Heuristics-based SPARQL Query Planning

Fuqi Song, Olivier Corby

Heuristics-based SPARQL Query Planning

Fuqi Song, Olivier Corby

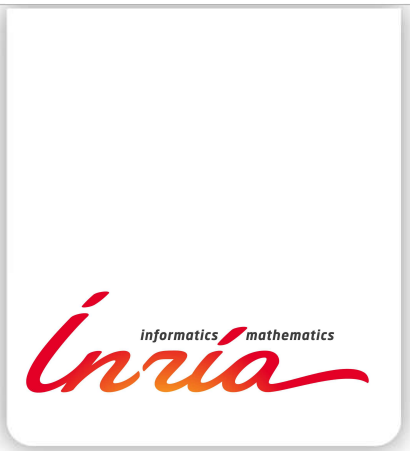
RESEARCH REPORT

N° 8655

December 2014

Project-Teams Wimmics

ISSN 0249-6399 ISRN INRIA/RR--8655--FR+ENG



Heuristics-based SPARQL Query Planning

Fuqi Song *, Olivier Corby †

Équipes-Projets Wimmics

Rapport de recherche n° 8655 — December 2014 — 17 pages

Résumé : La planification de requête joue un rôle essentiel dans l'optimisation de l'exécution des requêtes SPARQL. Ce rapport présente une méthode de planification basée sur des heuristiques pour optimiser l'exécution de requêtes et une implémentation réalisée dans la plateforme Corese. Dans un premier temps, nous proposons une représentation abstraite des énoncés SPARQL en généralisant les représentations habituellement utilisées à d'autres énoncés que les simples triplets. Ensuite, nous étendons les heuristiques utilisées habituellement pour estimer le coût des énoncés. Les méthodes proposées sont évaluées sur le benchmark BSBM. Les résultats montrent que les méthodes proposées optimisent effectivement le temps d'exécution du moteur Corese et montrent également certains avantages comparés à Jena et Sesame utilisés en mode mémoire vive.

Mots-clés : SPARQL, query planning, query optimization, heuristics, Corese

* Inria, I3S, fuqi.song@inria.fr
† Inria, I3S, olivier.corby@inria.fr

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Heuristics-based SPARQL Query Planning

Abstract: SPARQL query planning, as an essential task of query optimizer in SPARQL query engine, plays a significant role in improving query execution performance. Based on Corese query engine, this report presents a heuristic-based approach for performing query planning and optimization. First, this report generalizes SPARQL query statement representation by taking other expressions into account, aiming at overcoming the limitations of only using basic query triple patterns. Second, this report extends the heuristics for estimating the cost of query triple pattern. The proposed query planning methods are implemented within Corese and the system is evaluated using BSBM benchmark. The results suggest that the proposed methods optimized effectively the query execution time by comparing to the original system. In addition, Corese system with the new query planning method also showed certain advantages over Jena and Sesame system in term of query execution time using in-memory storage mode.

Key-words: SPARQL, query planning, query optimization, heuristics, Corese

1 Introduction

SPARQL (SPARQL Protocol and RDF Query Language) [1] is a query language for RDF (Resource Description Framework) [2] proposed by W3C and is recognized as one of the key components in the domain of Semantic Web and Linked Data [1]. The latest recommendation is SPARQL 1.1 Query & Update published in March 2013. As the size of RDF data set increasing, especially the huge amount of RDF data is being published for Semantic Web and Linked Data, efficient SPARQL query engines are expected and demanded for retrieving data rapidly. To this goal query engines usually furnish a query optimizer to undertake the task of optimization aiming at reducing the query execution cost for large data sets. Since SPARQL is declarative, a SPARQL query can be written differently with several orders of expressions. However, the execution time for the different orders that are generated from the same SPARQL statement can vary much. Sometimes, simple re-orderings can reduce the querying time considerably. Therefore Query Planning (QP), which evaluates the possible query plans and finds a best one for the query engine, is regarded as an essential task in query optimizer .

SPARQL query planning involves two main research issues: 1) how to represent SPARQL statements and 2) how to evaluate the cost of SPARQL expressions. Concerning the first issue, the mostly used approach is to construct triple pattern graph from SPARQL statement, such as [2, 3]. These approaches only take basic query triple patterns into account, however other expressions, which play also important roles in performing QP, such as filter and named graph pattern, are not taken into account.

`FILTER` expression can affect the query execution time with its different positions in the query statement to be executed. Moreover, `FILTER` is widely used by users in SPARQL query, Arias et al. [4] did a survey on real-world SPARQL queries executed on DBPedia [3] (5 million queries) and SWDF [4] (2 million queries). The results showed that 49.19% and 47.28% of queries used at least once `FILTER` expression from DBPedia and SWDF respectively. Besides, since RDF 1.1 (published on 25 Feb. 2014), the *data set and named graph* concepts are introduced and several serialization formats supporting named graph, namely, JSON-LD, TriG and N-Quads, are proposed, thus more and more RDF data sources will use data sets. In SPARQL, `GRAPH` expression is used for querying data from specific named graphs. Intuitively, to execute a query from a specific data set first other than from all data sets is helpful in reducing query execution time. Therefore, we think, besides basic triple pattern, the study of these expressions has much significance for performing query planning. Our focus of this report is to extend the SPARQL statement representation by also considering these expressions.

Regarding the second issue in doing query planning: how to evaluate the cost of SPARQL expressions, the mostly used approaches are pre-computed statistics-based [5] and heuristic-based [3]. The first approach calculates certain summary data on RDF source, usually using histogram-based methods, and then utilize the data to evaluate the cost of query plans. Heuristic-based method first defines certain heuristics according to the observation on RDF data sources and then apply these heuristics to estimate the cost. Each approach has some advantages and also some limitations. Statistics-based method is usually more expensive in terms of implementation and resource (time, storage) but maintaining relatively higher accuracy. Heuristic-based approach is easier to implement and costs less, but may be less efficient for some particular data sets.

According to Tsialiamanis et al. [3], heuristic-based method can produce promising results for SPARQL QP due to the particular features of RDF data set, which have certain fixed patterns

1. `http://www.w3.org/TR/sparql11-query/`
2. `http://www.w3.org/TR/rdf11-concepts/`
3. `http://dbpedia.org/`
4. `http://data.semanticweb.org/`

with triples. Also considering the advantages of heuristic-based approach mentioned above, this report will focus on developing the heuristic-based methods by extending basic query triple pattern graph.

Our research work about query planning is carried out based on Corese [5], which is a Semantic Web factory with a SPARQL 1.1 query engine. Corese serves as a RDF triple store using native in memory storage and abstract graph representation [6, 7]. Current Corese system already has certain query optimization considerations, including splitting filters and setting filters just after the query triples patterns that use them. Nevertheless it lacks a systematic declarative query planning mechanism. Thus a new query optimizer component is designed and implemented within Corese aiming at improving the query performance.

Query planning was studied and applied in RDBS for decades, the generic problem for SPARQL query planning is similar. Chaudhuri [8] stated that, to do a query planning we need: 1) generated query plans, 2) cost estimation techniques, and 3) enumeration algorithm. A desirable optimizer is the one, where 1) the generated query plans includes plans that have low cost, 2) the cost estimation technique is accurate, and 3) the enumeration algorithm is efficient. Based on this statement, a 3-steps query planning is designed as illustrated in Figure 1. The first step is to generate the Extended SPARQL triple pattern Graph (ESG), which is different from Chaudhuri's statement that we don't generate all the candidate plans in advance, all the plans are evaluated at the same time while searching ESG for finding the best plan (step 3) using the estimated costs (step 2). Briefly speaking, we use the estimated cost (step 2) to search (step 3) the generated ESG (step 1) in order to find the best plan and use it to rewrite the original query statement. These steps are elaborated in following sections.

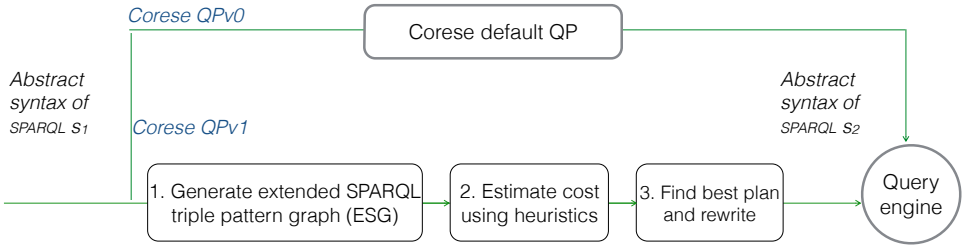


FIGURE 1 – 3-steps query planning

The reminder of this report is organized as follows. Section 2 investigates related work in SPARQL query planning and analyzes the existing issues. Section 3 presents the Extended SPARQL query triple pattern graph (ESG) generated from SPARQL statement, ESG serves as the basis for exploring query planning. Section 4 elaborates the proposed heuristics and cost models for estimating the cost based on ESG. Section 5 describes the algorithms for finding the best query plan using ESG and the estimated cost. Section 6 performs the experiments and discusses the obtained results. Section 7 draws the conclusions of this report and tackles the future work.

2 Related work

The related work is studied from the following aspects: 1) SPARQL query statement representation approaches, and 2) the cost estimation approaches as listed in Table 1.

5. http://wimmics.inria.fr/corese

First, we investigated the methods for representing the SPARQL query statement, basically, most of the authors use basic triple patterns to build graph that is composed of nodes and edges as query planning space. In Liu et al. [9], they differentiated the nodes as *normal vertices* and *triple vertices*, the difference is that the former refers to the variables with constraints in a triple pattern and the latter refers to the whole triple pattern, so forth the *normal edges* and *triple edges*. This approach maintains not only the connections between triple patterns but also the relations within the triple patterns themselves. Tsialiamanis et al. [3] built a graph called *variable graph*, which only considers the variables appearing in triple patterns, and then they assign a weight to these variables by using the number of their occurrence in the SPARQL query. This representation is related directly to their cost estimation model, which aims to find maximum weight independent sets of the variable graph. In this report, we will adopt a method similar to [2, 5], which constructs the graph by using basic triple pattern as nodes and creating edges if there exist shared variables. However this report extended the graph by considering more SPARQL expressions and defining a cost model for each node and edge, these models will be used to formalize the cost estimation approach.

Second, we studied the approaches used to estimate the cost of query plans. Mainly two kinds of approaches are being used in existing work: heuristic-based and statistics-based. Borrowing from the domain of RDBS query optimization, histogram-based [11] methods are widely used for storing summary data, this method maintains relative high accuracy but cost much time and storage space, which can be expensive if no compact summary data structure and efficient data accessing mechanism. The work [2, 5, 10] used this approaches. However given the visible features of RDF data sets, heuristics-based approaches have been studied and developed based on the observations on the data sources. Tsialiamanis et al. [3] proposed several heuristics for query optimization, supported by their experiments results, this approach out-stands many other approaches at time then, heuristic-based method has certain advantages, among which, it is easy to implement and much less resource-consuming in terms of time and storage. In addition, it is less constrained by the environment, for instance, in distributed environment it is difficult to obtain statistics data from endpoint, thus it is not easy to apply statistics-based approaches. Given the promising advantages of heuristics-based approaches, this report focuses on studying this approach. Besides, in order to improve the accuracy on the condition that the stats data are available, this report uses some basic summary data, such as number of distinct resources, predicates and objects, etc, to complement the heuristic-based method.

Table 1 – SPARQL query planning approaches

Author	**SPARQL representation**	**Cost estimation**
Tsialiamanis 2012 [3]	Pattern variable graph	Heuristic-based
Huang 2010 [10]	-	Stats-based
Liu 2010 [9]	SPARQL query graph	Stats-based
Stocker 2008 [2]	Basic pattern graph	Hybrid
Neumann 2008 [5]	Basic pattern graph	Stats-based
Our work	Extended triple pattern graph	Heuristic-based

3 Extended SPARQL query triple pattern Graph (ESG)

Conceptualizing and modeling SPARQL statements is the first step to do query planning. We use Extended SPARQL query triple pattern Graph (ESG), which is defined as ESG = (V, E), where V denotes a set of vertices v and E refers to a set of edges e that are composed of two

vertices from V. Vertex v is defined in Eq. (1).

$$v = (exp, type, cost\ model, cost) \tag{1}$$

where exp refers to the abstract syntax of SPARQL expressions with expression type including basic query triple pattern (T), filter (F), values (VA) and named graph (G). $cost$ refers to the resources needed for executing the expression that the vertex represents while $cost\ model$ is a data structure for estimating the value of $cost$.

If two SPARQL expressions share at least one variable, then one edge e will be created, but no edge will be created between expressions with type F or VA (i.e. F - F or F - VA or VA - VA). An edge e connecting two vertices is defined as

$$e = (v_1, v_2, type, vars, cost\ model, cost) \tag{2}$$

where v_1 and v_2 are from set V, $type$ refer to the edge type (directed or bi-directional) and $vars$ denotes the shared variables between v_1 and v_2, $cost$ refers to the resource needed for executing v_1 and v_2 in order, $cost\ model$ is a data structure for estimating the value of cost. Cost of vertex and edge is denoted by a real number between 0 and 1 where bigger values indicate higher costs.

Figure 2 illustrates an example using an ESG to represent a SPARQL query. Each expression is denoted by an ESG vertex. ESG does not contain sub graphs, which means that we perform query planning on partial statement where the types of the expressions are within T, F, VA, or G. One SPARQL query can generate several ESGs according to different levels and type of expressions. QP is performed on each single ESG. The example in Figure 2 only contains one ESG, thus the QP will be performed only once. For instance, the example shown in Figure 3 generates three ESGs. The first one contains one basic triple pattern and two named graph patterns (without investigating the inside). For querying named graphs ?all and ex : $researcher$, each of them will generate one ESG with the expressions it contains. The other examples include OPTIONAL, UNION and sub queries, for these cases, each of them will generate one ESG.

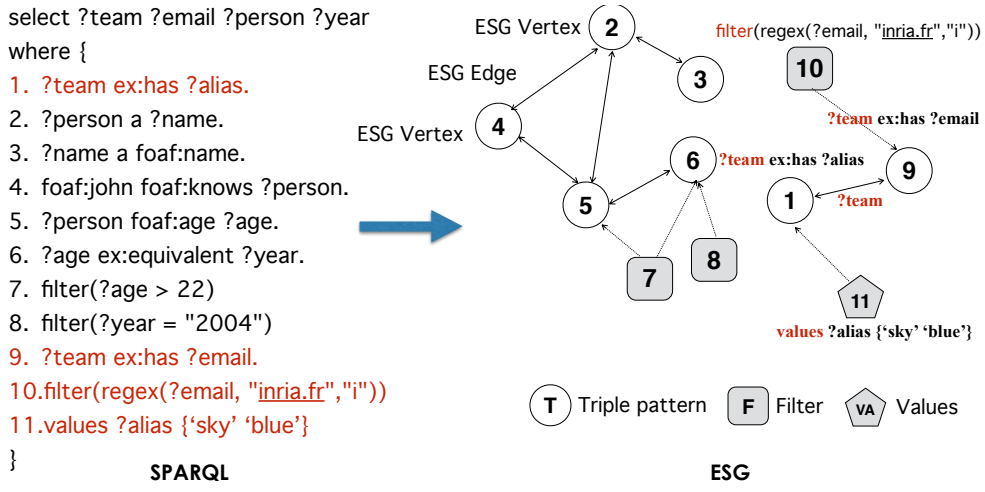


FIGURE 2 – Use ESG to represent SPARQL statement

Based on ESG, relevant graph search algorithms [12] can be used to explore the query plan. For instance, star-mode and chain-mode [9] are two query modes widely used, with ESG the two structures can be detected as well as other special user interested structures.

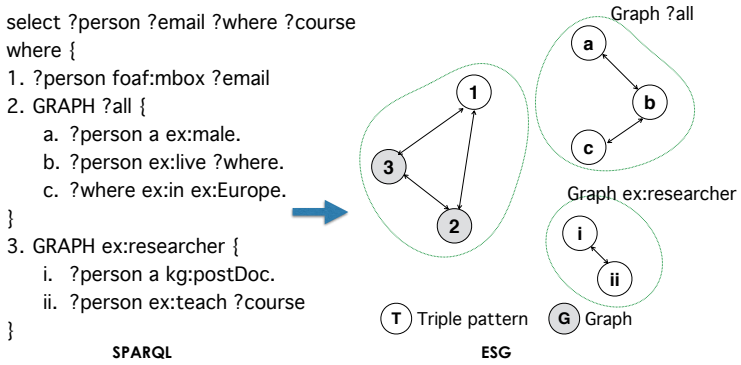


FIGURE 3 – Example of SPARQL containing multiple ESGs

4 Cost estimation

The cost mainly refers to time consumed and storage space needed for executing one or several SPARQL expressions in the query plan. Both of them can be evaluated by the number of triples queried from the RDF data set, namely, if the execution of the expressions returns large set of results, then the cost is high in terms of query execution time and space needed. From this perspective, we introduce the notion *selectivity* to measure the cost. The more a triple pattern is selective, the less is the number of returned results.

Cost for an ESG vertex refers to resources needed for executing the single expression it represented, while cost for an ESG edge refers to resources needed for executing the expressions of v_1 and v_2 in order. The following sections present the estimation for ESG vertex and edge respectively. For both of them, first we present the heuristics and the cost model formalizing these heuristics. Second we present how to estimate the cost using the cost model.

4.1 ESG vertex cost estimation

ESG vertex cost estimation only estimates the ones with type G (named graph pattern) and T (query triple pattern). Because in our approach of QP, for the ones with type F (filter) and VA (values), only their positions in query statement and connections to other vertices with type G and T matter, thus we do not estimate the cost for those vertices. This part will be elaborated in the searching algorithm of Section 5.

4.1.1 Heuristics

Tsialiamanis et al. [3] and Stocker et al. [2] proposed some heuristics for evaluating the cost of a triple pattern. First we generalize H1 defined by [3] to H1' on condition that some basic summary data is available, then we propose heuristics (H2 - H5) considering filters and named graphs. These heuristics try to estimate the cost from a qualitative point of view. In order to compare then from a quantitative perspective we propose a cost model in next section to formalize these heuristics. Heuristics H1 to H3 are for estimating costs among basic query triple patterns (T), while H4 is for estimating expressions between T and G, and H5 is for estimating the costs between named graph patterns G.

H1: The cost for executing query triple pattern is ordered as: $c(s,p,o) \leq c(s,?,o) \leq c(?,p,o) \leq c(s,p,?) \leq c(?,?,o) \leq c(s,?,?) \leq c(?,p,?) \leq c(?,?,?)$, where ? denotes a variable while s, p and

o denote a value.

H1 is defined based on the hypothesis that the number of distinct predicate is less than subjects, and the number of distinct subjects is less than objects. To generalize and adapt diverse cases, we extend it to H1' on condition that the number of distinct subject, predicate and object can be obtained.

H1': Given the distinct number of subject, predicate and object: N_s, N_p and N_o. We use α, β and γ to denote s, p and o in ascending order of N_s, N_p and N_o. To simplify the notation, we use $c'(\alpha)$ (likewise $c'(\beta)$ and $c'(\gamma)$) to refer to $c(s,?,?)$ or $c(?,p,?)$ or $c(?,?,o)$ depending on the value that α represents, for instance if α represent p then $c'(\alpha) = c(?,p,?)$, similarly, $c'(\beta,\gamma)$ (likewise $c'(\alpha,\beta)$ and $c'(\alpha,\gamma)$) refers to $c(s,?,o)$ or $c(?,p,o)$ or $c(s,p,?)$ depending on the value that β and γ represent, for instance if β and γ represent s and o then $c'(\beta,\gamma) = c(s,?,o)$. The list of cost is $c(s,p,o) \leq c'(\beta,\gamma) \leq c'(\alpha,\gamma) \leq c'(\alpha,\beta) \leq c'(\gamma) \leq c'(\beta) \leq c'(\alpha) \leq c(?,?,?)$ ordered by increasing cost. H1 is a particular case of H1' where $N_p \leq N_s \leq N_o$.

H2: The triple pattern that is related to more filters has higher selectivity and costs less, F_F denotes the number of filters related to this triple pattern, $F_F \in \{0,1,2,...,n\}$.

H3: The triple pattern that has more variables appearing in filters has higher selectivity and costs less, F_V denotes the number of variables in a triple pattern appearing in filters, $F_V \in \{0,1,2,3\}$.

H2 and H3 are defined based on the observation that a filter usually can reduce intermediate results even if the triple pattern only has one filter. Moreover, the more variables appear in filters, the more results can be reduced, because filters are applied on different variables and hence may reduce more the number of results.

H4: Basic query triple pattern has higher selectivity and costs less than a named graph pattern. But when the basic query triple pattern conforms to $t(?,?,?)$, we will consider it costs more than a named graph pattern if the named graph pattern does not only contain query triple patterns like $t(?,?,?)$.

H5: A query executed with a specific named graph has higher selectivity and costs less, for instance, `graph ?g { ... }` costs more than `graph foaf:bob { ... }` in a SPARQL query.

4.1.2 Cost model

In order to formalize these heuristics, the cost estimation model for ESG vertex v is formalized as follows:

$$v.model = (S, P, O, G, F_V, F_F) \tag{3}$$

The ranges of value are ($s|?$, $p|?$, $o|?$, $g|?$, $[0,3]$, $\mathbb{N}$), where s, p and o denote ground terms in triple pattern (URI or Literal), ? denotes a variable (or a blank node) and the numbers denotes the possible value for F_F and F_V. We detect bound variables which can be considered as values. For instance, if we have filter $?age = 2$, then variable $?age$ is considered as a value and the pattern for ($?person\ foaf{:}age\ ?age$) is set to (?, p, o).

4.1.3 Cost estimation algorithm

Algorithm 1 presents the process to estimate the cost of ESG vertex using cost model. The general idea to assign the cost is first to sort the vertices using the models and then assign a value to each vertex based on its position in the sorted list. Lines 2 – 4 are for re-generating the basic triple patters of H1 if the stats are available. Lines 5 – 8 compute the number of F_V and F_F. Lines 11 – 29 compare the cost of two vertices. Line 9 sorts the vertex list in V and line 10 assigns each vertex a value of cost.

Data: $ESG = (V, E), V = \{v\}, v = \{exp, type, model, cost\}, v.model = \{S, P, O, G, F_V, F_F\}$, default basic triple pattern orders $Patterns$

initialization;
if *stats data is available* **then**
  Re-generate the list of basic triple patterns order ($H1'$) and assign it to $Patterns$;
end
for v *in* V **do**
  get the number of linked filters ff and assign $v.model.F_F = ff$;
  get the number of variables appearing in filters fv and assign $v.model.F_V = fv$;
end
List *sortedVertices* = `sort`(V, $Patterns$, `Comparator`);
for *Vertex* v_i *in sortedVertices* **do**
  $v_i.cost = i/(len - 1)$;
end
Function `Comparator`(v_1, v_2)
  if $v_1.type = G$ *and* $v_2.type = G$ **then**
    return (v_1.model[G] > v_2.model[G]) ? 1 : -1 `// (H5)`;
  else if $v_1.type = T$ *and* $v_2.type = G$ **then**
    return (v_1.pattern = (?, ?, ?)) ? 1 : -1 `// (H4)`;
  else if $v_1.type = G$ *and* $v_2.type = T$ **then**
    return (v_2.pattern = (?, ?, ?)) ? -1 : 1 `// (H4)`;
  else
    Get the index i_1, i_2 of v_1, v_2 in $Patterns$;
    if $i_1 \neq i_2$ **then**
      return $i_1 > i_2$? 1 : -1 `// (H1 or H1')`;
    else if $v_1.ff \neq v_2.ff$ **then**
      return ($v_1.ff < v_2.ff$) ? 1 : -1 `// (H2)`;
    else if $v_1.fv \neq v_2.fv$ **then**
      return ($v_1.fv < v_2.fv$) ? 1 : -1 `// (H3)`;
    else
      return 0;
    end
  end

Algorithm 1: ESG vertex cost estimation

4.2 ESG edge cost estimation

Vertices represent SPARQL expressions (e.g. triple patterns) and edges link vertices that share common variable(s). Cost estimation of ESG edge concerns two vertices related by an oriented edge. We reuse one heuristic H6 proposed by [3] and propose one new heuristic H7.

H6: The position of joint variable of two vertices in one edge affect the selectivity of the join operation, the cost is ordered as $p \bowtie o < s \bowtie p < s \bowtie o < o \bowtie o < s \bowtie s < p \bowtie p$, where s, p, o refer to the position of the joint variable appearing in the two vertices of one edge. The cost of an edge increases accordingly to the order listed above.

H7: Edges whose vertices share several variables are more selective than those sharing only one variable. For instance, (?*person* ?*play ex*: *football*) joined with (?*person* ?*play* ?*game*) (two shared variables ?*person* and ?*play*) is regarded costing less than with (?*person foaf*: *name* ?*name*)

(one shared variable ?*person*).

The two heuristics are formulated as:

$$e.model = (J_{type},\ N_{share}) \tag{4}$$

where J_{type} indicates the position in the list of joined basic patterns listed in H6, the value of J_{type} is assigned from 6 to 1 since we use 1 / $J_{type} * N_{share}$ to denote the cost, therefore the pattern in the front of the list will lead to a smaller value of cost. In the case that one of the vertices is with type G, J_{type} is assigned to the average value, namely, 3.5. N_{share} is the number of shared variables between two vertices. The cost of edge e is defined in Eq. (5), when no variables shared between two vertices and the other type of vertices, the cost will be assigned to a huge number $cost_{max}$.

$$e.cost = \begin{cases} \frac{1}{N_{share} * J_{type}} & if \quad v_1.type = T \wedge v_2.type = T \\ \frac{1}{3.5 * N_{share}} & if \quad v_1.type = G \vee v_2.type = G \\ cost_{max} & if \quad other\ types \vee no\ shared\ variables \end{cases} \tag{5}$$

5 ESG search and find best query plan

Aiming at finding the best query plan, we search ESG using the estimated costs on vertices and edges from section 4. As mentioned in previous sections, we do not generate all candidate query plans in advance because a trade-off is necessary when doing optimization, since the planning time needs to be counted as part of the execution time. Thus, this report adopts a greedy algorithm starting at the vertex with smaller cost and searching linked vertices recursively. The purpose is to reduce the size of the data set to be queried *as much as possible, as soon as possible*. The algorithm balances between optimization time and accuracy of query planning in order not to spend too much resources on the task for optimizing the query. Hence, the algorithm may not find the most optimal solution in some cases due to the constraints of greedy algorithm.

Algorithm 2 illustrates the ESG searching algorithm. The general idea is first to find the vertex with lowest cost and use it as the starting point. The next vertex is selected from a map called *nextNodesPool*, all the vertices linked to the already visited vertices are added to this pool as candidates for next steps. The next selected vertex is the one with lowest cost in the pool. Each time a vertex is found, it is added to the *visited* list and removed from the *notVisited* list.

Before adding the chosen vertex to the *visited* list, we check whether there are any vertices with type VA related to this node. If there exist some, we put them into the list *visited* before adding the chosen vertex. The intuition is to add `VALUES` clause just before where they will be used. On the contrary, `FILTER` clause are added just after where the nodes related to the filter have been visited and added to the visited list. The final result is the *visited* list and it will be used as query plan for execution.

```
   Data: ESG = (V, E), V = {v}, E = {e}
   Data: v = {exp, type, model, cost}, e = {v1, v2, type, model, cost}
 1 Initialize empty list visited, notVisited; vertex first;
 2 Put all the nodes with type T and G to list notVisited;
 3 while notVisited is not empty do
 4 |   Initialize new Map (vertex, cost) nextNodesPool;
 5 |   Find the vertex first with lowest cost from list notVisited;
 6 |   Route(first, nextNodesPool);
 7 end
 8 Function Route(previous, nextNodesPool)
 9 |   if previous = null then
10 |   |   return ;
11 |   end
12 |   Vertex next;
13 |   while (next = FindNext(nextNodesPool)) != null do
14 |   |   Route(next, nextNodesPool);
15 |   end
16 Function FindNext(nextNodesPool)
17 |   if nextNodesPool is empty then
18 |   |   return null;
19 |   else
20 |   |   Find the vertex with the lowest cost from nextNodesPool and assign it to
   |   |   next; Queue(next, nextNodesPool);
21 |   |   return next;
22 |   end
23 Function Queue(vertex, nextNodesPool)
24 |   For all vertices that have not been visited with type VA linked to vertex, add
   |   them to list visited;
25 |   visited.add(vertex);
26 |   notVisited.remove(vertex);
27 |   For all vertices with type F whose linked vertices all have been visited, add
   |   them to list visited;
28 |   // fill the next nodes pool
29 |   Get the linked edges lEdges of vertex;
30 |   for edge e in lEdges do
31 |   |   Get the other vertex v2 of e;
32 |   |   costNew = e.cost * v2.cost;
33 |   |   if v2 is not visited then
34 |   |   |   nextNodesPool.put(v2, costNew);
35 |   |   end
36 |   end
```

Algorithm 2: ESG searching for finding best plan

6 Evaluation

The query planning method described in this report is implemented in Java within Corese Semantic Web Factory [6]. Section 6.1 describes the data set and test cases used. In Section 6.2, we present the results obtained and, based on the results, the discussions and analysis are made.

6.1 Data set and test cases

The experiment is performed on Berlin SPARQL Benchmark (BSBM) V3.1 [7], BSBM defines a suite of test cases around an e-commerce use case. It provides relevant tools for generating data sets, sending queries to query engine and generating experimental results. The data generated and used for our experiments are listed in Table 2, including the size of data and the number of triples. The number of triples ranges from 40 thousands to 7 millions, while the size of file in Turtle format ranges from 4 MB to 647 MB.

TABLE 2 – BSBM data sets

Name	**File size(.ttl, MB)**	**Number of triples**
40k	3.8	40 377
115k	10.8	115 987
191k	17.7	191 496
374k	34.7	374 911
725k	66.6	725 305
1M	98.4	1 075 637
1.8M	166	1 809 874
2.5M	230	2 511 942
3.5M	325	3 564 773
5.3M	486	5 319 388
7M	647	7 073 571

Corese QPv_0 uses the default Corese query planning settings. QPv_1 uses the heuristics-based query optimizing method described in this report. For the purpose of comparison, the tests are also performed on two other SPARQL query engines: *Jena Fuseki* [8] and *Sesame* [9], both of them are tested using in-memory mode in order to compare them with Corese on the same basis.The machine used to perform the tests is a Mac, OS X 10.9.4, CUP 2.6 GHz Intel Core i5 with 8 GB RAM and 500 GB hard disk. The criteria used to evaluate and to compare the results is the query execution time for returning correct results. For all tests, the maximum Java heap size is set to 8G (-Xmx8G).

In this report, we tested BSBM *Explore* use case and *Business Intelligence* use case. A query mix is composed of several distinct queries in which each of them appeared once or several times. A query mix simulates one scenario of user's activity. All the tests are performed using BSBM tools.

- BSBM *Explore* uses case is for measuring the performance of triple stores that expose SPARQL endpoints, it contains 11 distinct queries (Q01-Q12, Q6 is excluded by BSBM in V3.1). A query mix is composed of 25 queries from Q01 to Q12 and simulates the business process of a consumer looking for a specific product. The queries in this use case

6. http://wimmics.inria.fr/corese
7. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
8. http://jena.apache.org/documentation/serving_data/
9. http://rdf4j.org/

are relatively simple (rather flat, do not contain nested query) and take little time, so this use case is used for evaluating the described QP approach by comparing to the original system QPv_0. The purpose is to see the effects of the proposed approach since except for the QP approach, the other aspects are identical, such as the data storage and indexing mechanism, query processing, etc.

– BSBM *Business Intelligence* use case simulates different stakeholders asking analytic questions against the data set. The query mix consists of 8 distinct queries (Qb1 - Qb8), each query mix contains 15 queries. The queries in this use case are relatively complex and take longer time. This use case is used for comparing *Corese* with the other systems: *Jena* and *Sesame* to see the performance difference.

6.2 Results and discussion

The QP strategy of QPv_1 in Corese is that first we try to apply the QPv_1, but if it is not applicable, then we will turn to QPv_0. Thus for a SPARQL query containing multiple ESGs, two QP strategies might be applied. For the *Explore* use case, we focus on analyzing the queries that have been rewritten by QPv_1 to see whether the proposed method is useful. For the *Business Intelligence* use case, we emphasize on comparing Corese system (taking as a whole) with the other systems.

6.2.1 BSBM Explore use case

For this use case, each query mix (containing 25 queries) is executed 120 times including 20 times of warm-ups that are not included in the computation of result. The results include the average execution time for each single query and for the whole query mix. The time for this test is presented in milliseconds. This use case contains 11 distinct queries, and only four of them are rewritten differently from the original statement by QPv_1 as shown in Figure 4, for the rest, whose execution time are approximately equal, are not listed.

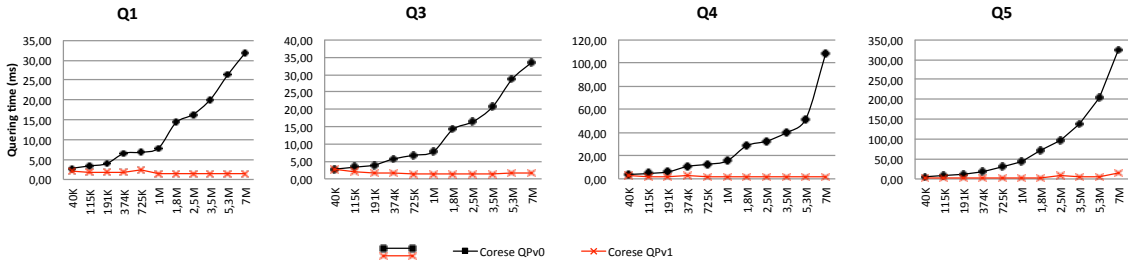


FIGURE 4 – Optimized queries Q1, Q3, Q4 and Q5

The analysis of results between *Corese* QPv_0 and QPv_1 are made from two aspects: optimized queries and whole query mix as illustrated in Figure 5 (a) - (d). The evaluation criterion are the query execution time and percentage [10] of optimized execution time. Figure 5 (a) grouped the queries that have been optimized, we can see from the figure that the execution time (sum of all query execution time) of QPv_0 grows linearly while for QPv_1 the time remains relatively stable and low. Figure 5 (b) presented the percentage of optimized execution time considering these optimized queries. The percentage is 29% for data set 40K triples and increasing to 96%

10. *Calcution* : $p = (T_{QPv0} - T_{QPv1})/T_{QPv0}$, where T_{QPv0} and T_{QPv1} refer to execution time of mix query for *QPv0* and *QPv1*

for data set 7M triples, the average optimization percentage is 80.37%. This result suggest that the optimization is effective if the query can be rewritten, particularly for larger data sets.

Figure 5 (c) presented the execution time of the query mix. Figure 5 (d) illustrated the percentage of optimized execution time. The figure shows that the optimization becomes more significant as the data size increases from 40K triples (optimized 1%) to 7M triples (optimized 89%), in other words it is more effective for larger data set which is also the purpose of optimization. Averagely, QPv_1 optimized 59.68% query execution time of Corese query engine, namely it is 59.68% faster than the original system according to this test.

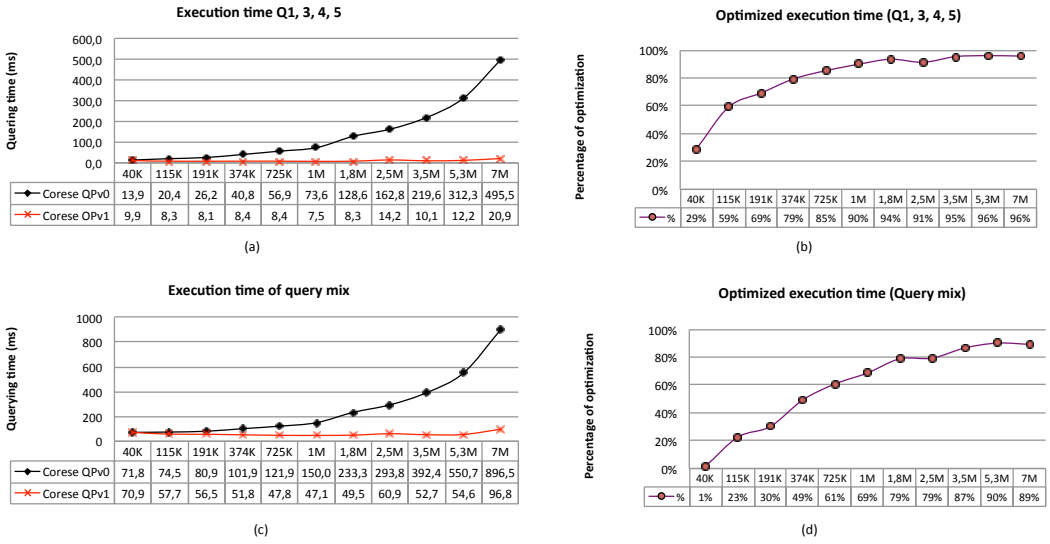


FIGURE 5 – Comparison between *Corese* QPv_0 and QPv_1

6.2.2 BSBM Business Intelligence use case

For this test case, due to its long execution time, each query mix is executed 60 times including 10 times warm-ups. The time unit is second for this test. The timeout limit for single query is set to 600 seconds. The average execution time of each single query is listed in Figure 6. For query Qb8, *Jena* times out (exceeded 600 seconds) on data set 5.3M triples and 7M triples. For query Qb4, *Corese* query execution runs out of memory on data set 7M triples. Therefore, the results on these points are not provided.

First we have an overall analysis on the results, Figure 6 suggested that,

- *Corese* uses less time than *Jena* and *Sesame* for queries Qb1, Qb2 Qb3 and Qb7, for queries Qb4, Qb5 and Qb8, *Corese* uses less time than *Jena* and more time than *Sesame*, for Qb6 Corese costs more than both ;
- Except for query Qb6, *Jena* costs more time for executing the queries, especially for Qb8 it takes about 3.5 minutes and 10 minutes on data set 1.8M and 3.5M respectively ;
- For queries Qb1, Qb2, Qb3 and Qb7, *Sesame* uses less time than *Jena* and more time than *Corese*. For the rest queries, it costs less time than both ;
- For all queries, the execution time increase linearly as the size of data set increasing for all three systems. Generally speaking, *Corese* and *Sesame* have similar performance and have certain advantages over *Jena* in term of query execution time.

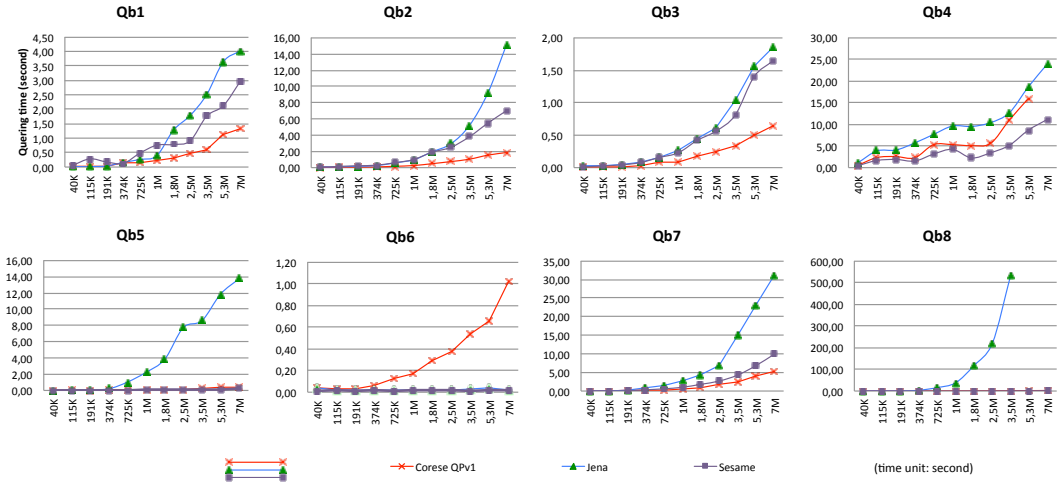
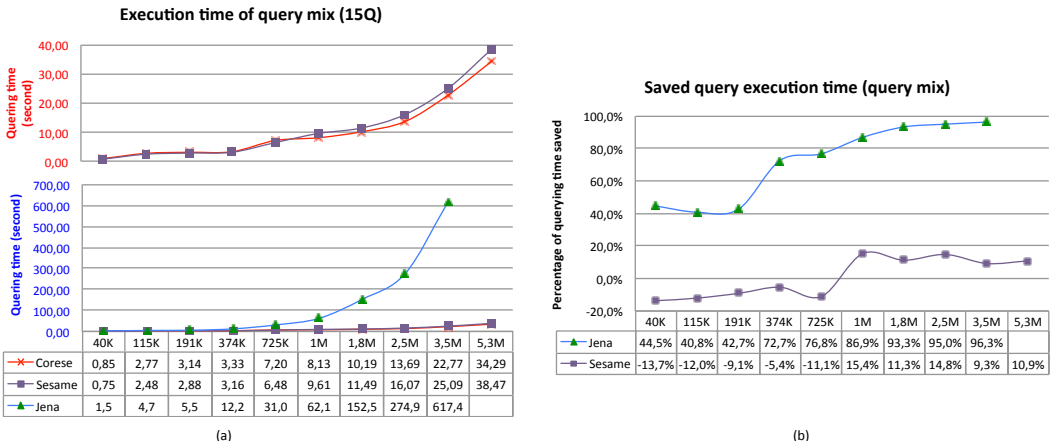


FIGURE 6 – Query execution time of Qb1 - Qb8 for *Corese*, *Jena* and *Sesame*

Figure 7 (a) - (b) presents the test results of query mix. Figure 7 (a) presents the average execution time for the query mix containing 15 queries and (b) illustrates the percentage of saved query execution time comparing to *Jena* and *Sesame*. Since the time between *Corese* and *Sesame* are very similar thus we presented separately on the top of (a) and the bottom one presents all of the three. From the figures in (a), we can see that the execution time for *Corese* ranges from 0.85 seconds (40K) to 34.29 seconds (5.3M), while for *Jena* it takes from 1.5 seconds (40k) to 617.4 seconds (3.5M).



	40K	115K	191K	374K	725K	1M	1,8M	2,5M	3,5M	5,3M
Corese	0,85	2,77	3,14	3,33	7,20	8,13	10,19	13,69	22,77	34,29
Sesame	0,75	2,48	2,88	3,16	6,48	9,61	11,49	16,07	25,09	38,47
Jena	1,5	4,7	5,5	12,2	31,0	62,1	152,5	274,9	617,4	

	40K	115K	191K	374K	725K	1M	1,8M	2,5M	3,5M	5,3M
Jena	44,5%	40,8%	42,7%	72,7%	76,8%	86,9%	93,3%	95,0%	96,3%	
Sesame	-13,7%	-12,0%	-9,1%	-5,4%	-11,1%	15,4%	11,3%	14,8%	9,3%	10,9%

FIGURE 7 – Comparisons among *Corese*, *Jena* and *Sesame*

Figure 7 (b) illustrated the percentage of saved query execution time. Comparing to *Jena*, for data sets from 40K to 191K, the percentages remains approximately 40%, from 374K the figure increases linearly to 96.3%, which suggests that *Corese* has better performance than it for bigger data set. Averagely, Corese saved 72.12% of time. Comparing to *Sesame*, for data set from 40K to 725K, the percentage is negative and around -10% meaning *Corese* uses approximately 10% more time than *Sesame* on these data sets. Between data sets 1M to 5.3M, *Corese* saved around

12% execution time. In other words, *Corese* has certain advantage over *Sesame* for larger data sets.

7 Conclusions and future work

This report presented a 3-steps heuristic-based SPARQL query planning approach including generating the extended SPARQL query pattern graph ESG, estimating costs using heuristics and finding best query plan. From our point of view, the work of this report has the following two main contributions: 1) extend SPARQL query statement representation by taking more expressions into account for doing QP and 2) extend in a systematic way the heuristics for evaluating the cost of query triple pattern. Supported by the experiments performed in Section 6, the results suggested that the proposed methods improved effectively the query performance of original Corese query engine, while comparing to Jena and Sesame, the Corese system with the proposed QP approach also showed certain advantages in term of query execution time in particularly for larger data sets.

The work will be continued from following perspectives. First, the heuristics for estimating the costs will be enriched in order to improve the accuracy of the QP approach for diverse data sources, especially the heuristics related to evaluating the cost of named graph patterns can be further studied. Second, as Linked Data (LD) being widely applied and used, one of the big trends is that more distributed applications are getting involved and many systems are deployed in distributed environment, thus adapting query planning to distributed query processing is necessary. We would like to work towards this direction in the future work.

Acknowledgment

We would like to thank Inria for the support of project `ADT KGRAM`.

Références

[1] Berners-Lee, T. (2006-07-27, 2009-06-18). Linked Data - Design Issues. *W3C*. Retrieved 2014-11-18, from `http://www.w3.org/DesignIssues/LinkedData.html`.

[2] Stocker, M., et al. (2008). SPARQL basic graph pattern optimization using selectivity estimation. *17th international conference on World Wide Web (WWW)*, Beijing, China, ACM.

[3] Tsialiamanis, P., et al. (2012). Heuristics-based query optimisation for SPARQL. *15th International Conference on Extending Database Technology (ICDT)*, New York, ACM.

[4] Arias, M., et al. (2011).An empirical study of real-world SPARQL queries. *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) in the 20th International World Wide Web Conference (WWW)*, Hyderabad, India.

[5] Neumann, T. and G. Weikum (2008). RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1(1): 647-659.

[6] Corby, O. and C. Faron-Zucker (2010). The KGRAM Abstract Machine for Knowledge Graph Querying. *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, Toronto.

[7] Corby, O., et al. (2012). *KGRAM Versatile Inference and Query Engine for the Web of Linked Data*. Proceedings of the 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01, IEEE Computer Society.

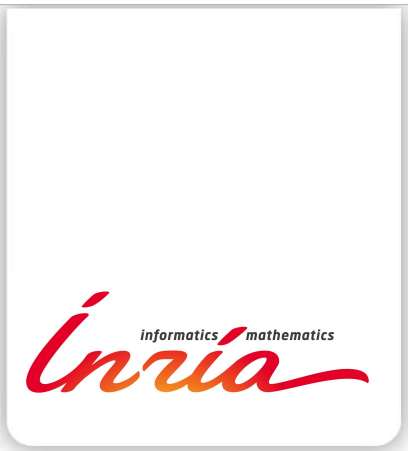
[8] Chaudhuri, S. (1998). An overview of query optimization in relational systems. *the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, Seattle, Washington, USA, ACM.

[9] Liu, C., et al. (2010). Towards efficient SPARQL query processing on RDF data. *Tsinghua Science & Technology* 15(6): 613-622.

[10] Huang, H. and C. Liu (2010). Selectivity estimation for SPARQL graph pattern. *19th international conference on World Wide Web (WWW2010)*, Raleigh, North Carolina, USA, ACM.

[11] Poosala, V., et al. (1996). Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD Record* 25(2): 294-305.

[12] Harary, F. (1994). Graph Theory, Perseus Books.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399