





Cost- and Robustness-Based Query Optimization for Linked Data Fragments

Lars Heling^(✉)  and Maribel Acosta 

Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{heling,acosta}@kit.edu

Abstract. Client-side SPARQL query processing enables evaluating queries over RDF datasets published on the Web without producing high loads on the data providers' servers. Triple Pattern Fragment (TPF) servers provide means to publish highly available RDF data on the Web and clients to evaluate SPARQL queries over them have been proposed. For clients to devise efficient query plans that minimize both the number of requests submitted to the server as well as the overall execution time, it is key to accurately estimate join cardinalities to appropriately place physical join operators. However, collecting accurate and fine-grained statistics from remote sources is a challenging task, and clients typically rely on the metadata provided by the TPF server. Addressing this shortcoming, we propose CROP, a cost- and robust-based query optimizer to devise efficient plans combining both cost and robustness of query plans. The idea of robustness is determining the impact of join cardinality estimation errors on the cost of a query plan and to avoid plans where this impact is very high. In our experimental study, we show that our concept of robustness complements the cost model and improves the efficiency of query plans. Additionally, we show that our approach outperforms existing TPF clients in terms of overall runtime and number of requests.

1 Introduction

Different means to publish RDF and Linked Data on the web have been proposed ranging from data dumps with no support to directly query the data to SPARQL endpoints which allow for executing complex SPARQL queries over the data [17]. Motivated by the low availability and high server-side cost of SPARQL endpoints, Triple Pattern Fragments (TPFs) have been proposed as a lightweight triple pattern-based query interface [17]. The goal is to increase the availability of data by reducing server-side costs and shifting the cost for evaluating large queries to the client. Given a triple pattern, the TPF server returns all matching triples split into pages as well as additional metadata on the estimated number of total matching triples and the page size. Evaluating SPARQL queries over datasets published via TPF server requires a specific client with query processing capabilities. A key challenge of such clients is devising efficient query plans able to minimize both the overall query execution time as well as the number of requests submitted to the TPF server. Different clients implement

heuristics based on the provided metadata to devise efficient query plans over TPF servers [1, 16, 17]. However, a major drawback of the heuristics implemented by those clients is the fact that they fail to adapt to different classes of queries which can lead to long runtimes and produce large numbers of requests. This can be attributed to the following reasons. First, they follow a greedy planning strategy and do not explore and compare alternative query plans. Second, they only rely on basic cardinality estimation functions to estimate the cost of query plans and to place physical join operators. To overcome these limitations, a more flexible way of query planning in TPF clients can be realized by implementing both a cost model to estimate the cost of query plans and a query planner that explores alternative plans.

To this end, we propose a new cost model incorporating both the cost at the client (execution time) as well as the cost induced at the server (number of requests) to devise efficient query plans. Our cost model relies on a basic estimation function to estimate the number of intermediate results and join cardinalities of sub-queries based on the TPF metadata. Due to the limited statistics, we additionally propose the concept of robustness for query plans to avoid query plans which are very susceptible to errors in the estimations. Therefore, the robustness of a query plan is determined by the ratio of its best-case cost and its average-case cost. A higher ratio indicates that the best-case cost deviates less from the average case cost and the plan is considered more robust. Finally, we present a query plan optimizer that combines both the cost model and the concept of robustness to select the most appropriate query plan which ideally minimizes the overall evaluation runtime and the number of requests submitted to the TPF server. In summary, our contributions are

- a cost model for executing query plans over TPF servers,
- the concept of robustness for SPARQL query plans,
- a query plan optimizer using iterative dynamic programming to explore alternative query plans with the goal to obtain both cheap and robust query plans, and
- an implementation of the approach evaluated in an extensive experimental study.

The remainder of this paper is structured as follows. First, we present a motivating example in Sect. 2. In Sect. 3, we present our approach and evaluate it in Sect. 4 by analyzing the results of our experimental study. Next, we discuss related work in Sect. 5. Finally, we summarize our work in Sect. 6 and point to future work.

2 Motivating Example

Consider the query from Listing 1.1 to *obtain persons with “Stanford University” as their alma mater, the title of their thesis, and their doctoral advisor* using the TPF for the English version of DBpedia¹ with a page size of 100. The estimated

¹ <http://fragments.dbpedia.org/2014/en>.

triples per triple pattern provided as metadata from the TPF server are indicated in Listing 1.1.

Listing 1.1. Query to get persons with “Stanford University” as their alma mater, the title of their thesis and their doctoral advisor. Prefixes are used as in <http://prefix.cc/>.

```

0 SELECT * WHERE {
1 ?u rdfs:label "Stanford University"@en .      # Count: 2
2 ?s dbo:almaMater ?u .                        # Count: 86088
3 ?s dbp:thesisTitle ?t .                     # Count: 1187
4 ?s dbo:doctoralAdvisor ?d . }               # Count: 4885

```

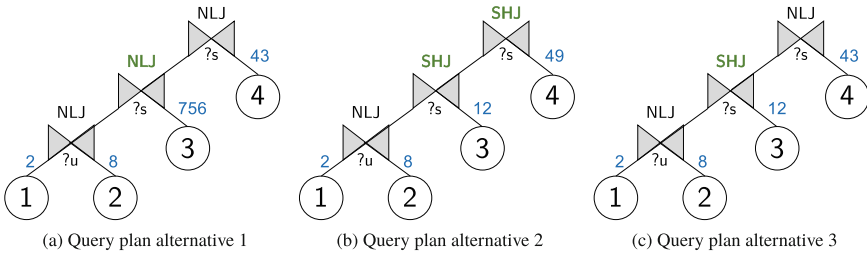


Fig. 1. Three alternative query plans for the SPARQL query from Listing 1.1. Indicated on the edges are the number of requests to be performed according to the corresponding join operators: nested loop join (NLJ) and symmetric hash join (SHJ).

We now want to investigate the query plans produced by the recent TPF clients *Comunica* and *nLDE*. When executing the query using *comunica-sparql*², the client produces 813 requests to obtain the 29 results of the query. The heuristics first sorts the triple patterns according to the number of triples they match. They are then placed in ascending order in the query plan with Nested Loop Joins (NLJs) as the physical operators [16]. The corresponding physical plan is shown in Fig. 1a, where the number of requests is indicated on the edges. First, 4 requests are performed to obtain the statistics on the triple patterns, and thereafter, the plan is executed with 809 requests. Executing the query using *nLDE*³ results in a total of 75 requests. First, 4 requests are performed to receive the triple patterns’ statistics. Next, by placing two Symmetric Hash Join (SHJ) instead of NLJs only, the number of requests for executing the plan is reduced to a total of 71. In the heuristic of the *nLDE* query planner, the number of results produced by a join, i.e., the join cardinality estimations, are computed as the sum of the incoming cardinalities of the join. Based on these estimations the planner places either an NLJ or an SHJ [1]. The corresponding query plan is shown in Fig. 1b. Taking a closer look at the query, we observe that neither

² <https://github.com/comunica/comunica/tree/master/packages/actor-init-sparql>.

³ <https://github.com/maribelacosta/nlde>.

comunica-sparql nor nLDE find the query plan which minimizes the number of requests to be performed. The optimal plan is shown in Fig. 1c and it requires a total of 69 requests only. This is achieved by placing an NLJ at first and third join operator, and an SHJ at the second join operator. This example emphasizes the challenge for heuristics to devise efficient query plans based on only the count statistic provided by the TPF servers. In this example, the subject-object join of triple patterns 1 and 2 yields 756 results. This can be difficult to estimate relying on the TPF metadata alone. On the one hand, an optimistic heuristic assuming low join cardinalities (for example the minimum) can lead to sub-optimal query plans as the query plan in Fig. 1a shows. On the other hand this also true for more conservative cardinality estimation models that assume the higher join cardinalities, for example the sum, which may lead to overestimating cardinalities and too conservative query plans.

The motivating example illustrates the challenge of query planning in the absence of fine-grained statistics in client-side SPARQL query evaluation over remote data sources such as TPF servers. In such scenarios, query plans should ideally consider not only the estimated cost of the query plans but also its robustness with respect to errors in the join cardinality estimations. Therefore, we investigate how query planning for TPFs can be improved by considering not only the cost of a given query plan but also its robustness. Furthermore, we investigate for which class of queries the concept of robustness is most beneficial and whether such queries can be identified by our robustness metric.

3 Our Approach

We propose CROP, a Cost- and Robustness-based query plan Optimizer to devise efficient plans for SPARQL queries over Triple Pattern Fragment (TPF) servers. The key components of our approach are: (Sect. 3.1) **a cost model** to estimate the cost of executing a query plan over a TPF server, (Sect. 3.2) **the concept of plan robustness** to assess how robust plans are with respect to join cardinality estimation errors, and (Sect. 3.3) **a query plan optimizer** combining both cost and robustness to obtain efficient query plans.

3.1 Cost Model

We present a cost model for estimating the cost of query plans for conjunctive SPARQL queries, i.e. Basic Graph Patterns (BGPs). Given a query plan P for a conjunctive query $\mathcal{Q} = \{tp_1, \dots, tp_n\}$ with $|\mathcal{Q}| = n$ triple patterns, the cost of P is computed as

$$Cost(P) := \begin{cases} 0 & \text{if } P \text{ is a leaf } tp_i \\ Cost(P_1 \bowtie P_2) + Cost(P_1) + Cost(P_2) & \text{if } P = P_1 \bowtie P_2. \end{cases} \quad (1)$$

where $Cost(P_1 \bowtie P_2)$ is the cost of joining sub-plans P_1 and P_2 using the physical join operator \bowtie . Note that the cost for a leaf is 0 as its cost is accounted for

as part of the join cost $Cost(P_i \bowtie P_j)$. In our model, the cost of joining two sub-plans is comprised of two aspects: (i) *request cost*, the cost for submitting HTTP requests to the TPF server if necessary; and (ii) *processing cost*, the cost for processing the results on the client side. Hence, the cost of joining sub-plans P_1 and P_2 using the join operator \bowtie is given by:

$$Cost(P_1 \bowtie P_2) := Proc(P_1 \bowtie P_2) + Req(P_1 \bowtie P_2) \quad (2)$$

where $Proc$ are the processing cost and Req the request cost. Note that both components depend on the physical join operator. First, we distinguish the processing cost joining sub-plans P_1 and P_2 using the physical operator Symmetric Hash Join (\bowtie_{SHJ}) and Nested Loop Join (\bowtie_{NLJ}) as

$$Proc(P_1 \bowtie P_2) := \begin{cases} \phi_{SHJ} \cdot card(P_1 \bowtie P_2), & \text{if } \bowtie = \bowtie_{SHJ} \\ \phi_{NLJ} \cdot (card(P_1 \bowtie P_2) + card(P_2)), & \text{if } \bowtie = \bowtie_{NLJ} \end{cases} \quad (3)$$

Note the first parameter of the cost model $\phi \in [0, \infty)$ allows for weighting the local processing cost with respect to the request cost. For instance, $\phi = 1$ indicates that processing a single tuple locally is equally expensive as one HTTP request. The impact of processing cost and request cost on the query execution time depends on the scenario in which the TPF server and client are deployed. In a local scenario, where network latency and the load on the TPF server are low, the impact of the processing cost on the execution time might be higher than in a scenario with high network latency, where the time for submitting requests has a larger share on the execution time. Furthermore, including $card(P_2)$ in the processing cost for the NLJ allows the optimizer to estimate the cost of alternative plans more accurately. For instance, if we assume the minimum as the estimation function and do not consider the cardinality of the inner relation, a plan $(A \bowtie_{NLJ} B)$ could be chosen over $(A \bowtie_{NLJ} C)$ even if B has a higher cost than C .

The expected number of requests to be performed for joining two sub-plans depends on the physical operator and the estimated number of results produced by the sub-plans. Therefore, we introduce the request cost function for two common physical join operators. In the following, we denote $|P_i|$ as the number of triples in sub-plan P_i .

Nested Loop Join. The cost of a Nested Loop Join (NLJ) combines the cost induced by the requests for obtaining the tuples of the outer plan P_1 and then probing the instantiations in the inner plan P_2 . Therefore, the request costs are computed as

$$Req(P_1 \bowtie_{NLJ} P_2) := \llbracket |P_1| = 1 \rrbracket \cdot \left\lceil \frac{card(P_1)}{p} \right\rceil + d(P_1, P_2) \cdot \max \left\{ card(P_1), \left\lceil \frac{card(P_1 \bowtie P_2)}{p} \right\rceil \right\}, \quad (4)$$

⁴where p is the page size of the TPF server, and $d(P_1, P_2)$ computes a discounting factor. In this work, we restrict the inner plan in the NLJs to be triple patterns only, i.e. $|P_2| = 1$, as it allows for more accurate request cost estimations. The first summand calculates the number of requests for obtaining solution mappings for P_1 . In case P_1 is a triple pattern, the number of requests is given the cardinality of P_1 divided by the page size. The second summand is the estimated number of requests to be performed on the inner plan multiplied by a discount factor. The minimum number of requests that need to be performed is given by the cardinality for P_1 , i.e. one request per binding. However, in the case the join produces more results per binding than the page size, such that paginating is required to obtain all solutions for one binding in the inner relation, we need to consider this in the maximum as well. The discounting factor is computed using the parameter $\delta \in [0, \infty)$ and the maximum height of the sub-plans as

$$d(P_1, P_2) := (\max\{1, \delta \cdot \text{height}(P_1), \delta \cdot \text{height}(P_2)\})^{-1}.$$

The rationale for including a discount factor for the requests on the inner plan of the NLJ is twofold. First, since the join variables are bound by the terms obtained from the outer plan, the number of variables in the triple pattern is reduced and as the empirical study by Heling et al. [10] indicates, this also leads to a reduction in response times of the TPF server. Second, for star-shaped queries, typically the number of results reduces with an increasing number of join operations and, therefore, the higher an NLJ is placed in the query plan, the more likely it is that it needs to perform fewer requests in the inner plan than the estimated cardinality of the outer relation suggests. The discount factor $d(P_1, P_2)$ allows for considering these aspects and its parameter $\delta \in [0, \infty)$ allows for setting the magnitude of the discount factor. With $\delta = 0$, there is no discount and with increasing δ , placing NLJs higher in the query plan becomes increasingly cheaper.

Symmetric Hash Join. The request cost is computed based on the number of requests that need to be performed if either or both sub-plans are triple patterns.

$$Req(P_1 \bowtie_{SHJ} P_2) := \llbracket |P_1| = 1 \rrbracket \cdot \left\lceil \frac{card(P_1)}{p} \right\rceil + \llbracket |P_2| = 1 \rrbracket \cdot \left\lceil \frac{card(P_2)}{p} \right\rceil. \quad (5)$$

Note that the request cost can be computed accurately as it only depends on the metadata provided by the TPF server, with $card(P_i) = count(P_i)$ if $|P_i| = 1$.

Cardinality Estimation. The previous formulas for computing the cost rely on the expected number of intermediate results produced by the join operators, which is determined by recursively applying a join cardinality estimation function. Given two sub-plans P_1 and P_2 , we estimate the cardinality as the minimum of the sub-plans' cardinalities:

$$card(P_1 \bowtie P_2) := \min(card(P_1), card(P_2)), \quad (6)$$

⁴ $\llbracket \cdot \rrbracket$ denote Iverson brackets that evaluate to 1 if its logical proposition is true and to 0 otherwise.

where the cardinality for a single triple pattern is obtain from the metadata provided by the TPF server $card(P_i) = count(P_i), \forall |P_i| = 1$. In our cost model, we choose the minimum as the cardinality estimation function since it showed good results in our preliminary analysis. Next, we will show how the concept of robustness helps to avoid choosing the cheapest plan merely based on these optimistic cardinality estimations.

3.2 Robust Query Plans

Accurate join cardinality estimations aid to find a suitable join order and to properly place physical operators in the query plan to minimize the number of requests. However, estimating the cardinalities is challenging, especially when only basic statistics about the data are available. To address this challenge, we propose the concept of robustness for SPARQL query plans to determine how strongly the cost of a plan is impacted when using alternative cardinality estimations. The core idea is comparing the *best-case* cost of a query plan to the *average-case* cost. To obtain the average-case cost, the cost of the query plan is computed using different cardinality estimation functions for joining sub-plans. The results are several cost values for the query plan under different circumstances which can be aggregated to an averaged cost value. Consequently, a robust query plan is a query plan in which the best-case cost only slightly differs from the average-case cost.

Example 1. Let us revisit the query from the motivating example. For the sake of simplicity, we only consider the sub-plan $P = ((tp_1 \bowtie tp_2) \bowtie tp_3)$ and request cost with $\delta = 0$. Let us consider the alternative query plans $P_1 = ((tp_1 \bowtie_{NLJ} tp_2) \bowtie_{NLJ} tp_3)$ and $P_2 = ((tp_1 \bowtie_{NLJ} tp_2) \bowtie_{SHJ} tp_3)$. For comparing the robustness of P_1 and P_2 , we not only use the cardinality estimation of the cost model (the minimum, cf. Eq. 6) but compute the cost using different cardinality estimation functions, for example using the maximum and mean as alternatives. The resulting cost values allow for deriving the average-case cost and thus the robustness of P_1 and P_2 . Depending on the cardinality estimation function, we obtain the following cost for the query plans P_1 and P_2 :

	Cardinality Estimation Function		
	minimum	mean	maximum
$Cost(P_1)$	5	43,477.45	86,950.88
$Cost(P_2)$	15	444.45	874.88

Query plan P_1 yield the lowest *best-case* cost. However, we observe that the cost for query plan P_2 is not as strongly impacted by different estimation functions. Hence, its average-case cost does not deviate as strongly from its best-case cost in comparison to P_1 . As a result, query plan P_2 is considered a more robust query plan.

Definition 1 (Query Plan Robustness). Let P be a physical query plan for query \mathcal{Q} , $Cost^*(P)$ the best-case and $\overline{Cost}(P)$ the average-case cost for P . The

query plan robustness for P is defined as

$$Robustness(P) := \frac{Cost^*(P)}{Cost(P)}.$$

Namely, the robustness of a plan is the ratio between the cost in the *best-case* $Cost^*$ and the cost in the *average-case* $Cost$. A higher ratio indicates a more robust query plan since its expected average-case cost are not as strongly affected by changes in the cardinality estimations with respect to its best-case cost. Next, we extend the definition of our $Cost$ function to formalize the average-case cost of a query plan. Let $O = \{o_1, \dots, o_{n-1}\}$ be the set of binary join operators for plan P ($|P| = n$) for a conjunctive query \mathcal{Q} , and $E = [e_1, \dots, e_{n-1}]$ a vector of estimation functions with e_i the cardinality estimation function applied at join operator o_i . A cardinality estimation function $e_i : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$ maps the cardinalities of a join operators' sub-plans $a = card(P_1)$ and $b = card(P_2)$ to an estimated join cardinality value. We then denote the cost for a query plan P computed using the cardinality estimation function given by E as $Cost_E(P)$.

Definition 2 (Best-case Cost). *The best-case cost for a query plan P is defined as*

$$Cost^*(P) := Cost_E(P), \text{ with } e_i = f, \forall e_i \in E, \text{ and } f : (a, b) \mapsto \min\{a, b\}.$$

In other words, at every join operator in the query plan, we use the minimum cardinality of the sub-plans to estimate the join cardinality. This is identical to the estimations used in our cost model. The computation of the average-case cost requires applying different combinations of such estimation functions at the join operators.

Definition 3 (Average-case Cost) *Let $F = \{f_1, \dots, f_m\}$ be a set of m estimation functions with $f : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0, \forall f \in F$. The average-case cost for a query plan P is defined as the median of its cost when applying all potential combinations of estimation functions $E \in F^{n-1}$ for the operators of the query plan:*

$$\overline{Cost}(P) := \text{median}\{Cost_E(P) \mid \forall E \in F^{n-1}\}.$$

We empirically tested different sets of estimation functions in F and found that the following produce suitable results for the average-case cost: $F = \{f_1, f_2, f_3, f_4\}$ with

$$\begin{aligned} f_1 : (a, b) &\mapsto \min\{a, b\}, & f_2 : (a, b) &\mapsto \max\{a, b\}, \\ f_3 : (a, b) &\mapsto a + b, & f_4 : (a, b) &\mapsto \max\{a/b, b/a\}. \end{aligned}$$

Furthermore, we observed that for subject-object (s-o) and object-object (o-o) joins the cardinalities were more frequently misestimated in the original cost model, while for all other types of joins, such as star-shaped groups, it provided adequate estimations. Therefore, we only consider alternative cardinality estimation function e_i for a join operator o_i , if the join performed at o_i is either of type s-o or o-o.

3.3 Query Plan Optimizer

The idea of robust query plans yields two major questions: (i) when should a more robust plan be chosen over the cheapest plan, and (ii) which alternative plan should be chosen instead? To this end, we propose a query plan optimizer combining cost and robustness to devise efficient query plans. Its parameters define when a robust plan is chosen and which alternative plan is chosen instead. The main steps of the query plan optimizer are:

1. Obtain a selection of query plans based on Iterative Dynamic Programming (IDP).
2. Assess the robustness of the cheapest plan.
3. If the cheapest plan is not robust enough, find an alternative more robust query plan.

The process is detailed in Algorithm 1. The inputs are a SPARQL Query \mathcal{Q} , block size $k \in [2, \infty)$, the number of top $t \in \mathbb{N}$ cheapest plans, the robustness threshold $\rho \in [0, 1]$ and the cost threshold $\gamma \in [0, 1]$. The first step is to obtain a selection of alternative query plans using IDP. We adapted the original “*IDP₁ – standard – bestPlan*” algorithm presented by Kossmann and Stocker [11] in the following way. Identical to the original algorithm, we only consider select-project-join queries, i.e. BGP queries, and each triple pattern $tp_i \in \mathcal{Q}$ is considered a *relation* in the algorithm. Given a subset of triple patterns $S \subset \mathcal{Q}$, the original algorithm considers the single optimal plan for S according to the cost model in $optPlan(S)$ by applying the *prunePlans* function to the potential candidate plans. However, as we do want to obtain alternative plans, we keep the top t cheapest plans for S in $optPlan(S)$ for $|S| > 2$. When joining two triple patterns ($|S| = 2$), we always chose the physical join operator with the lowest cost for the following reasons: We expect accurate cost estimations for joining two triple patterns as the join estimation error impact is low and it reduces the number of plans to be evaluated in the IDP.

Example 2. Consider the query for the motivating example and $S_1 = \{tp_1, tp_2\}$. According to the cost model, the cheapest plan is $optPlan(S_1) = \{(tp_1 \bowtie_{NLJ} tp_2)\}$. However, for $|S| > 2$ we need to place at least two join operators where the cost of at least one join operator relies on the estimated cardinality of the other. Therefore, we need to keep alternative plans in case a robust alternative plan is required. For instance with $S_2 = \{tp_1, tp_2, tp_3\}$, the optimal plan according to the cost model is $P_1 = ((tp_1 \bowtie_{NLJ} tp_2) \bowtie_{NLJ} tp_3)$, however as it turns out, the true optimal plan for S is $P_2 = ((tp_1 \bowtie_{NLJ} tp_2) \bowtie_{SHJ} tp_3)$. As a result, the algorithm cannot prune all but one plan such that it can choose an alternative robust plan if necessary. Combining the latter observations, we can set $optPlan(S_2) = \{P_1, P_2\}$.

Given the set of t candidate query plans \mathcal{P} from IDP, the overall cheapest plan P^* is determined (Line 2). If the cheapest plan P^* is considered robust enough according to the robustness threshold ρ , it becomes the final plan and is returned

Algorithm 1: CROP Query Plan Optimizer

Input: BGP query \mathcal{Q} , block size k , top t , robustness threshold ρ , cost threshold γ

```

1  $\mathcal{P} = \text{IDP}(\mathcal{Q}, k, t)$ 
2  $P^* = \arg \min_{P \in \mathcal{P}} \text{Cost}(P)$ 
3 if  $\text{Robustness}(P^*) < \rho$  and  $|\mathcal{P}| > 1$  then
4    $\mathcal{R} = \{R \mid R \in \mathcal{P} \wedge \text{Robustness}(R) \geq \rho\}$ 
5   if  $\mathcal{R} == \emptyset$  then
6      $\mathcal{R} = \mathcal{P} \setminus \{P^*\}$ 
7    $R^* = \arg \min_{R \in \mathcal{R}} \text{Cost}(R)$ 
8   if  $\frac{\text{Cost}(P^*)}{\text{Cost}(R^*)} > \gamma$  then
9      $P^* = R^*$ 
10
11 return  $P^*$ 

```

(Line 10). However, if the plan is not robust enough with respect to ρ and there are alternative plans to choose from (Line 3), the query plan optimizer tries to obtain a more robust alternative plan. First, the planner selects all plans which are above the robustness threshold as \mathcal{R} . If no such plans exist, it will consider all alternative plans except the cheapest plan. If the ratio of best-case cost of the cheapest plan P^* to the best-case cost of the alternative plan R^* is higher than the cost threshold γ , the alternative plan R^* is selected as the final plan P^* . For instance, for $\rho = 0.1$ and $\gamma = 0.2$, a robust plan is chosen over the cheapest plan if (i) for the cheapest plan P^* , $\text{Cost}(P^*)$ is 10 times higher than $\text{Cost}^*(P^*)$ and (ii) for alternative robust plan R^* , $\text{Cost}^*(R^*)$ is no more than 5 times ($1/\gamma$) higher than $\text{Cost}^*(P^*)$. Hence, smaller robustness threshold values lead to selecting alternative plans when the cheapest plan is less robust, and smaller cost threshold values lead to less restriction on the alternative robust plan with respect to its cost. The combination of both parameters allows for exploring alternative robust plans (ρ) but does not require to choose them at any cost (γ) and therefore limit the *performance degradation risk* [19]. Next, we investigate the time complexity of the proposed optimizer.

Theorem 1. *With the number top plans t and the set of estimation functions F constant, the time complexity of the query plan optimizer is in the order of*

Case 1: $\mathcal{O}(2^n)$, for $2 \leq k < n$,

Case 2: $\mathcal{O}(3^n)$, for $k = n$.

Proof. The time complexity of the query plan optimizer is given by the IDP algorithm and computing the average-case cost in the robustness computation. Kossmann and Stocker [11] provide the proofs for the former. For the latter, given $|F| = m$ different estimation functions and the top t query plans, the upper bound for the number of alternative cardinality estimations per query plan is $t \cdot m \cdot 2^{n-1}$. As t and m are considered constants, the time complexity of

the robustness computation is in the order of $\mathcal{O}(2^n)$. Combining these complexity results, we have:

Case 1: For $k < n$, the time complexity of computing the robustness exceeds the time complexity of IDP, which is $\mathcal{O}(n^2)$, for $k = 2$ and $\mathcal{O}(n^k)$, for $2 < k < n$. As a result, the time complexity is in the order of $\mathcal{O}(2^n)$.

Case 2: For $k = n$, the time complexity of IDP exceeds the time complexity of the robustness computation and therefore, we have that the time complexity of the query plan optimizer is in the order of $\mathcal{O}(3^n)$.

4 Experimental Evaluation

We empirically evaluate the query plan optimizer with the proposed cost model and the concept of robust query plans. First, we study how the parameters of the cost model and the IDP algorithm impact on the efficiency of the query plans. Thereafter, we study different robustness and cost thresholds in the query plan optimizer to find a good combination of both. Finally, we compare our implementation with the found parameters to state of the art TPF clients: `comunica-sparql` (See footnote 2) (referred to as `Comunica`) and `nLDE` (See footnote 3).

Datasets and Queries. We use the datasets used in previous evaluations: DBpedia 2014 (`nLDE`) and WatDiv [3] (`Comunica`). For DBpedia, we choose a total of 35 queries including Q1-Q10 from the `nLDE` Benchmark 1 and all from Benchmark 2. For WatDiv, we generated a dataset with scale factor = 100 and the corresponding default queries with query-count = 5 resulting in a total of 88 distinct queries.⁵ The resulting 123 queries are used for our experimental study. In addition, to showcase the benefits of including the robustness in the query plan optimizer on a variety of datasets, we designed an additional benchmark with 10 queries for 3 datasets (DBpedia 2014, GeoNames 2012⁶, DBLP 2017 (See footnote 6)) that include either a `s-o` and or an `o-o` join and 3-4 triple patterns.⁷

Implementation. CROP is implemented based on the `nLDE` source code (See footnote 3). We additionally implemented our cost model, robustness computation and the query plan optimizer.⁸ No routing adaptivity features, i.e. routing policies, are used in our implementation. The engine was implemented in Python 2.7.13 and we used the `Server.js` v2.2.3⁹ to deploy the TPF server with HDT backend for all datasets. Experiments were executed on a Debian Jessie 64 bit machine with CPU: 2x Intel(R) Xeon(R) CPU E5-2670 2.60 GHz (16 physical cores), and 256 GB RAM. The timeout was set to 900s. After a warm-up run, the queries were executed three times in all experiments.

⁵ Complex queries do not contain placeholders, leading to one distinct query in `C1`, `C2`, and `C3`.

⁶ <http://www.rdfhdt.org/datasets/>.

⁷ <https://github.com/Lars-H/crop-analysis>.

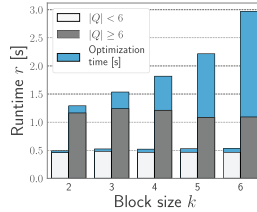
⁸ <https://github.com/Lars-H/crop>.

⁹ <https://github.com/LinkedDataFragments/Server.js>.

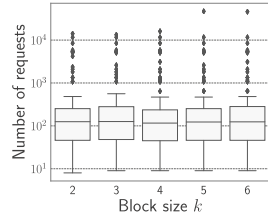
Evaluation Metrics. The following metrics are computed: (i) *Runtime*: Elapsed time spent by a query engine to complete the evaluation of a query. For our implementation, we measure both the *optimization time* to obtain the query plan and the *execution time* to execute the plan separately. (ii) *Number of Requests*: Total number of requests submitted to the server during the query execution. (iii) *Number of Answers*: Total number of answers produced during query execution. If not stated otherwise, we report mean values for all three runs. All raw results are provided in the supplemental material.

δ	Runtime	Requests	PCC
0	1236.45	84,604	0.27
1	1106.58	81,735	0.44
2	971.51	83,209	0.49
3	940.97	88,763	0.52
4	938.44	88,437	0.52
5	995.29	93,815	0.5
6	983.64	99,157	0.49
7	987.08	99,438	0.49

(a) Mean Runtime, number of requests and the Pearson Correlation Coefficient (PCC) for height discounts δ .



(b) Median runtime for queries with $|\mathcal{Q}| < 6$ and $|\mathcal{Q}| \geq 6$ for block sizes k . Indicated in blue is the optimization time.



(c) Box plots of the number of requests per query (log-scale) for block sizes k .

Fig. 2. Experimental results for the parameters of the cost model and IDP.

4.1 Experimental Results

Cost Model and IDP Parameters. First, we investigate how the parameters of the cost model impact runtime and the number of requests. In the optimizer, we disable robust plan selection ($\rho = 0.00$), set the default block size to $k = 3$, and select the top $t = 5$ plans. We focus on the parameter δ which we set to $\delta \in \{0, 1, 2, 3, 4, 5, 6, 7\}$. We do not investigate different processing cost parameters and set $\phi_{NLJ} = \phi_{SHJ} = 0.001$, as they are more relevant when considering different deployment scenarios, where network delays have a stronger impact on the cost. Figure 2a shows the mean runtime, the mean number of requests per run, and the Pearson Correlation Coefficient (PCC) between the cost and the number of requests. The latter provides an indication of how well the estimated cost of a query plan reflects the actual number of requests to be performed. The best runtime results are observed for $\delta = 4$, even though the number requests are about 8% higher than for $\delta = 1$. Furthermore, the highest positive correlation between the estimated cost and the number of requests of a plan are observed for $\delta \in \{3, 4\}$ with $PCC = 0.52$. The worst runtime results are observed for $\delta = 0$, which is equal to no height discount at all. This shows that the parameter allows for adjusting the cost model such that the estimated cost better reflects the number of requests resulting in more efficient query plans. Based on the findings, we set $\delta = 4$ in the following experiments.

Table 1. Mean runtime (r), mean number of requests (Req.) and the number of robust plans ($|R^*$) selected by the query plan optimizer. Indicate in bold are best overall runtime and minimum number of requests.

γ	$\rho = 0.05$			$\rho = 0.10$			$\rho = 0.15$			$\rho = 0.20$			$\rho = 0.25$		
	r	Req.	$ R^*$	r	Req.	$ R^*$	r	Req.	$ R^*$	r	Req.	$ R^*$	r	Req.	$ R^*$
0.1	556.86	64,167	16	534.16	82,657	17	2048	99,015	28	2094	103,709	31	2784	124,070	38
0.3	552.44	64,157	16	533.7	82,629	17	930	105,337	10	937	105,537	10	940	105,728	11
0.5	957.0	86,640	6	911.71	90,932	6	910	91,175	9	908	91,388	9	911	91,511	9
0.7	950.98	86,634	6	909.64	90,962	6	910	91,161	8	913	91,173	5	909	91,298	5
0.9	947.87	86,627	6	915.23	90,934	6	909	90,937	7	907	90,939	4	910	90,986	2

Next, we focus on the parameter of the IDP algorithm and investigate how the block size impacts on both the efficiency of the query plans and the optimization time to obtain these plans. We set $t = 5$ and keep $\delta = 4$ as suggested by the previous experiment. We study $k \in \{2, 3, 4, 5, 6\}$. The median runtimes \tilde{r} for all queries per k are shown in Fig. 2b. Note that $k = \min\{k, |\mathcal{Q}|\}$ and, therefore, we show the results separately for small queries ($|\mathcal{Q}| < 6$) and larger queries ($|\mathcal{Q}| \geq 6$). Indicated in blue is the proportion of the optimization time. The results show that for small queries the median runtime, as well as the optimization time proportion, is similar for all k . However, for larger queries, the median runtimes increase with k . Interestingly, this increase is due to an increased proportion of optimization time spent on obtaining *ideally* better plans. At the same time the execution time (lower part of the bars) is similar ($k = 4$) or slightly lower ($k \in \{5, 6\}$). The box plots for the number of requests for the query plans per k are shown in Fig. 2c. The results show that the median number of requests is minimal for $k = 4$ and the 25 – 75% quantile is most compact for $k = 4$ as well. Moreover, the most extreme outliers are observed with $k = 5$ and $k = 6$. Based on these observations to avoid disproportionate optimization times but still explore the space of possible plans sufficiently, we set k in a dynamic fashion with: $k = 4$ if $|\mathcal{Q}| < 6$ and $k = 2$ otherwise.

Robustness and Cost Threshold. After determining appropriate parameters for the cost model and IDP, we investigate the parameters that impact the decision when an alternative robust plan should be chosen over the cheapest

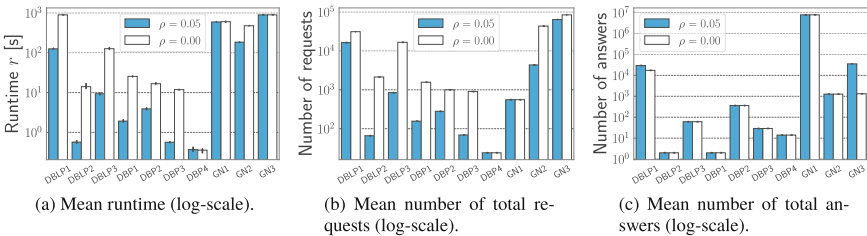
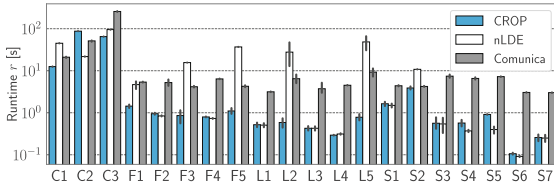


Fig. 3. Results for the 10 queries of the custom benchmark.

plan. The robustness threshold ρ determines *when* an alternative plan should be considered, while the cost threshold γ limits the alternative plans to those which are not considered too expensive. We tested all 25 combinations of $\rho \in \{0.05, 0.10, 0.15, 0.20, 0.25\}$ and $\gamma \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ and run all queries for each combination three times. The averaged results of the total runtimes and the number of requests per run for all 25 parameter configurations are listed in Table 1. Also included is the number of queries for which an alternative robust query plan was selected over the cheapest plan as $|R^*|$. Lowest runtime and number of requests are indicated in bold. The parameters configuration $\rho = 0.10$, $\gamma = 0.3$ yield the best runtime results while the lowest number of requests are performed with the configuration $\rho = 0.05$, $\gamma = 0.3$. Taking a closer look at the two configurations, we find that the runtime is only about 3.5% higher for $\rho = 0.05$, $\gamma = 0.3$, but the number of requests is about 22% lower. Moreover, when comparing the values for $\rho = 0.05$ to $\rho = 0.10$ for all cost threshold values, we find that the runtimes and the total number of requests for $\rho = 0.05$ (388,227) is substantially lower than for $\rho = 0.10$ (438,116) while the total runtime is just slightly higher for $\rho = 0.05$ (3965.16 s) than for $\rho = 0.1$ (3804.43 s). Following these findings, we set the cost threshold to $\gamma = 0.3$ and the robustness threshold $\rho = 0.05$ in the following experiments. The results in Table 1 show that for the parameter configuration $\rho = 0.05$, $\gamma = 0.3$, for 16 out of 123 queries the robust alternative query plan R^* is chosen over the cheapest plan. And 15 out of the 16 queries stem from the WatDiv L2, L5, and S7 queries. To show that other classes of queries for which the more efficient alternative robust query plan can be identified using our approach, we investigate the experimental results for our benchmark with 10 queries over the three datasets DBpedia (DBP), GeoNames (GN), DBLP (DBLP). We keep the same parameters that we obtained from the previous experimental evaluation on the other benchmarks. In Fig. 3, the results of always choosing the cheapest query plans ($\rho = 0.00$) and the results when enabling robust query plans to be chosen with $\rho = 0.05$ and $\gamma = 0.3$ are shown. It can be observed that in 8 queries (DBLP1-3, DBP1-3, GN2-3) robustness allows for obtaining more efficient query plans. For these queries, the runtime and total number of requests are lower and at the same time, the robust alternative query plans produce the same number of answers or even more. Even though the runtime of the robust query plan for query GN3 reaches the timeout, it produces more answers with fewer requests during the time. The results show that our approach devises efficient query plans even for queries where the cost model produces high cardinality estimation errors. The low robustness of the cheapest plans drives our optimizer to choose more robust plans which reduce the query execution times as well as the number of requests.

Comparison to the State of the Art. Given the parameters determined by the previous experiments, we want to compare the performance of the proposed approach to existing TPF clients, namely nLDE and Comunica. Analogously to the previous experiments, we run the 123 queries from both the nLDE Benchmark and WatDiv with all three engines. In Fig. 4a the mean runtimes for all three clients are shown for the WatDiv queries. The results show that our



(a) Mean runtime (log-scale) on the WatDiv queries for our approach, nLDE and comunica.

	CROP	nLDE	Comunica
$\sum r$	540.69	2815.3	1087.3
\tilde{r}	0.71	0.7	2.71
Req.	64,155	151,594	204,453
Ans.	456,090	438,056	456,054
$\frac{\text{Ans.}}{\text{Req.}}$	1.18	0.79	0.22

(b) Summarized results for all TPF clients for all queries.

Fig. 4. Experimental results: comparison to state-of-the-art clients.

proposed approach has a similar performance to the existing engines nLDE and Comunica. Only for the complex query C2 our approach yields the highest average runtime, while for 6 query types (C1, C3, F1, F3, F5, L5) it outperforms the other engines. The results for all queries are summarized in Fig. 4b. Regarding the runtime, our approach yields the lowest overall runtime ($\sum r$) while Comunica has the second-highest and nLDE the highest overall runtime. Taking a closer look, we find that nLDE reaches the timeout (900s) for queries Q5 and Q8 in the nLDE Benchmark 1, explaining the highest overall runtime. In contrast, nLDE has the lowest the median runtime \tilde{r} . Our approach produces the highest number of answers while Comunica only produces a few answers less. The fewest answers are produced by nLDE, likely due to the queries where the timeout is reached. Next, we consider the mean ratio of answers and requests per query (Ans./Req.) as a measure of productivity. It can be observed that our approach on average produces the most answers per request, followed by nLDE and then Comunica with the fewest answers per request. Increasing this productivity can have two key benefits: (i) it reduces query runtimes at the client, and (ii) reduces the load on the TPF server. Finally, we additionally investigated the diefficiency [2] to evaluate the continuous efficiency of the clients. The highest *dieff@t* (where t is set to the maximum execution time across all engines per query) is observed in 39% of all queries for CROP, 54% of all queries for nLDE and 7% of all queries for Comunica. The results suggest that even though the overall runtimes for CROP are the lowest, nLDE is outperforming the approach with respect to its continuous behavior of producing answers. Additional results for queries Q11-Q20 of the nLDE Benchmark 1 are provided as part of our supplemental material online (See footnote 7). For those queries, we observe that all engines time out more often, yet CROP outperforms nLDE and Comunica in the remaining queries.

Concluding our findings, the results show that the proposed approach is competitive with existing TPF clients and on average produces more efficient query plans minimizing both the runtime and the number of requests to be performed. Nonetheless, it must be pointed out that in contrast to heuristics, the proposed cost model and query optimizer rely on parameters that need to be chosen appropriately. On one side, this allows for adapting these parameters

to the specifics of different types of datasets to be queried. On the other side, it might require preliminary testing to optimize the parameter settings.

5 Related Work

We start by discussing cost model-based decentralized SPARQL query processing approaches and approaches for querying different Linked Data Fragments (LDF). An overview of these approaches with their main features is presented in Table 2.

The first group of approaches [7, 8, 13–15] consists of engines that evaluate queries over federations of SPARQL endpoints. These approaches rely on statistics to perform query decomposition, source selection, and estimate the intermediate results of sub-queries which is the basis for their cost model. The contents and granularity of these statistics vary across the approaches. While the specific computation of cost, supported physical join operators and sub-query cardinality estimations differ for all these approaches, their commonality is factoring in the cost of transferring the result tuples. SPLENDID [8], SemaGrow [7] and Odyssey [13] rely on Dynamic Programming (DP) while DARQ [14] implements Iterative Dynamic Programming (IDP) and CostFed [15] implements a greedy heuristic to find an efficient plan. However, all of the aforementioned approaches rely on dataset statistics for accurate cardinality estimations and they do not consider the concept of robust query plans with respect to errors in the estimations.

With the advent of Triple Pattern Fragments (TPFs), different approaches for decentralized SPARQL query processing over this Web interface have been introduced. The TPF Client proposed by Verborgh et al. [17] evaluates conjunctive SPARQL queries (BGPs) over TPF server. The TPF Client intertwines query planning and evaluation based on the metadata provided by the TPF server to minimize the number of requests: the triple pattern with the smallest estimated number of matches is evaluated and the resulting solution mappings are used to instantiate variables in the remaining triple patterns. This procedure is executed continuously until all triple patterns have been evaluated. Comunica [16] is a meta query engine supporting SPARQL query evaluation over heterogeneous interfaces including TPF servers. The authors propose and evaluate two heuristic-based configurations of the engine. The *sort* configuration sorts all triple patterns according to the metadata and joins them in that order, while the *smallest* configuration does not sort the entire BGP, but starts by selecting the triple pattern with the smallest estimated count on each evaluation call. The network of Linked Data Eddies (nLDE) [1] is a client for adaptive SPARQL query processing over TPF servers. The query optimizer in nLDE builds star-shaped groups (SSG) and joins the triple patterns by ascending number of estimated matches. The optimizer places physical operators to minimize the expected number of requests that need to be performed. Furthermore, nLDE realizes adaptivity by adjusting the routing of result tuples according to changing runtime conditions and data transfer rates based. Different from the existing clients, our query plan optimizer relies on a cost model, the concept of robust query plans, and IDP to generate alternative, potentially efficient plans.

Table 2. Overview and features of decentralized SPARQL query processing approaches over different Linked Data Fragment (LDF) interfaces.

Approach	Federation	LDF	Query Planner			
			Cost	Robustness	Statistics	Strategy
DARQ [14]	✓	SPARQL	✓	✗	Service descriptions	IDP
SPLendid [8]	✓	SPARQL	✓	✗	VOID descriptions	DP
SemaGrow [7]	✓	SPARQL	✓	✗	TP-based statistics	DP
Odyssey [13]	✓	SPARQL	✓	✗	Characteristic sets	DP
CostFed [15]	✓	SPARQL	✓	✗	Data summaries	Heuristic
TPF Client [17]	✓	TPF	✗	✗	TPF metadata	Heuristic
nLDE [1]	✗	TPF	✓	✗	TPF metadata	Heuristic
Comunica [16]	✓	TPF, SPARQL	✗	✗	TPF metadata	Heuristic
brTPF Client [9]	✗	brTPF	✗	✗	TPF metadata	Heuristic
SaGe [12]	✗	SaGe-SERVER	✗	✗	–	Heuristic
smart-KG [4]	✗	SMART-KG	✗	✗	TPF metadata	Heuristic
CROP	✗	TPF	✓	✓	TPF metadata	IDP

Other LDF interfaces include brTPF, smart-KG, and SaGe. Hartig et al. [9] propose bindings-restricted Triple Pattern Fragments (brTPF), an extension of the TPF interface that allows for evaluating a given triple pattern with a sequence of bindings to enable bind-join strategies. To this end, the authors propose a heuristic-based client that builds left-deep query plans which aims to reduce the number of requests and data transferred. Smart-KG [4] is a hybrid shipping approach proposed to balance the load between client and server when evaluating SPARQL queries over remote sources. The smart-KG server extends the TPF interface by providing access to compressed partitions of the graph. The smart-KG client determines which subqueries are evaluated locally over the partitions and which triple pattern requests should be evaluated at the server. SaGe [12] is a SPARQL query engine that supports Web preemption by combining a preemptable server and a corresponding client. The server supports the fragment of full SPARQL which can be evaluated in a preemptable fashion. As a result, the evaluation of BGPs is carried out at the server using a heuristic-based query planner. Our client focuses on the TPF interface and can be adapted to support additional LDF interfaces. For instance, by extending the cost model with bind joins to support brTPF or implementing our concept of robustness as part of the query planner in the smart-KG client or the SaGe server.

In the realm of relational databases, various approaches addressing uncertainty in statistics and parameters of cost models have been suggested. In their survey, Yin et al. [19] classify robust query optimization methods with respect to estimation errors, which can lead to sub-optimal plans as the error propagates through the plan. One class of strategies they present are *Robust Plan Selection* approaches in which not the “optimal” plan but rather a “robust” plan that is less sensitive to estimation errors are chosen. For instance, robust approaches may use a probability density function for cardinality estimations instead of single-point values [5]. Other approaches define cardinality estimation intervals where the size of the intervals indicate the uncertainty of the optimizer [6]. In

a recent paper, Wolf et al. [18] propose robustness metrics for query plans and the core idea is considering the cost for a query plan as a function of the cardinality and selectivity estimations at all edges in the plan. Similar to our work, the authors propose computing the k -cheapest plans and selecting the estimated most robust plan. These works rely on fine-grained statistics to assess the selectivity of joins, however, in a decentralized scenario, it is not possible to obtain such detailed dataset information. Therefore, we propose a robust plan selection approach and introduce a new concept of robustness for SPARQL query plans based on the TPF metadata.

6 Conclusion and Future Work

We have proposed CROP, a novel cost model-based robust query plan optimizer to devise efficient query plans. CROP implements a cost-model and incorporates the concept of robustness for query plans with respect to cardinality estimations errors. Our proposed concept of robust query plans is based on comparing the best-case to the average-case cost of query plans and could be combined with other existing cost models as well. Combining these concepts, CROP uses iterative dynamic programming (IDP) to determine alternative plans and decides when a more robust query plan should be chosen over the cheapest query plan. In our experimental study, we investigated how the parameters of the cost model and IDP impact the efficiency of the query plans. Thereafter, we studied different combinations of the robustness and the cost thresholds in the query plan optimizer. The parameters allow for finding a good balance between choosing alternative robust plans over the cheapest plan but not at any cost. Therefore, our concept of robustness complements the cost model in helping to find better query plans. Finally, we compared our approach to existing TPF clients. The results show that our approach is competitive with these clients regarding runtime performance. Moreover, the query plans of our query plan optimizer require fewer requests to produce the same number of results and, thus, reduce the load on the TPF server. Future work can focus on alternative strategies to IDP for exploring the space of plans more efficiently and extending the optimizer to apply the concept of robustness in federated querying. Our robust query planning approach may be implemented in Linked Data Fragment clients such as Comunica or smart-KG and the cost model may be further extended to include bind joins supported by brTPF. Besides, Linked Data Fragment interfaces, such as SaGe (HDT) may also benefit from including the notion of query plan robustness.

Acknowledgement. This work is funded by the German BMBF in QUOCA, FKZ 01IS17042.

References

1. Acosta, M., Vidal, M.-E.: Networks of Linked Data Eddies: an adaptive web query processing engine for RDF data. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 111–127. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_7
2. Acosta, M., Vidal, M.-E., Sure-Vetter, Y.: Diefficiency metrics: measuring the continuous efficiency of query processing approaches. In: d’Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 3–19. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_1
3. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
4. Azzam, A., Fernández, J.D., Acosta, M., Beno, M., Polleres, A.: SMART-KG: hybrid shipping for SPARQL querying on the web. In: WWW 2020: The Web Conference 2020 (2020)
5. Babcock, B., Chaudhuri, S.: Towards a robust query optimizer: a principled and practical approach. In: ACM SIGMOD International Conference on Management of Data (2005)
6. Babu, S., Bizarro, P., DeWitt, D.J.: Proactive re-optimization. In: ACM SIGMOD International Conference on Management of Data (2005)
7. Charalambidis, A., Troumpoukis, A., Konstantopoulos, S.: SemaGrow: optimizing federated SPARQL queries. In: SEMANTICS 2015 (2015)
8. Görlitz, O., Staab, S.: SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In: COLD 2011 (2011)
9. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: Debruyne, C., et al. (eds.) OTM 2016. LNCS, vol. 10033, pp. 762–779. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48472-3_48
10. Heling, L., Acosta, M., Maleshkova, M., Sure-Vetter, Y.: Querying large knowledge graphs over triple pattern fragments: an empirical study. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11137, pp. 86–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_6
11. Kossmann, D., Stocker, K.: Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.* **25**(1), 43–82 (2000)
12. Minier, T., Skaf-Molli, H., Molli, P.: Sage: web preemption for public SPARQL query services. In: WWW 2019: The Web Conference 2019 (2019)
13. Montoya, G., Skaf-Molli, H., Hose, K.: The *Odyssey* approach for optimizing federated SPARQL queries. In: d’Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10587, pp. 471–489. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68288-4_28
14. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 524–538. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68234-9_39
15. Saleem, M., Potocki, A., Soru, T., Hartig, O., Ngomo, A.N.: CostFed: cost-based query optimization for SPARQL endpoint federation. In: SEMANTICS 2018 (2018)
16. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular SPARQL query engine for the web. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11137, pp. 239–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_15

17. Verborgh, R., et al.: Triple pattern fragments: a low-cost knowledge graph interface for the web. *J. Web Semant.* **37**, 184–206 (2016)
18. Wolf, F., Brendle, M., May, N., Willems, P.R., Sattler, K., Grossniklaus, M.: Robustness metrics for relational query execution plans. *PVLDB* **1**(11), 1360–1372 (2018)
19. Yin, S., Hameurlain, A., Morvan, F.: Robust query optimization methods with respect to estimation errors: a survey. *SIGMOD Rec.* **44**(3), 25–36 (2015)