_k, Gf_j, M) \\ \in\, output\_MP_{qs_i}}} \mathcal{N}_{\mathcal{NP}}(Gf_k, Gf_j, M) \tag{3}$$

Number of exchanged packages. Let us detail the function $\mathcal{N}_{\mathcal{NP}}$ calculating the number of packages as the product between the number of mappings M, its size S and the output of the function $loc(Gf_k, Gf_j)$ returning 1 if the fragments are located on distinct sites and 0 otherwise. Its formulation is as follows:

$$\mathcal{N}_{\mathcal{NP}}(Gf_k, Gf_j, M) = \Big(M * size_i * loc(Gf_k, Gf_j)\Big) / size_{pack}$$

The parameter $size_i$ represents the size of a single mapping for the current query star and all the previously found mappings and $size_{pack}$ indicates the size of a single package transmitted over the network.

The estimation of the number of mappings M from one query star to another is done at the graph fragment level of each query star. We start calculating the number of data stars loaded per graph fragment for the first query star (*input_MP*). Then, based on this input, we estimate for each graph fragment the number of mappings produced after the execution of the first query star (*valid_MP*). These mappings, named *valid mappings*, are multiplied by the selectivity factors between the graph fragments of the first and second query stars to obtain the *output mapping* for each pair of graph fragments. The output mapping of the first query star becomes the input of each fragment in the following query star. The next mappings are estimated following the same methodology (calculating the input mappings, valid input mappings and the output mappings for each fragment in the query stars). Next, we detail the formulas used to calculate the mappings at each step.

Input mapping: The input of the first query star is computed similarly as the inputs of the first query star for the disk cost in which we distinguish two cases:

(i) If the head of the query star is a variable:

$$input\_MP_{qs_1} = \{(Gf_j, M) | Gf_j \models qs_1 \land M = dist(GF_j)\}$$

(ii) If the head of the star query is a constant:

$$input\_MP_{qs_1} = \begin{cases} (Gf_j, 1) \mid Head(qs_1) \in GF_j \land Gf_j \models qs_1 \\ (Gf_j, 0) \mid Head(qs_1) \in GF_j \land Gf_j \not\models qs_1 \end{cases}$$

Valid mapping: The valid mapping of the query star is computed using the input mapping calculated in the previous step. The valid input estimates how many variable bindings exist after executing the current query star. For each input $((Gf_j, M))$ we calculate the product between the number of mappings in the input M and the estimated number of mappings after the execution of the current query star. The valid input is calculated for each input (Gf_j, M) as follows:

$$valid\_MP_{qs_i} = \{(Gf_j, M') \mid M' = M * P(qs_i, Gf_j)\}$$

The number of mappings after the execution of the current query star are the product of the mappings for each edge of the query star. For each edge e of the query, we calculate a permutation between n and k where n is the difference of the mean number of edges on each data star of the fragment having the same predicate as $e.label$ and the number of edges in the query star pointing to a constant node. The value of k equals to the number of edges in the query pointing to a variable node. More precisely,

$$P(qs_i, Gf_j) = \prod_{e \in Edges(qs_i)} \frac{n!}{(n-k)!}$$

where,

$$n = \lceil count(e.label, Gf_j)/dist(Gf_j) \rceil - N\_const(e.label, qs_i)$$
$$k = count(e.label, qs_i) - N\_const(e.label, qs_i)$$

and N_const is a function returning the number of edges labeled as $e.label$ in the query star pointing to a bounded node.

Output mapping: After computing the *valid_MP* of the first query star, we calculate the number of exchanged results between fragments using the selectivity factor. This value is calculated for each graph fragment from the current query star qs_i to the graph fragments of the next query star qs_{i+1}. For each valid mapping found previously, the output mapping is a set of triples defined as follows:

$$output\_MP_{qs_i} = \{(Gf_k, Gf_j, M'') | Gf_j \models qs_{i+1} \land M'' = \lceil IntMP \rceil\}$$

The intermediate mappings $IntMP$ is a function returning the number of mappings sent to each graph fragment based on the selectivity factor.

$$IntMP = \begin{cases} M' * SF(Gf_k, Gf_j, p) & , if\ qs_i\ has\ neighbor \\ M' * SF(Gf_k, Gf_j, -1) & , otherwise \end{cases}$$

The input of the following query star is calculated as follows:

$$input\_MP_{qs_i} = \{(Gf_j, M) \mid Gf_j \models qs_i \land M = Nbr\_MP(Gf_j)\}$$

The total number of mappings for a single graph fragment is the sum of all the mappings received from all the graph fragments in the previous query star. It is calculated as:

$$Nbr\_MP(Gf_j) = \sum_{(Gf_k, Gf_i, M'') \in output\_MP_{qs_{i-1}} \land Gf_i = Gf_j} M''$$

4.3 Stars Ordering and Selection Problem

Several execution plans can be used to evaluate a given query. In Section 4.2 we develop a cost model allowing to compare equivalent plans for a query. Finding an optimal acceptable execution plan $\mathcal{AP}^*$ consists in selecting the acceptable plan for a given query such that it minimizes a given cost function (defined in Equation 1). We name this problem as the Stars Ordering and Selection problem since we seek to find the optimal ordering of Query Stars in the plan. Its definition and complexity are given in Proposition 4.2 and Theorem 1 respectively.

Proposition 4.2. *Stars Ordering and Selection (SOS) problem Given a query q, find an acceptable plan $\mathcal{P}^*$ such that:*

$$minimize \quad Total\_Cost(\mathcal{P}^*) \quad (Eq.\ 1)$$

Theorem 1. *The Stars ordering and Selection (SOS) problem is NP-Hard.*

Theorem 1 is explained as follows: our problem is as difficult as the well-known problems belonging to the NP-Hard class. There is no efficient (polynomial) algorithm that can solve this problem. Then we face two cases: either an exact and exponential algorithm or a polynomial and not exact algorithm. In the next section we describe a branch and bound based algorithm allowing to find the optimal query plan based on some parameters. Due to the lack of space the proof of this theorem is found online [3].

4.4 Optimal $\mathcal{P}$ Finding Algorithm

We present in this section our parametric algorithm allowing to find the best plan for a given query. Our algorithm relies on a branch and bound strategy to enumerate candidate solutions. To prune invalid execution plans it relies on the concepts of *Allowed_Stars* and *Star_Distance* defined next.

Allowed Stars. This concept guarantees that all the generated execution plans are acceptable plans $\mathcal{AP}$s. For a given plan X, an *Allowed_Star* is the set containing the query stars (forward and backward) such that any of them can be added to X and produce an $\mathcal{AP}$. For the example query of Figure 2, the *Allowed_Star* set for the plan $[\overrightarrow{QS}(?c)]$ is $\{\overrightarrow{QS}(?f), \overleftarrow{QS}(\texttt{El Prat})\}$ since both plans ($[\overrightarrow{QS}(?c), \overrightarrow{QS}(?f)]$, $[\overrightarrow{QS}(?c), \overleftarrow{QS}(\texttt{El Prat})]$) are acceptable. The formal definition is given in Proposition 4.3.

Proposition 4.3. *(Allowed stars) Let X be a valid plan, the allowed stars set is defined as follows:*

$$Allowed\_stars(X) = \{qs | qs \in \overrightarrow{QS} \cup \overleftarrow{QS} \text{ and } [X, qs] \text{ is an } \mathcal{AP}\}$$

Stars Distance. This user-defined parameter allows skipping some combination that the user does not want to explore based on the concept of distance. The distance between two stars in a query graph is the number of edges separating the heads of the star queries by considering the shortest path. For example, for the query in Figure 2, the distance between the star queries $\overrightarrow{QS}(?c)$ and $\overleftarrow{QS}(?m)$ is 2 since between the heads of both heads there are two predicates (`has_flight` and `plane_model`). The distance between stars allows considering only plans that privilege to evaluate neighbors' stars. The formal definition is given in Proposition 4.4.

Proposition 4.4. *(Stars Distance) Given two query stars $QS(x)$ and $QS(y)$, the distance between both queries is given by:*

$$distance(QS(x), QS(y)) = |\{p | p \text{ is a path between } x \text{ and } y\}|$$

[3] Theorem 1's proof & Experimental Queries: https://www.lias-lab.fr/~amesmoudi/papers/dolap2020/SOS-NP-hardness-proof.pdf

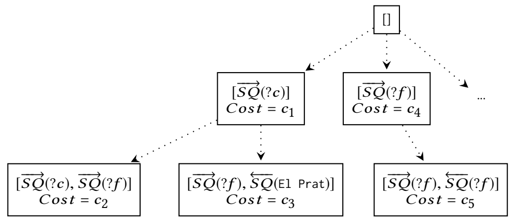

[]
$[\overrightarrow{SQ}(?c)]$ $Cost = c_1$
$[\overrightarrow{SQ}(?f)]$ $Cost = c_4$
...
$[\overrightarrow{SQ}(?c), \overrightarrow{SQ}(?f)]$ $Cost = c_2$
$[\overrightarrow{SQ}(?f), \overleftarrow{SQ}(\texttt{El Prat})]$ $Cost = c_3$
$[\overrightarrow{SQ}(?f), \overleftarrow{SQ}(?f)]$ $Cost = c_5$

Figure 6: Decision Tree for Query of Figure 2

Algorithm overview. The optimal execution plan discovery algorithm follows a branch and bound strategy in which the set of candidate plans is enumerated in a decision tree. The root of the tree contains the union of the sets of forward and backward query stars for a specific query. Each node of the decision tree contains a candidate plan, the *Allowed_stars* set for this plan and its cost (cost function described in Section 4.2). The children nodes are the execution plans resulting from adding a query star from the set of *Allowed_stars* to the execution plan of the parent node. The *Allowed_stars* set is empty if the node's plan is an $\mathcal{AP}$. If the cost of the plan in the node is greater than the cost of the best plan (at that moment), then the node will not be expanded to its children nodes even if there are still query stars in the *Allowed_stars* set. The cost of the best plan is initialized to infinite so the first $\mathcal{AP}$ generated becomes the best plan at an early exploration.

Let us consider the example tree shown in Figure 6 in which the the first star query to be explored is $\overrightarrow{SQ}(?c)$, with $Allowed\_Stars = \{\overleftarrow{SQ}(\texttt{El Prat}), \overrightarrow{SQ}(?f)\}$ and with a cost c_1. Since c_1 is not greater than infinite, we must continue expanding the node to the query stars in the *Allowed_Stars* set of $\overrightarrow{SQ}(?c)$. The tree is expanded and the child node contains the plan $[\overrightarrow{SQ}(?c), \overrightarrow{SQ}(?f)]$ which is an $\mathcal{AP}$ because its *Allowed_stars* is empty. Since the cost c_2 of the plan is lower than infinite, the plan of the node becomes the best query plan (at the moment). The tree continues to expand creating the child $[\overrightarrow{SQ}(?f), \overleftarrow{SQ}(\texttt{El Prat})]$. Assuming that the cost $c_3 > c_2$, the node will not be expanded even if the star query $\overrightarrow{SQ}(?f)$ is still in the *Acceptable_stars* set of the node. The algorithm continues exploring similarly until no more star queries in the root could be expanded and the best possible plan has been found.

The tree exploration technique is given in Algorithm 1. The initialization is done in steps 1-3. We start by creating a node under the root of the decision tree (step 5). Each node of the decision tree is characterized by three elements: the plan, the allowed query stars and the cost. In steps 6-8 we initialize the elements for the node created in step 5. Then in step 9, we call a function named *Add_Query_Star* for each query star. This function is described in Algorithm 2.

The *Add_Query_Star* function (Alg. 2) receives as parameters a query star, the node of the decision tree, the best plan (at the moment) and the stars distance. It starts calculating the node's elements: it adds the query star to the plan (step 1), then calculates the allowed star (step 2) and the cost as defined in Section 4.2 (step 3). Next, if the cost of the plan is smaller than the cost of the plan defined as best plan then we call the *Enumerate_Child_Branch* function (step 5) defined in Algorithm 3. If the cost is greater, then it exits (step 7).

The *Enumerate_Child_Brach* (Alg. 3) function has as inputs the node of the decision tree, the best plan at the moment and the

Algorithm 1: Optimal $\mathcal{P}$ Finding

```
    INPUTS: d: stars distance, $\overrightarrow{QS}(x)$, $\overleftarrow{QS}(x)$: forward and
    backward star queries
    OUTPUT: $\mathcal{P}$: Best Plan
 1: $\mathcal{P}.Plan \longleftarrow$ [ ]
 2: $\mathcal{P}.Cost \longleftarrow \infty$
 3: $T$: decision tree
 4: for $qs \in \overrightarrow{QS}(x) \cup \overleftarrow{QS}(x)$ do
 5:     Create a new node $N$ in $T$
 6:     $N.Plan \longleftarrow$ [ ]
 7:     $N.Allowed\_QSs \longleftarrow$ [ ]
 8:     $N.Cost \longleftarrow 0$
 9:     $Add\_Query\_Star(qs, N, \mathcal{P}, d)$
10: end for
11: Return $\mathcal{P}$
```

Algorithm 2: Add_Query_Star

```
    INPUTS: qs: Query Star,$N$: decision Tree node,$\mathcal{P}$: Best
    Plan, d: stars distance
 1: $N.Plan \longleftarrow N.Plan \cup qs$
 2: $N.Allowed\_QSs \longleftarrow Allowed\_Stars$ (N.Plan)
 3: N.Cost$\longleftarrow$ Total_Cost(N.Plan) ◃ Defined in Sect. 4.2
 4: if $N.Cost < \mathcal{P}.Cost$ then
 5:     $Enumerate\_Child\_Branch(N, \mathcal{P}, d)$
 6: else
 7:     EXIT (i.e., abandon this branch)
 8: end if
```

stars distance. If the set of allowed stars is empty, we consider the plan of this node and its cost as the new best plan and minimal cost respectively (steps 1-3). If there are query stars in the allowed stars set, then for each element that fulfills the distance constraint we call the *Add_Query_Star* function (steps 5-11).

Algorithm 3: Enumerate_Child_Branch

```
    INPUTS: $N$: decision Tree node, $\mathcal{P}$: Best Plan, d: stars
    distance
 1: if $N.Allowed\_QSs$ is empty then
 2:     $\mathcal{P}.Plan \longleftarrow N.Plan$
 3:     $\mathcal{P}.Cost \longleftarrow N.Cost$
 4: else
 5:     for $qs \in N.Allowed\_QSs$ do
 6:         $qs^l \longleftarrow$ last query star in $N.Plan$
 7:         if distance($head(qs^l)$, $head(qs) \leq d$) then
 8:             N' $\longleftarrow$ a copy of N
 9:             $Add\_Query\_Star(qs, N', \mathcal{P})$
10:         end if
11:     end for
12: end if
```

5 EXPERIMENTAL EVALUATION

We conducted our experiments in the QDAG system [9], storing the data as graph fragments and solving queries using a graph-exploration approach. We evaluated firstly the time needed to generate the proposed statistics. We do not give the total loading times for the tested datasets (they are found in [5, 9]), instead we prove that the dimensions and generation time of the proposed statistics is negligible compared to the size and the loading times of the datasets. Next, to study the accuracy of the estimations of data stars and mappings obtained with the cost model of Section 4.2, we compared the predicted value (of data stars and mappings) with the real number of structures exchanged in the solution of the plan considered as best plan. Finally, we evaluated the precision of the algorithm selecting the optimal execution time based on a precision measure that we define in Sect 5.4.

5.1 Experimental setup

Hardware: We conducted all experiments on a machine with an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz CPU, 1TB hard disk and 32GB of RAM. Ubuntu server 16.04 LTS is used as an operating system.
Datasets: We created statistics and evaluate execution times executing queries[3] on real and synthetic datasets. We utilize the popular LUBM (16GB) and Watdiv (15GB) benchmarks as synthetic datasets and a Yago2 (41GB) and DBLP (32GB) as real datasets.

5.2 Collection of statistics

The evaluation of the process of statistics generation are summarized in Table 2. As it is shown, the size of the statistics is very small compared to the actual size of the data, just a few MB for all the datasets. For example, in the Yago dataset (41GB) the statistics are stored in a file of only 82MB (0.2%). The time in minutes to generate the statistics is shown in the column *ST*. The time to generate the statistics is negligible compared to the loading times of the real database (in real datasets it was less than 10% of the loading time). In our case, we intentionally worked with a hardware with limited specifications to prove that the generation of statistics is scalable. We are able to generate the statistics without loading the entire database to main memory.

Table 2: Size of statistics M: millions, ST: Statistics, LT: loading time

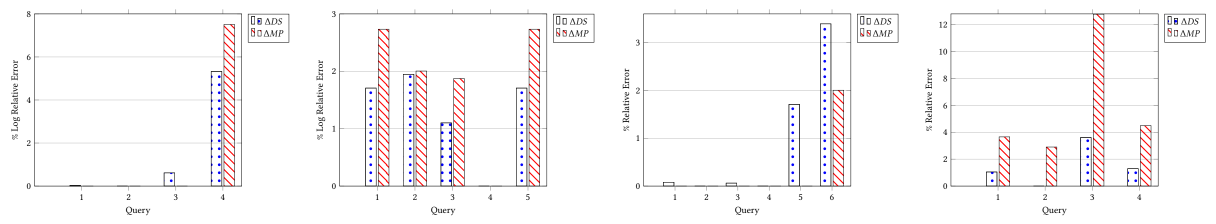
Dataset	Triples (M)	$\lvert\overrightarrow{Gf}\rvert$	$\lvert\overleftarrow{Gf}\rvert$	ST (MB)	% Size	ST (min)	% LT.
LUBM	19.9	11	13	0.008	0.00005	0.50	4.4
Watdiv	109	39,855	1,181	198	0.2	84.2	71.6
Yago	284	25,511	1,216	82	1.32	84.1	9.4
DBLP	207	247	26	0.196	0.0006125	4.7	3.6

5.3 Evaluation of cost model

We evaluated the estimations of our model measuring the relative error in the estimation of data stars and mappings for the query selected as best query. The results of these estimations are shown in Figure 7 (plotted with logarithmic scale for readability). The relative error is greater in queries that do not send back any result. However, as it is seen later, this estimation does not affect the choice of the best execution plan.

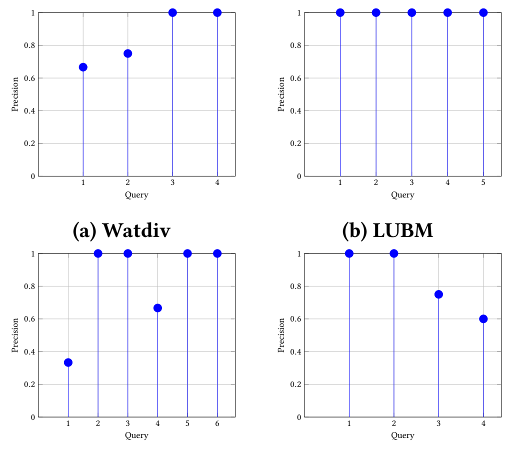
5.4 Optimal $\mathcal{P}$ Algorithm

We defined a precision measure to evaluate the choice of the plan made by the selection algorithm. We sorted the execution plans for each query based on their execution time. The precision measures how far is the best plan proposed by the algorithm compared to the actual best plan in terms of execution time. The



(a) Watdiv **(b) LUBM** **(c) DBLP** **(d) Yago**

Figure 7: Relative Error Estimation



(a) Watdiv **(b) LUBM** **(c) DBLP** **(d) Yago**

Figure 8: Precision of Best $\mathcal{P}$ Algorithm

precision is defined as:

$$Precision(\mathcal{P}) = (\#Plans - Pos(\mathcal{P}))/(\#Plans - 1)$$

where *Pos* is a function returning the plan's rank with respect to the sorted plans in terms of execution time. The results for each dataset are shown in Figure 8. For all datasets, the prediction of the best execution time escapes is either the best possible plan (according to the execution time) or one of the top best.

6 CONCLUSION

In this paper, inspired from the relational model, we first provided logical structures to model the execution plan based on graph exploration techniques. Then, we proposed a novel cost model comparing equivalent logical execution plans based on statistics collected for clusters of triples (that we denoted graph fragments). The cost model estimates the disk and network interactions for a specific logical plan. Furthermore, we studied formally the complexity of the problem related to the choice of the best execution plan and we proposed a branch and bound like algorithm allowing to find the best plan for a specific query. For experimentations, we used synthetic and real datasets. The results showed that cardinality estimations based on our model are very precise even if the collected statistics' size is negligible.

For future work, we plan to explore other optimization strategies such as real time execution plan auto-adaptation, also we plan to use Machine Learning techniques on runtime logs.

REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. 2009. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.* 18, 2 (2009), 385–406.

[2] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB* 10, 13 (2017), 2049–2060.

[3] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. 2002. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web - ISWC First International Semantic Web Conference, Sardinia, Italy, June 9-12.* 54–68.

[4] Lei Gai, Xiaoming Wang, and Tengjiao Wang. 2018. ROSIE: Runtime Optimization of SPARQL Queries over RDF Using Incremental Evaluation. In *11th International Conference, KSEM 2018, Changchun, China, August 17-19.* 117–131.

[5] Jorge Galicia, Amin Mesmoudi, and Ladjel Bellatreche. 2019. RDFPartSuite: Bridging Physical and Logical RDF Partitioning. In *21st International Conference, DaWaK 2019, Linz, Austria, August 26-29.* 136–150.

[6] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. In *Proceedings of the 17th EDBT 2014, Athens, Greece, March 24-28.* 439–450.

[7] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. [n.d.]. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proceedings of the 2014 ACM SIGMOD, year = 2014, location = Snowbird, Utah, USA, pages = 289–300, numpages = 12.*

[8] Yannis E. Ioannidis. 2003. The History of Histograms (abridged). In *Proceedings of 29th VLDB 2003, Berlin, Germany, September 9-12.* 19–30.

[9] Abdallah Khelil, Amin Mesmoudi, Jorge Galicia, and Mohamed Senouci. 2019. Should We Be Afraid of Querying Billions of Triples in a Graph-Based Centralized System?. In *Model and Data Engineering - 9th International Conference, MEDI 2019, Toulouse, France, October 28-31.* 251–266.

[10] Brian McBride. 2002. Jena: A Semantic Web Toolkit. *IEEE Internet Computing* 6, 6 (2002), 55–59.

[11] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th ICDE 2011, April 11-16, Hannover, Germany.* 984–994.

[12] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *PVLDB* 1, 1 (2008), 647–659.

[13] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal* (2010), 91–113.

[14] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *VLDB J.* 25, 2 (2016), 243–268.

[15] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter A. Boncz. 2015. Deriving an Emergent Relational Schema from RDF Data. In *Proceedings of the 24th WWW 2015, Florence, Italy, May 18-22.* 864–874.

[16] Theoni Pitoura and Peter Triantafillou. 2008. Self-Join Size Estimation in Large-scale Distributed Data Systems. In *Proceedings of the 24th ICDE, April 7-12, Cancún, Mexico.* 764–773.

[17] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. (1996), 294–305.

[18] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the World Wide Web Conference on World Wide Web, WWW , Lyon, France, April 23-27.* 1043–1052.

[19] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th WWW 2008, Beijing, China, April 21-25.* 595–604.

[20] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *15th EDBT'12, Berlin, Germany, March 27-30.* 324–335.

[21] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. 2013. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *29th IEEE ICDE, Brisbane, Australia, April 8-12.* 565–576.

[22] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. gStore: a graph-based SPARQL query engine. *VLDB J.* 23, 4 (2014), 565–590.