Robust Query Processing for Linked Data Fragments

Lars Heling [a,b,*] and Maribel Acosta [c,*]
[a] *Institute AIFB, Karlsruhe Institute of Technology, Germany*
[b] *Corporate Research, Robert Bosch GmbH, Germany*
E-mail: heling@kit.edu, lars.heling@de.bosch.com
[c] *Faculty of Computer Science, Ruhr University Bochum, Germany*
E-mail: maribel.acosta@rub.de

Editors: Axel-Cyrille Ngonga Ngomo, University of Paderborn, Germany; Muhammad Saleem, University of Leipzig, Germany; Ruben Verborgh, Ghent University – imec, Belgium
Solicited reviews: Hala Skaf-Molli, Nantes University, France; Stasinos Konstantopoulos, National Centre of Scientific Research "Demokritos", Greece; Oscar Corcho, Universidad Politécnica de Madrid, Spain

Abstract. Linked Data Fragments (LDFs) refer to interfaces that allow for publishing and querying Knowledge Graphs on the Web. These interfaces primarily differ in their expressivity and allow for exploring different trade-offs when balancing the workload between clients and servers in decentralized SPARQL query processing. To devise efficient query plans, clients typically rely on heuristics that leverage the metadata provided by the LDF interface, since obtaining fine-grained statistics from remote sources is a challenging task. However, these heuristics are prone to potential estimation errors based on the metadata which can lead to inefficient query executions with a high number of requests, large amounts of data transferred, and, consequently, excessive execution times. In this work, we investigate robust query processing techniques for Linked Data Fragment clients to address these challenges. We first focus on robust plan selection by proposing CROP, a query plan optimizer that explores the cost and robustness of alternative query plans. Then, we address robust query execution by proposing a new class of adaptive operators: Polymorphic Join Operators. These operators adapt their join strategy in response to possible cardinality estimation errors. The results of our first experimental study show that CROP outperforms state-of-the-art clients by exploring alternative plans based on their cost and robustness. In our second experimental study, we investigate to what extent different planning approaches can benefit from polymorphic join operators and find that they enable more efficient query execution in the majority of cases.

Keywords: SPARQL, Query Processing, Robustness, Adaptivity, Linked Data Fragments

1. Introduction1

Linked Open Data initiatives led to the publication of Knowledge Graphs covering a large variety of domains on the Web[1]. Linked Data Fragments (LDFs) refer to Web interfaces for publishing and querying such Knowledge Graphs [1]. In recent years, several LDF interfaces have been proposed which mainly differ in their expressivity and the metadata they provide [1–4]. For example, Triple Pattern Fragments (TPF) is a popular LDF interface that allows for querying Knowledge Graphs with high availability [1]. TPF servers provide a lightweight interface that supports triple pattern-based querying to reduce server-side costs and increase server availability. 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. More expressive LDF interfaces allow for exploring different trade-offs in client- and server-side query processing. This led to

*Corresponding authors. E-mail: heling@kit.edu, lars.heling@de.bosch.com, maribel.acosta@rub.de.

[1] https://lod-cloud.net/

the development of specific clients to support SPARQL query processing over these interfaces. A key challenge of such clients is devising efficient query plans that minimize the overall query execution time by reducing the data transferred and the number of requests submitted to the server. To this end, the clients commonly rely on the metadata of the LDF interfaces and implement heuristics to achieve efficient query processing [1–3, 5, 6]. However, a drawback of these heuristics is the fact that they fail to adapt to different classes of queries which can lead to extensive runtimes while producing many requests. This can be attributed to the following reasons. First, the clients often follow specific planning paradigms resulting in either left-deep or bushy query plans and do not explore alternative plans [5, 6]. Second, due to the limited metadata of the LDF interfaces, the clients rely on basic cardinality estimations to determine the join order and to place physical operators. Finally, even though many clients process queries in an adaptive fashion with non-blocking operators and adapting the join order, they still adhere to the predefined join strategies during execution set by the planner.

In this work, we investigate robust query processing techniques for LDFs to address these limitations. Robust query processing comprises different techniques with the common goal to overcome inefficient query execution performance caused by query planning errors and unexpected adverse runtime conditions [7, 8]. These approaches accept the fact that cost-models and cardinality estimation approaches can be inaccurate and aim to implement query processing techniques that are not highly affected by potential inaccuracies, planning errors, and unexpected runtime conditions. Our techniques aim to support robustness with respect to cardinality estimation errors at two points during query processing: (i) *Query planning*, by devising efficient query plans that consider both the cost and robustness of plans, and (ii) *Query execution* with *intra-operator adaptivity*, to switch the join strategies in response to wrongly placed physical join operators. Therefore, we focus on the following research question.

RQ 14 How can we measure the robustness of query plans with respect to cardinality estimation errors? With the first research question, we want to investigate a suitable measure to determine the robustness of query plans during query planning. Specifically, we want to study means to assess the robustness of a query plan in the presence of high-level metadata, that is commonly provided by LDF interfaces.

RQ 24 How does incorporating robustness during query planning impact the efficiency of query plans? The second research question focuses on the impact on query execution efficiency when incorporating robust query plan selection in the optimizer. With this question, we want to study the trade-off between selecting a robust alternative plan over the cheapest plan. To this end, we investigate how a feasible selection of alternative robust plans can be determined and under which conditions the selection of a robust plan is favorable.

RQ 34 To what extent does adapting the join strategies during query execution support robust query processing? Finally, we want to understand whether runtime adaptivity allows for overcoming inefficient query execution due to cardinality estimation errors. Since query planners rely on cardinality estimations for placing physical join operators, estimation errors may lead to the selection of sub-optimal operators. Therefore, we investigate whether adapting the join strategy of physical join operators in response to estimation errors increases the execution robustness.

In this work, we study these questions using the example of the Triple Pattern Fragment (TPF) interface due to the following reasons. First, the existing state-of-the-art clients for TPFs [5, 6] follow different query planning paradigms to which we can compare the effectiveness of our approach. Second, other queryable LDF interfaces also support the evaluation of triple patterns as the atomic component of SPARQL. As a result, our approach can be extended and tailored to clients and servers of more expressive LDF interfaces. Lastly, the insights gained from our experimental study on TPFs can be used as a basis for future investigations on robust query processing for other LDFs.

Contributions4 This work is based on a previous paper of ours [9], which studies cost- and robustness-based query optimization (CROP) for Linked Data Fragments. We extend this work by refining our cost model to better generalize for other LDF interfaces. Moreover, we study the concept of robustness from the perspective of adaptive query processing for Linked Data Fragments. To this end, we propose a new class of operators that we call *Polymorphic Join Operators*. These operators aim to achieve robustness by adapting their join strategy during query execution in response to potential cardinality estimation errors. In summary, the novel contributions of this work are as follows:

C 1 a refined cost model, additional experimental results, and a more detailed evaluation of CROP,

C 2 new adaptive join operators: the Polymorphic Bind Join (PBJ) and Polymorphic Hash Join (PHJ),

C 3 results on the theoretical properties and correctness of the PBJ and the PHJ, and

C 4 an experimental evaluation of the PBJ and PHJ for different query planning approaches.

Structure of this Paper4 The remainder of this paper is structured as follows. We present a motivating example in Section 2. In Section 4, we present our cost model, robustness metric, and our query plan optimizer that combines cost and robustness. In Section 5, we present two adaptive join operators from a new class of Polymorphic Join Operators. We empirically evaluate the effectiveness of our query planning approach and the adaptive join operators in Section 6. In Section 3, we discuss related work. Lastly, we summarize our contributions and point to future work in Section 7.

2. Motivating Example1

As a motivating example, consider the query in Listing 1 that obtains *persons with "Stanford University" as their alma mater, the title of their thesis, and their doctoral advisor* from the Triple Pattern Fragment (TPF) server for the English version of DBpedia[2] with a page size of 100. The number of estimated triples matching each triple pattern (count) provided as metadata from the TPF server is also indicated in Listing 1.

Listing 1: Query to get persons with "Stanford University" as their alma mater, the title of their thesis and their doctoral advisor.

```
SELECT * WHERE {
 ?u rdfs:label "Stanford University''@en . # count(tp1) = 2
 ?s dbo:almaMater ?u .                      # count(tp2) = 86088
 ?s dbp:thesisTitle ?t .                    # count(tp3) = 1187
 ?s dbo:doctoralAdvisor ?d .                # count(tp4) = 4885
}
```

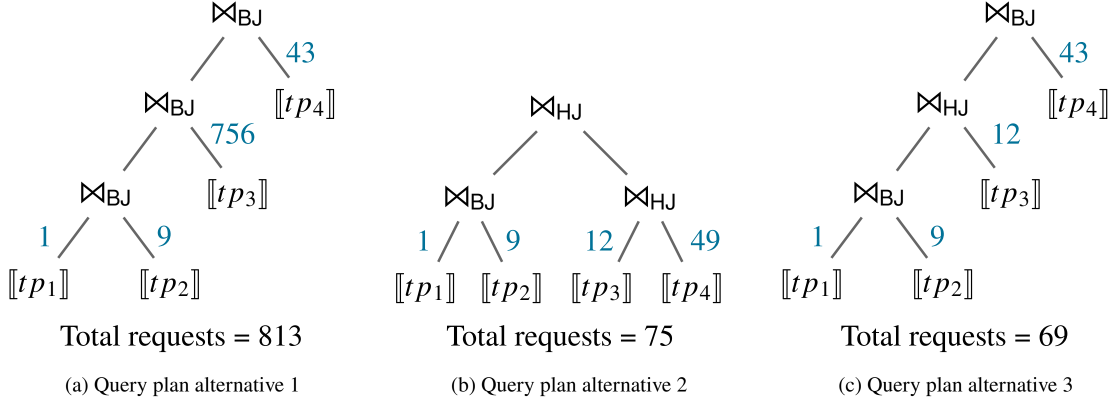
Evaluating SPARQL queries over the TPF server requires the client to obtain efficient query plans that minimize the query execution time, the number of requests, and the amount of data transferred. Typically, clients implement a query planning heuristic that relies on the metadata provided by the TPF server. The *sort* heuristics implemented by comunica sorts the triple patterns according to the number of triples they match in ascending order and places Nested Loop Joins (NLJs) as the physical operators [6]. Evaluating the query over the TPF server using comunica-sparql[3] requires the client to perform 813 requests to obtain the 29 results of the query. The corresponding physical query plan is shown in Figure 1a, where the number of requests is indicated on the edges. The client performs 4 requests to obtain the statistics (counts) on the triple patterns, and thereafter, it executes the plan with 809 requests, leading to a total of 813 requests. An alternative query planning heuristic is implemented in the network of Linked Data Eddies (nLDE)[4] which is another client for for TPF servers. The query planning heuristic builds bushy plans around star-shaped subqueries and places Nested Loop Join or Symmetric Hash Join (SHJ) operators such that the estimated number of requests is minimized [5]. To this end, nLDE first performs 4 requests to obtain the count values of the triple patterns and thereafter, builds the bushy query plan as shown in Figure 1b. The execution of the query plans requires 71 requests, leading to a total of 75 requests. When inspecting the query in detail, we observe that neither comunica-sparql nor nLDE finds the query plan which minimizes the number of requests to be performed. The optimal plan is shown in Figure 1c and it requires a total of 69 requests only: 4 requests to obtain the counts and 65 requests to execute the plan. The number of requests in the query plan can be reduced by sorting the triple patterns similar to comunica by ascending count values. Furthermore, placing the appropriate physical join operators according to the join cardinalities of the sub-plans minimizes the number of requests.

The example query showcases the challenge for heuristics to devise efficient query plans based only on the count statistic provided by the TPF servers. In the query, the subject-object join of triple patterns tp_1 and tp_2 yields 756 results. This can be difficult to estimate relying on the TPF metadata alone with $count(tp_1) = 2$ and $count(tp_2) = 86088$. On the one hand, an optimistic heuristic assuming small join cardinalities (for example the minimum) can lead to sub-optimal query plans as the query plan in Figure 1a shows. On the other hand, a more conservative cardinality estimation model that assume the higher join cardinalities, for example, the sum, may lead to overestimating cardinalities and to too conservative query plans. Consequently, accu-

[2]http://fragments.dbpedia.org/2014/en

[3]https://github.com/comunica/comunica

[4]https://github.com/maribelacosta/nlde



(a) Query plan alternative 1

(b) Query plan alternative 2

(c) Query plan alternative 3

Fig. 1. Three alternative query plans for the SPARQL query from Listing 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).

rate join cardinality estimations are crucial to obtain efficient query plans. However, they are challenging to compute in the absence of fine-grained statistics in client-side SPARQL query evaluation over remote data sources such as TPF servers. Therefore, when following the *optimize-then-execute* paradigm, a robust query planning approach may help to identify query plans that are less prone to cardinality estimation errors.

In addition, adaptive query processing strategies [10] allow to eradicate potential query planning errors by adapting the join processing during query plan execution. Take for example the query plan in Figure 1a which requires a large number of requests by probing each individual tuple from $[\![tp_1 \text{ AND } tp_2]\!]$ in the inner relation tp_3. An optimistic planner which assumes that $[\![tp_1 \text{ AND } tp_2]\!]$ produces few tuples would choose an NLJ operator in this case, even though obtaining all tuples from tp_3 to perform an SHJ merely requires 12 requests. Consequently, knowing that probing each tuple from $[\![tp_1 \text{ AND } tp_2]\!]$ requires at least one request using a NLJ, the NLJ is guaranteed to require more requests than the SHJ if $|[\![tp_1 \text{ AND } tp_2]\!]| > 12$. Instead of continuing to follow a predefined join strategy, an adaptive client could decide to change the join strategies based on the information it obtains during query execution. In the example, the client could decide to switch to a SHJ after realizing that $|[\![tp_1 \text{ AND } tp_2]\!]| > 12$, which would reduce the number of requests from 813 to just 82 requests. The number of requests are given by: 4 requests to obtain the count values, 10 requests for $[\![tp_1 \text{ AND } tp_2]\!]$, 13 requests for probing tuples in tp_3 before switching, 12 requests for obtaining all tuples from tp_3 after switching, and 43 requests for probing the tuples in tp_4.

Our motivating example illustrates how efficient client-side query processing over TPF server can be achieved by (i) obtaining efficient query plans that are less prone to cardinality estimation errors, and (ii) adapting the join strategy during query execution. In Section 4, we present our approaches for robust query planning that not only considers the best-case scenario but also an average-case scenario when determining an efficient query plan. Moreover, in Section 5, we present the concept of *Polymorphic Join Operators* which are able to adapt their join strategy during query execution.

3. Background and Related Work1

Different cost models and adaptive techniques have been proposed in federated SPARQL query engines. Therefore, we first present related work on federated SPARQL query processing (§3.1). Thereafter, we present the planning techniques implemented by client-side query processing for Linked Data Fragments (§3.2). Finally, we present adaptive and robust query processing approaches from the area of relational databases which address estimation errors in query planning and query execution (§3.3).

3.1. Federated SPARQL Query Processing2

A variety of cost models [11–15] and adaptive query processing approaches [16, 17] have been proposed to employ efficient federated SPARQL query processing. DARQ [11] implements a cost model to reduce the amount of data transferred and the number of transmissions. The cost estimation functions for nested loop

joins and bind joins rely on join cardinality estimations derived from statistical information from the service descriptions of the federation members. The optimizer uses Iterative Dynamic Programming (IDP) to obtain an efficient query plan. The federated engine SPLENDID [12] also implements a cost model to devise efficient plans. The cost model incorporates the network communication based on the estimated number of tuples to be transferred for either a bind or a hash join operator. Join cardinalities are estimated using precomputed indices based on VoID descriptions. The optimizer uses Dynamic Programming (DP) to find the cost-optimal plan. The cost model in SemaGrow [13] incorporates the costs for querying endpoints, transferring tuples, and processing them locally. The cardinality estimations for computing these costs rely on detailed statistics including the number of distinct subjects, predicates, and objects for a given triple pattern. The query plan optimizer uses DP to enumerate the space of possible join plans and prunes inferior plans. The cost function of Odyssey [14] is only based on the cardinalities of intermediate results to favor plans that produce fewer intermediate results. The cardinalities of intermediate results are estimated using characteristics sets statistics of the federation members, and DP is applied to enumerate alternative join orders for the subexpressions. CostFed's [15] cost model also relies on detailed data summaries to estimate the cardinalities of subexpressions. The cost model is tailored to physical query plans with symmetric hash joins and bind joins. The optimizer follows a greedy-heuristic to incrementally build sub-plans with minimal cardinalities.

Federated SPARQL query processing approaches that focus on runtime adaptivity include ANAPSID [16] and ADERIS [17]. ANAPSID [16] is a federated SPARQL query engine that adapts to data availability and runtime conditions of endpoints to hide delays from users. The engine implements a query planner based on adaptive sampling and two non-blocking join operators that aim to produce query results even in the case that SPARQL endpoints get blocked. The ADERIS system [17] focuses on adaptive join ordering for federated SPARQL queries. The system relies on basic statistics (predicates per source) for an initial decomposition of the query. In contrast to a static query plan, ADERIS adapts the join order during query execution using a cost model that uses the cardinalities and selectivities determined during runtime.

Similar to our work, existing federated querying approaches [11–15] implement cost models for comparing alternative query plans. The cost models typically combine local processing and network communication cost and consider different join operators. However, they rely on fine-grained cardinality estimations that are based on detailed statistics about the data sources. Moreover, the optimizers are able to employ different plan enumeration approaches, such a DP, since typically the search space for federated plans is smaller as they work with subexpressions which are typically composed of several triple patterns and other operators. Similar to the Polymorphic Join Operators, ANAPSID [16] also focuses on intra-operator adaptivity, however, the join operators aim to adapt to delayed or bursty data traffic rather than cardinality estimation errors. In contrast to our work, ADERIS [17] implements inter-operator adaptivity and the cost model relies on cardinality and selectivity statistics produced during the query execution.

3.2. Linked Data Fragments and Clients2

Linked Data Fragments (LDF) are interfaces for accessing and querying RDF graphs on the Web [1]. A central difference between these interfaces is their expressivity and the metadata they provide [18, 19], resulting in different client-side query processing approaches for LDF interfaces. The original Triple Pattern Fragment (TPF) client [1] supports the evaluation of SPARQL queries over TPF servers and implements a heuristics to process the query which tries to minimize the number of requests. The TPF client implements non-blocking operators and the join order is given by sorting the triple patterns according to their cardinality. 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 during runtime until all triple patterns have been evaluated. Comunica [6] is a modular query engine that supports SPARQL query evaluation over heterogeneous interfaces including TPF servers. The client is embedded in the Comunica framework which aims to provide a research platform for SPARQL query evaluation to support the development of modular, web-based query engines. Comunica currently supports two heuristic-based configurations. The *sort* configuration sorts all triple patterns according to the metadata and joins them in that order similar to [1]. The *smallest* configuration does not sort the entire BGP but selects the triple pattern with the smallest estimated count on every recursive evaluation call. The network of Linked Data Eddies (nLDE) [5] is an adaptive client-side SPARQL

query engine over TPF servers. The query optimizer in nLDE builds star-shaped groups (SSG) and joins the triple patterns by ascending cardinality. The optimizer places either symmetric hash join or nested loop join operators to minimize the expected number of requests that need to be performed. Furthermore, nLDE realizes adaptivity by adjusting the routing of result tuples during the query execution according to changing runtime conditions and data transfer rates.

More expressive LDF interfaces include brTPF and smart-KG. Bindings-restricted Triple Pattern Fragments (brTPF) [2] are an extension of the TPF interface that allows for evaluating a given triple pattern with a sequence of bindings to enable more efficient bind join strategies. Given a triple pattern and sequence of bindings, the brTPF server instantiates the variables of the triple pattern with the bindings and evaluates them over the RDF graph to return the matching triples. The authors propose a heuristic-based client that builds left-deep query plans which aims to reduce the number of requests and data transferred by leveraging bind joins. Smart-KG [3] is a hybrid shipping approach which aims to balance the load between clients and servers when evaluating SPARQL queries over remote sources. The smart-KG server extends the TPF interface by providing access to compressed partitions of the graph. These partitions are based on the concept of *predicate families* to support the evaluation of star-shaped subexpression over the partition at the client. The smart-KG client determines which subexpressions are evaluated locally over the shipped partitions and which triple patterns should be evaluated at the server.

Finally, SaGe [4] is a query engine that supports Web preemption by combining a preemptable server and a corresponding smart client. The server supports the fragment of SPARQL which can be evaluated in a preemptable fashion. The client decomposes a query such that the resulting subexpressions can be answered by the server and it also handles the preemptable execution of the query. As a result, the evaluation of the subexpressions is carried out at the server using a heuristic-based query planner that builds left-deep plans with index loop joins.

Different from the existing LDF clients, our query planner relies on a cost model, a robustness measure, and IDP to devise efficient query plans. Specifically, we focus on the TPF interface to showcase the effectiveness of our approach, however, it can be extended to support additional LDF interfaces. For instance, by extending the *probe* function in our cost-model to also support brTPF with several bindings per request. In addition, more expressive LDF interfaces and their clients may also benefit from our robustness measure. For example, to devise efficient and robust query plans in the smart-KG client or the SaGe server. While existing clients implement adaptivity during runtime [1, 5, 6], in contrast to the Polymorphic Join Operators, these adaptive approaches do not adjust the join strategy in response to estimation errors but focus on changing the join order and routing of tuples during the execution.

3.3. Robust Query Processing in Relational Databases2

In the realm of relational databases, various approaches address uncertainties in the statistics and parameters used in cost models. Wiener et al. [8] consider different types of robustness including (i) *query optimizer robustness* as the ability of the optimizer to choose good plans under unexpected conditions, and (ii) *query execution robustness* as the efficient execution of a given plan under different runtime conditions. Yin et al. [7] focus on the former by investigating robust query optimization methods which are robust with respect to estimation errors. Their classification of such methods includes *Robust Plan Selection*, which comprises approaches that select a "robust" plan which is less sensitive to estimation errors over the "optimal" plan. These approaches, for example, use probability density functions for cardinality estimations instead of single-point values [20] or define cardinality estimation intervals where the size of the intervals indicate the uncertainty of the optimizer [21]. Wolf et al. [22] propose cardinality-based and selectivity-based robustness metrics for query plans. The core idea is computing the cost of a query plan as a function of the cardinality and selectivity estimations at all edges in the plan. The robustness metrics are computed based on the slope and area under the resulting cost function.

The CROP query planner can be considered a *Robust Plan Selection* approach. In contrast to [20] and [21], CROP has to rely on coarse grained statistics that do not allow for computing selectivity or cardinality estimation probabilities. Similar to [22], our measure computes the robustness of query plans and selects a robust plan from a selection of the cheapest plans.

Besides robust query planning, a variety of adaptive query processing approaches [10, 23] have been proposed to support *query execution robustness*. Similar to our work, operator replacement considers approaches, where physical operators can be replaced with a logically equivalent operator at runtime [23]. Op-

erator replacement typically occurs due to mid-query re-optimization [10]. For example, progressive query optimization (POP) [24] adapts to cardinality estimation errors by re-optimizing the query plans potentially leading to operator replacement. To this end, POP determines validity ranges of sub-plan cardinalities that trigger query plan re-optimization if the actual cardinalities violate these ranges. Materialized views and corresponding checkpoints allow the re-optimized query plans to reuse intermediate results.

Similarly, Rio [21] adapts to cardinality estimation errors by (i) considering the robustness of query plans during query optimization based on bounding-boxes for estimations, and (ii) creating *switchable* plans during the optimization phase that can be used as alternatives plans in response to estimation errors at runtime.

The proposed Polymorphic Join Operators can be considered as dynamic operator replacement. However, in contrast to [24] and [21], the Polymorphic Join Operators do not require a re-optimization of the query plan or precomputed switchable plans. Moreover, the Polymorphic Join Operators independently decide whether they adapt their join strategy solely based on runtime conditions without detailed statistics for validity ranges or bounding-boxes. This makes our proposed solution suitable for query execution over LDFs, where the engine has access only to coarse-grained statistics.

4. Robust Query Planning1

We present CROP, a cost- and robustness-based query plan optimizer to devise efficient plans for SPARQL queries over Linked Data Fragment (LDF) servers. An overview of the approach is provided it Figure 2. Given a SPARQL query, the query plan optimizer determines a set of alternative query plans. The efficiency and robustness of these plans are estimated by our cost model and robustness measure. Finally, the optimizer selects a query that yields an appropriate trade-off of both cost and robustness. In the following, we start by introducing the preliminaries and thereafter, present the cost model (§4.2), robustness measure (§4.3), and query planner (§4.4) in detail.

4.1. Preliminaries2

The foundation of this work is the Resource Description Framework (RDF). Consider the three pairwise disjoint sets of Internationalized Resource Identifiers (IRIs) I, blank nodes B, or literals L. An RDF term is an element in $I \cup B \cup L$ and an RDF triple is a 3-tuple of RDF terms: $t = (s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, with s the subject, p the predicate, and o the object of the triple. A finite set of RDF triples is called an RDF graph G and the universe of RDF graphs is denoted by $\mathcal{G}$. SPARQL is the recommended query language for RDF which allows to construct queries by replacing RDF terms with variables. Following the notation introduced by Schmidt et al. [25], let V be the set of variables disjoint from I, B, and L.

Definition 4.1 (SPARQL Expression [25])**.** A SPARQL expression is an expression that is recursively defined as follows.

(1) A triple pattern $tp \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a SPARQL expression.
(2) If P_1 and P_2 are expressions and R is a SPARQL filter condition, then the expressionsP_1 Filter R, $(P_1$ And $P_2)$, $(P_1$ Union $P_2)$ and $(P_1$ Opt $P_2)$ are SPARQL expressions.

Furthermore, we denote the universe of SPARQL expression as $\mathcal{P}$. A basic graph pattern (BGP) P is either a triple pattern or an expression of the form $P = (P_1$ And $P_2)$, where P_1 and P_2 are either a conjunctive expression (And) or a triple pattern. By $[\![P]\!]_G$, we denote the evaluation of a SPARQL expression P over an RDF graph G. We denote the number of triple patterns in a BGP P by $|P|$. We assume set semantics for SPARQL as defined by Schmidt et al [25] and thus, the evaluation of an expression yields a set of *solution mappings* $\Omega = \{\mu_1, \ldots, \mu_n\}$, where a solution mapping is a partial function $\mu : V \rightarrow IBL$, mapping variables to RDF terms. The set of variables for which μ is defined is called the domain of a solution mapping $dom(\mu) \subset V$. Moreover, we define a function $vars : \mathcal{P} \rightarrow V$ that maps an expression to the set of variables in the expression. Finally, given a BGP P and a solution mapping μ, we denote $\mu(P)$ replacing all variables $?x \in dom(\mu) \cap vars(P)$ in P by $\mu(?x)$.

In the remainder of this work, we focus on query plans for BGPs to be evaluated over Linked Data Fragment (LDF) servers. An LDF server is a Web interface to access and query RDF graphs. We identify an LDF server by its IRI $c \in I$. Similar to [26] and [19], we defined a function $ep : I \rightarrow \mathcal{G}$ that maps the IRI of an LDF server to the (default) graph available at the server. Moreover, we denote the type of interface of an LDF server $int(c)$.

Example 4.1. The TPF server for DBpedia of our motivating example can be defined as

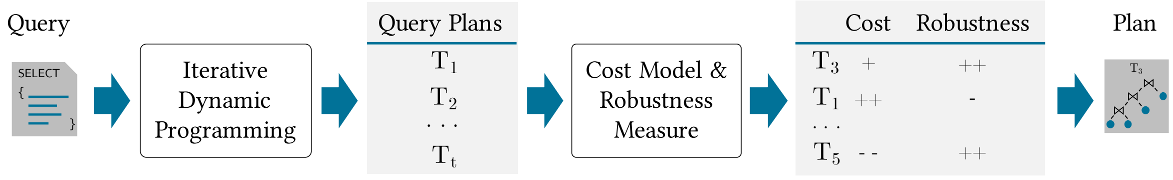


Fig. 2. Overview of CROP

– $c_{DBpedia}$ = <http://fragments.dbpedia.org/2014/en>,
– $ep(c_{DBpedia}) = G_{DBpedia}$, and
– $int(c_{DBpedia})$ = TPF.

The central components of query plans for evaluating BGPs over LDF servers are its access operators. An access operator evaluates a SPARQL expression over a given LDF server by performing the necessary HTTP requests to obtain all solution mappings according to the interface.

Definition 4.2 (Access Operator)**.** An access operator is a tuple $A = (SE, c)$ with $SE \in \mathscr{P}$ a SPARQL expression and c the IRI of an LDF server.

A join query plan defines the join order, the physical join operators, and the access operator for evaluating a BGP P in a tree structure[5].

Definition 4.3 (Join Query Plan)**.** A join query plan T is a binary tree, where the leaves $\mathscr{A}(T) = \{A_1, \ldots, A_n\}$ of the tree are access operators and the internal nodes are physical join operators.

For query plan T, we denote the number of leaves as $|T| = |\mathscr{A}(T)|$. The number of join operations is then given as $|T| - 1$. The longest path from the root of a (sub) query plan T to any access operator is denoted as $height(T)$ and for an access operator A, we have $height(A) = 0$. Moreover, the estimated number of solution mappings obtained by evaluating T is denoted $card(T)$. Note that we distinguish algebraic join operators ($\ltimes\!\!\!\rtimes$) and physical join operators ($\bowtie$). In addition, we indicate the join algorithm of a physical join operator by its subscript. For example, a bind join operator is denoted by $\bowtie_{\mathrm{BJ}}$. In this work, we consider a physical join operator to be *correct*, if it adheres to the set semantics defined in [25].

Definition 4.4 (Evaluation of a Join Query Plan)**.** Given a query plan T, the evaluation of T is defined as

$$eval(T) = \begin{cases} [\![SE]\!]_{ep(c)}, & \text{if } T = A = (SE, c) \\ eval(T_1) \ltimes\!\!\!\rtimes eval(T_2), & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

[5]We only consider binary join operators, resulting in a binary tree.

Furthermore, we define a function $expr(T)$ that maps a query plan T to its algebraic structure. This function traverses the tree structure of the query plan and replaces the access operators by their algebra expression and physical operators by the corresponding algebra operators as defined in Def. 4.4. Hence, $expr(T)$ allows us to compare different plans and to determine whether they are algebraically equivalent.

Proposition 4.1. Given an LDF interface c for RDF graph G with $ep(c) = G$ and a BGP P. The evaluation of a join query plan T for P is correct, that is,

$$eval(T) = [\![P]\!]_G,$$

if $expr(T)$ is algebraically equivalent to P, all physical operators in T are correct, and $c_i = c \ \forall (SE_i, c_i) \in \mathscr{A}(T)$.

The proposition follows from the algebraic equivalences defined by Schmidt et al. [25] and the fact that all expressions are evaluated over the same graph $ep(c)$. Finally, $T(P)$ denotes a query plan for a SPARQL expression P, that is $expr(T) = P$.

4.2. Cost Model2

We now present our cost model to estimate the cost of evaluating a query plan for a basic graph pattern over an LDF server. In principle, the cost for evaluating a query plan in a decentralized scenario are determined by (i) the *server cost* for processing the requests (i.e., evaluating the expression) on the server, (ii) the *network cost* for transporting requests and responses over the network, and (iii) the *client cost*, for processing the tuples of the response. Specifically, determining the server and network costs is challenging in practice since they are influenced by a large number of factors, such as the server load or potential network delays. Therefore, we use the number of requests that need to be performed by an access operator as a proxy for server and network costs. Consequently, for the sake of comparability of the individual requests of the access operator and because any LDF interface (except data

dumps) allows for evaluating triple patterns, we assume the subexpressions in the access operators to be triple patterns, i.e., $SE_i = tp_i, \forall (SE_i, c) \in \mathscr{A}(T)$. Moreover, we only consider query plans which are evaluated over a single LDF server c. The *number* of requests that an access operator needs to perform depends on the specific LDF server. For example, TPF servers require several requests when the number of resulting tuples exceeds the page size of the server, while (in principle) a single request suffices when accessing SPARQL endpoints. Furthermore, other LDF servers, such as brTPF servers, may require fewer requests than TPF servers when probing tuples for a triple pattern because they support probing multiple bindings [2].

In the following, we focus on the request cost for access operators for TPF servers and for the physical join operators Bind Join (BJ) and symmetric Hash Join (HJ). Given a query plan T for a conjunctive query the cost of evaluating T over LDF server c is computed as

$$cost(T) = \begin{cases} 0 & \text{if } T \text{ is a leaf } A_i, \\ cost(T_1 \bowtie T_2) + cost(T_1) + cost(T_2) & \text{if } T = T_1 \bowtie T_2, \end{cases}$$

where $cost(T_1 \bowtie T_2)$ is the cost of joining the solution mappings from sub-plans T_1 and T_2 at the client 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 $cost(T_i \bowtie T_j)$. In our model, the cost of joining two sub-plans is comprised of two aspects: (i) *request cost*, as the cost for submitting HTTP requests to the server if necessary; and (ii) *processing cost*, the client's cost for processing the tuples that it receives from the server. Hence, the cost of joining sub-plans T_1 and T_2 using the join operator $\bowtie$ is given by:

$$cost(T_1 \bowtie T_2) = \phi \cdot proc(T_1 \bowtie T_2) + req(T_1 \bowtie T_2)$$

where $proc$ are the processing cost, req the request cost, and $\phi \in [0, \infty)$ a weighting factor.

4.2.1. Processing Cost3

The processing costs account for the effort of handling the tuples at the client once they have been received from the server. For instance, this includes parsing the tuples into the corresponding data structures and potentially inserting them into hash tables in the join operators. 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 at the client 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 LDF server and client are deployed. In a local scenario, where network latency and the load on the LDF 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. The processing cost depends on the physical join operator $\bowtie$ and we distinguish two cases:

$$proc(T_1 \bowtie T_2) = \begin{cases} card(T_1 \bowtie T_2) & \text{if } \bowtie = \bowtie_{\text{HJ}}, \\ card(T_1 \bowtie T_2) + card(T_2) & \text{if } \bowtie = \bowtie_{\text{BJ}}. \end{cases}$$

In both cases, the estimated tuples produced by the join $card(T_1 \bowtie T_2)$ are considered. Including $card(T_2)$ in the processing cost for the BJ allows the optimizer to estimate the cost of alternative plans more accurately. For instance, if we assume the minimum as the cardinality estimation function and do not consider the cardinality of the inner relation, a plan $(A \bowtie_{\text{BJ}} B)$ could be chosen over $(A \bowtie_{\text{BJ}} C)$ even if B has a higher cost than C.

4.2.2. Request Cost3

In our cost model, we use the request cost as a proxy for the network cost and the server-side cost when evaluating an expression at the server. The request cost $req(T_1 \bowtie T_2)$ for joining two sub-plans T_1 and T_2 are determined by the join operator $\bowtie$ and whether the sub-plans T_1 and T_2 are access operators. In the following, we present how the request cost for the access operators as well as for a bind join and a symmetric hash join operators are computed in our cost model. In line with the evaluation of our approach, we will detail these cost functions for Triple Pattern Fragment servers. Nonetheless, the functions can be extended to support other LDF interfaces as well.

Access Operator4 If a T is an access operator $A = (SE, c)$, the cost of its requests are given by the number of requests that need to be performed to obtain all solution mappings for the expression SE at the LDF server c. Otherwise, if T is a sub-plan (i.e., $height(T) > 0$), no requests costs are associated with it. Specifically, we focus on access operators for a TPF server c ($int(c) =$ TPF) where the expression SE of the access operator is a triple pattern tp. Therefore, the request cost for an access operator for a TPF server c with page size p_c evaluating triple pattern tp over $ep(c)$ is given as

$$acc(T) = \begin{cases} \left\lceil \frac{card(tp,c)}{p_c} \right\rceil & \text{if } T = A = (tp, c), \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Bind Join4 The request costs of a Bind Join (BJ) are determined by the request cost for obtaining the tuples of the outer plan $acc(T_1)$ and the request cost for probing the instantiations in the inner plan T_2: $probe(T_1, T_2)$. Therefore, the request costs for the BJ are computed as

$$req(T_1 \bowtie_{\mathrm{BJ}} T_2) = acc(T_1) + d(T_1, T_2) \cdot probe(T_1, T_2)$$

where $probe(T_1, T_2)$ is the estimated number of requests for probing all tuples of $eval(T_1)$ in the inner plan T_2 and $d(T_1, T_2)$ is a factor for discounting the probing cost, which we detail in Eq. (3). In the remainder of this work, we only focus on BJs where T_2 is an access operator for a triple pattern because it allows for more accurate request cost estimations. In case T_1 is leaf, i.e., an access operator $A_1 = (tp_1, c)$, the number of requests to obtain the tuples from T_1 is given by $acc((tp_1, c))$ from Eq. (1). Otherwise, there are no request cost associated with T_1. In either case, the number of requests that need to be performed to probe the tuples from T_1 in T_2 needs to be considered. This number depends on the number of tuples in $eval(T_1)$, the number of tuples that are produced when probing each tuple, and the number of instantiations that can be probed at the server in a single request. For a TPF server c, the *accurate* number of requests for probing each tuple $\mu \in eval(T_1)$ in triple pattern tp is given by

$$\sum_{\mu \in eval(T_1)} \left\lceil \frac{[\![\mu(tp)]\!]_{ep(c)}}{p_c} \right\rceil$$

In practice, there are two major reasons why this cannot be computed accurately. First, the number of tuples in $eval(T_1)$ needs to be estimated by a cardinality estimation function. Second, it is infeasible to determine how many tuples are produced by instantiating the individual solution mapping. Therefore, we use the estimated cardinality of T_1 as a lower bound and the estimated join cardinality divided by the page size p_c as an upper bound.

$$probe(T_1, T_2) = \max\{card(T_1), \left\lceil \frac{card(T_1 \bowtie T_2)}{p_c} \right\rceil\}. \quad (2)$$

The minimum number of requests that need to be performed is given by the cardinality for T_1, i.e. one request per binding. However, it might be the case that 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. In this case, we use the estimated join cardinality and the page size to estimate the requests. Note that while we focus on Bind Join operators with a block size of 1, the *probe* function can be adapted for LDF servers that support bind join strategies with a block size > 1 as well, such as brTPF servers [2].

The discounting factor for BJs is computed using the parameter $\delta \in [0, \infty)$ and the maximum height of the sub-plans as

$$d(T_1, T_2) = \frac{1}{\max\{1,\ \delta \cdot height(T_1),\ \delta \cdot height(T_2)\}}. \quad (3)$$

The rationale for including a discount factor for the requests on the inner plan 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 which can reduce the cost per request. This was shown in an empirical study for TPF servers [27]. Second, for star-shaped queries, typically the number of tuples reduces with an increasing number of join operations and, therefore, the higher the BJ operator 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 $d(T_1, T_2)$ allows for considering these aspects and its parameter δ allows for setting the magnitude of the discount factor. With $\delta = 0$, there is no discount and with an increasing δ value, placing BJs higher in the query plan becomes increasingly cheaper.

Symmetric Hash Join4 The request cost for the symmetric Hash Join (HJ) operator is computed based on the number of requests that need to be performed if either or both sub-plans T_1 and T_2 are access operators.

$$req(T_1 \bowtie_{\mathrm{HJ}} T_2) = acc(T_1) + acc(T_2)$$

If both T_1 and T_2 are access operators, we sum up the corresponding number of requests according to the access operator. If just one sub-plan (e.g., T_2) is an access operator, we need to consider the number of requests for its access operator. Otherwise, when joining two sub-plans with $height(T_1) > 0$ and $height(T_2) > 0$,

there are no requests that need to be performed by the join operator.

Note that the number of requests for the HJ can be computed accurately if the true cardinalities of the expressions of the access operators are known. For example, the *count* metadata that provides (an estimation of) the number of triples matching a triple pattern allows for determining the number of requests accurately.

Cardinality Estimation4 Central to our cost model are the expected number of intermediate results produced by the join operators as this number affects both the local processing and the request costs. Since we focus on query plans for BGPs, we determine the number of intermediate results by recursively applying a join cardinality estimation function to the query plan T. Given a query plan T, we estimate the cardinality as

$$card(T) = \begin{cases} card_{acc}(SE, c) & \text{if } T = A = (SE, c), \\ \min\{card(T_1), card(T_2)\} & \text{if } T = T_1 \bowtie T_2. \end{cases} \tag{4}$$

To compute the cardinality estimation for an access operator $A = (SE, c)$ depends on the expression SE, the LDF server c, and the corresponding graph $ep(c)$. In the case of TPF servers, we can leverage the "estimate of the cardinality" [1] provided in the metadata when requests a triple pattern which we denote by $count$. Therefore, we use $card_{acc}(tp, c) = count(tp)$. Furthermore, in our cost model, we choose the minimum as the cardinality estimation function for joining sub-plans T_1 and T_2 as an optimistic estimation.

After presenting our cost model, we will now present the concept of robustness for query plans in order to avoid always choosing the cheapest plan merely based on these optimistic cardinality estimations.

4.3. Query Plan Robustness2

Query planning approaches benefit from accurate join cardinality estimations to determine a suitable join order and to properly place physical operators such that the execution time of the query plan is minimized. However, estimating the join cardinalities is a challenging task, especially in the case that only basic statistics about the data are available. Addressing this challenge, we propose a robustness measure in order to determine how strongly the costs of a query plan are affected by potential cardinality estimations errors. To this end, our robustness measure compares the *best-case* cost of a query plan to its *average-case* cost. The average-case cost of a query plan is computed by using different cardinality estimation functions in the cost model to cover alternative join cardinalities. The resulting cost for each estimation function and the same query plan can be aggregated to an average cost value. Consequently, a robust query plan is a plan in which the best-case cost only slightly differs from the average-case cost.

Example 4.2. Let us revisit the query plans from our motivating example in Section 2. As we focus on query plans with access operators for the same LDF server c, for the sake of readability, we omit the access operator in the following examples, i.e., $(tp_i, c) = tp_i$. For the sake of simplicity, we only consider the sub-plan $T = ((tp_1 \bowtie tp_2) \bowtie tp_3)$, and focus on the request cost with $\delta = 0$. Let us consider the alternative query plans

$$T_1 = ((tp_1 \bowtie_{\mathsf{BJ}} tp_2) \bowtie_{\mathsf{BJ}} tp_3),$$
$$T_2 = ((tp_1 \bowtie_{\mathsf{BJ}} tp_2) \bowtie_{\mathsf{HJ}} tp_3).$$

For comparing the robustness of T_1 and T_2, we not only use the optimistic cardinality estimation of the cost model (the minimum, cf. Eq. 4) but also compute the cost using different, less optimistic cardinality estimation functions. For instance, we can also consider the maximum and mean as alternatives. The resulting cost values allow for deriving the average-case cost and thus the robustness of T_1 and T_2. Depending on the cardinality estimation function, we obtain the following cost for the query plans T_1 and T_2:

	Cardinality Estimation Function		
	minimum	mean	maximum
$cost(T_1)$	5	43 477	86 951
$cost(T_2)$	15	445	875

Query plan T_1 yield the lowest *best-case* cost when considering the minimum. However, we observe that the cost for query plan T_2 is not as strongly impacted by the alternative estimation functions. As a consequence, its average-case cost does not deviate as strongly from its best-case cost in comparison to T_1 and as a result, query plan T_2 is considered a more robust query plan.

Definition 4.5 (Cost ratio robustness measure)**.** Let T be a query plan, $cost^*(T)$ the best-case and $\overline{cost}(T)$ the average-case cost for T. The cost ratio robustness (crr) for T is defined as

$$crr(T) := \frac{cost^*(T)}{\overline{cost}(T)}.$$

That is, the cost ratio robustness (crr) of a plan is the ratio between the cost in the *best-case cost** and the cost in the *average-case* $\overline{cost}$. A higher ratio indicates a more robust query plan because its expected average-case costs are not as strongly affected by changes in the cardinality estimations with respect to its best-case cost. Note that while we focus on query plans for basic graph patterns in this work, our robustness measure can be applied to query plans for other expressions as well.

We extend the definition of the $cost$ function from Section 4.2 to capture the average-case cost of a query plan by including the cardinality estimation functions applied to each join operator. Let $O = \{o_1, \ldots, o_{n-1}\}$ be the set of (binary) join operators for a query plan T ($|T| = n$). Let $E = [e_1, \ldots, e_{n-1}]$ be a vector of estimation functions with e_i the cardinality estimation function applied at join operator o_i. For the join operator o_i, the cardinality estimation function $e_i : \mathbb{N}_0^2 \to \mathbb{N}_0$ maps the cardinalities of the sub-plans $a = card(T_1)$ and $b = card(T_2)$ to an estimated join cardinality value. In practice, the values of a cardinality estimation function are bound within zero and the cross-product of a and b: $e_i(a, b) \in [0, a \cdot b]$. We then denote the cost for a query plan T computed using the cardinality estimation functions given by E as $cost_E(T)$.

Definition 4.6 (Best-case cost)**.** The best-case cost for a query plan T is defined as

$$cost^*(T) = cost_E(T),$$

with $e_i = f,\ \forall e_i \in E$ 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. Note that while in principle the cardinality for very selective joins can even be lower than the minimum, it still provides an optimistic estimation for computing the best case cost. The computation of the average-case cost requires applying different combinations of such estimation functions at the join operators.

Definition 4.7 (Average-case cost)**.** Given a set of m estimation functions $F = \{f_1, \ldots, f_m\}$ with $f : \mathbb{N}_0^2 \to \mathbb{N}_0,\ \forall f \in F$. The average-case cost for a query plan T 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:

$$\overline{cost}(T) = \text{median}\{cost_E(T) \mid \forall E \in F^{n-1}\}.$$

By applying a variety of cardinality estimation functions, different join selectivities can be reflected in the average-case cost. We empirically tested different sets of estimation functions in F and found that the following functions yield suitable estimations for computing the average-case cost: $F = \{f_1, f_2, f_3, f_4\}$ with

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

The rationale for selecting these functions is to capture different join selectivities based on the cardinality estimations of the sub-plans to be joined. The function f_1 captures high join selectivities (i.e., the join produces few solution mappings), while f_2 and f_3 capture medium and f_4 low join selectivities. Moreover, we observed that for subject-object (s-o) and object-object (o-o) joins, the cardinalities were more frequently misestimated with the optimistic cardinality estimation function, while for all other types of joins, such as in star-shaped groups, it provided adequate estimations. Similar observations about the challenge in estimating join cardinalities for s-o and o-o joins have been reported in other works as well [28]. 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. Thus, for s-s joins, we estimate their cost using only the best-case cost estimation (Def. 4.6). For s-o and o-o join, the alternative cost estimation functions in F are used to compute the average case cost as defined in 4.7.

Example 4.3. Let us consider the query plan alternative 1 from our motivating example (cf. Figure 1). In this example, the join $tp_1 \bowtie tp_2$ is a s-o join and all other joins are s-s joins. As a result, to compute the average case cost, the alternative estimation functions in F are applied to estimate $card(tp_1 \bowtie tp_2)$. Based on these four alternative cardinality estimations, the minimum cardinality (cf. Eq. (4)) is used to estimate the cardinality of the remaining joins (as they are s-s joins). With these cardinality estimation alternatives, the cost of the query plan is computed which results in four different cost estimations. Finally, to obtain the average-case cost value, the median of those four cost estimations is computed.

4.4. Query Plan Optimizer2

After presenting our cost model and robustness measure, we now present our optimizer that combines both

aspects. Introducing the concept of robustness for query plans in addition to their cost yields two major questions: (i) in which cases should a more robust plan be chosen over the cheapest plan, and (ii) which alternative robust plan should be chosen instead of the cheapest plan? To this end, we propose a query plan optimizer that combines both aspects. Its parameters allow for defining the sensitivity of when a robust plan should be selected and also which alternative plan should be chosen over the cheapest plan. The query plan optimizer follows three main steps:

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

The query plan optimizer is detailed in Algorithm 1. Given a BGP P, the query planner determines the best query plan T^* for P. The input parameters are the block size $k \in [2, \infty)$ and the number of top $t \in \mathbb{N}$ cheapest plans for the IDP algorithm. Moreover, the planner relies on a robustness threshold $\rho \in [0, 1]$ and a cost threshold $\gamma \in [0, 1]$. The first step is to obtain a selection of alternative query plans using IDP. We adapted the original "$IDP_1 - standard - bestPlan$" algorithm presented by Kossmann and Stocker [29] in the following way. Identical to the original algorithm, we only consider select-project-join queries, i.e. basic graph patterns, and each triple pattern $tp_i \in P$ is considered a *relation* in the algorithm. Given a subset of triple patterns $S \subset P$, 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 choose the physical join operator with the lowest cost. We follow this strategy for the following reasons. First, we expect accurate cost estimations for joining two triple patterns as the join estimation error impact is low in the base case. Second, by considering fewer alternatives that are likely to be discarded later in the algorithm, we can reduce the number of plans to be explored with IDP without loosing viable alternatives.

Example 4.4. Consider the query for the motivating example and $S_1 = \{tp_1, tp_2\}$. According to the cost model, the cheapest plan is $optPlan(S_1) = \{(tp_1 \bowtie_{\mathsf{BJ}} tp_2)\}$. As $|S_1| = 2$, we only consider this single cheapest sub-plan in the remaining iterations. 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 want to keep alternative plans in the case that 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 $T_1 = ((tp_1 \bowtie_{\mathsf{BJ}} tp_2) \bowtie_{\mathsf{BJ}} tp_3)$. As shown in our motivating example, it turns out that the true optimal sub-plan for S is $T_2 = ((tp_1 \bowtie_{\mathsf{BJ}} tp_2) \bowtie_{\mathsf{HJ}} tp_3)$. As a result, the algorithm does not prune all but a single plan, while keeping alternative plans for the case that a robust plan should be chosen. Combining the latter observations, we can set $optPlan(S_2) = \{T_1, T_2\}$

Algorithm 1: CROP Query Plan Optimizer

Input: BGP P, block size k, top t, robustness threshold ρ, cost threshold γ

1. $\mathcal{T} \leftarrow \texttt{IDP}(P, k, t)$
2. $T^* \leftarrow \arg\min_{T \in \mathcal{T}} \texttt{cost}(T)$
3. **if** $\texttt{crr}(T^*) < \rho \wedge |\mathcal{T}| > 1$ **then**
4. $\quad \mathcal{R} \leftarrow \{R \mid R \in \mathcal{T} \wedge \texttt{crr}(R) \geqslant \rho\}$
5. $\quad$ **if** $\mathcal{R} = \emptyset$ **then**
6. $\quad\quad \mathcal{R} \leftarrow \mathcal{T} \setminus \{T^*\}$
7. $\quad R^* \leftarrow \arg\min_{R \in \mathcal{R}} \texttt{cost}(R)$
8. $\quad$ **if** $\frac{\texttt{cost}(T^*)}{\texttt{cost}(R^*)} > \gamma$ **then**
9. $\quad\quad T^* \leftarrow R^*$
10. **return** T^*

Given the set of t candidate query plans $\mathcal{T}$ from the IDP, the overall cheapest plan T^* is determined (Line 2). If the cheapest plan is considered robust enough according to its cost ratio robustness $\texttt{crr}(T^*)$ and the robustness threshold ρ, it becomes the final plan and is returned (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 considers the set of plans $\mathcal{R}$ which are above the robustness threshold as potential alternatives. If no such plans exist, it considers all alternative plans except the cheapest plan (Line 6). If the ratio of best-case cost of the cheapest plan T^* 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 T^*. 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 T^*, the average-case cost $\overline{cost}(T^*)$ is 10 times higher than the best-case cost $cost^*(T^*)$ and (ii) for the alter-

native robust plan R^*, the best-case cost $cost^*(R^*)$ is no more than 5 times $(1/\gamma)$ higher than best-case cost of the cheapest plan $cost^*(T^*)$. 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, the *performance degradation risk* [7] is limited. Finally, we investigate the time complexity of the proposed optimizer.

Theorem 4.1. With the number top plans t and the set of estimation functions F constant, the time complexity of the query plan optimizer is for a BGP P with n triple patterns in the order of

CASE I: $\mathcal{O}(2^n)$, for $2 \leqslant k < n$,
CASE II: $\mathcal{O}(3^n)$, 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 [29] 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 I: 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 II: 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 $\mathcal{O}(3^n)$.

□

5. A New Class of Adaptive Join Operators1

We now present a new class of adaptive join operators which we call *Polymorphic Join Operators*. The goal of these operators is to improve the runtime efficiency by adapting the join strategy during the execution of a query plan. In particular, these operators adapt to potential join cardinality estimation errors of the planner to achieve an additional level of robustness during query execution.

A central task of the query planner is deciding on the appropriate physical join operators that maximize the query plan's efficiency with respect to the runtime and number of requests. The efficiency depends on the join strategies implemented by the operators and the number of intermediate results they need to process. Based on join cardinalities estimations of the sub-plans, the query planner decides to place operators that implement different join strategies (e.g., bind join, hash join, etc). A client following the *optimize-then-execute* paradigm would then execute the resulting query plan. However, this approach does not allow for adapting to potential estimations errors of the planner which can lead to sub-optimal join strategies. Alternatively, the client could support *intra-operator* adaptivity aiming to support robust query execution. To this end, we propose two new operators in this class of adaptivity, namely a Polymorphic Bind Join and a Polymorphic Hash Join operator. For the sake of simplicity, we present the operators with a bindings block size equal to one, that is a single tuple is probed during the bind join phase. The operators can easily be extended to support larger block sizes for more expressive LDF interfaces.[6]

5.1. Polymorphic Bind Join2

The Polymorphic Bind Join (PBJ) is a physical join operator that can switch its join strategy during query execution from a bind join to a hash join strategy. We denote the PBJ by $\bowtie_{\mathsf{PBJ}}$ and given a query plan $T = T_1 \bowtie T_2$, the PBJ can be placed when T_2 is an access operator in the query plan.

The PBJ operator is outlined in Algorithm 2. The operator receives a stream of tuples from the evaluation of the sub-plan T_1 as Ω_1 (Line 1). The end of the stream is indicated by an end-of-file (`EOF`) tuple. The operator receives and processes the tuples in an asynchronous, non-blocking fashion[7]. In its first phase (Line 4 to 11), the operator follows a bind join strategy and produces

[6] See for example [19], where the concept of *polymorphism* refers to adapting the block size according to the LDF interface while in this work it refers to adapting the join strategy.

[7] In the pseudocode, we use the `receive` keyword to indicate asynchronicity: The operator determines whether the `next` tuple is available in the stream and if this is not the case, it continues its operation without executing the dependent steps.

the resulting tuples to its output (Line 6). The operator keeps track of the number of probed tuples *cnt*, which is used to decide whether it should adapt its join strategy. The `switch` function in Line 8 determines whether the PBJ will switch from the bind join to the hash join strategy. The challenge in deciding whether the strategy should be switched is the fact that the operator does not certainly know the number of tuples that are still remaining from $eval(T_1)$ until the `EOF` is received. In our case, the operator assumes that a bind join strategy was chosen by the planner as it expected that the requests cost for probing all tuples in $eval(T_1)$ would be lower than for a hash join operator. To this end, the operator determines whether to switch its join strategy according to the number of probed tuples (cnt) and the access cost for T_2 if it was to switch to a hash join:

$$\texttt{switch}_{\texttt{PBJ}}(cnt, T_2) = \begin{cases} \texttt{True} & cnt > \lambda \cdot acc(T_2), \\ \texttt{False} & \text{otherwise.} \end{cases} \quad (5)$$

Adding the parameter $\lambda \in (0, \infty)$ allows for setting the sensitivity of the operator. Lower λ values indicate a higher sensitivity as the operator would decide earlier to switch its join strategy. Furthermore, λ could be set according to the height of the operator in the plan.

In the case that the operator decides to switch its join strategy (Line 8), it terminates the loop of the bind join. Thereafter, the operator sets up two hash tables (Line 13 and 14) and starts evaluating T_2 and process the tuples from the resulting stream. In the second phase (Line 17 to 25), the operator implements a non-blocking, symmetric hash join. It produces the join tuples by inserting and probing the remaining tuples it receives from $eval(T_1)$ and all tuples from $eval(T_2)$ in the corresponding hash tables. The operator finishes after receiving the `EOF` from both input streams.

Example 5.1. Let us consider the query plan from our motivating example shown in Figure 1a that is evaluated over the DBpedia TPF server with a page size of 100. We focus on the second join operator $(tp_1 \bowtie tp_2) \bowtie T(tp_3)$ and assume the planner places PBJ operators with $\lambda = 1$. Following an optimistic cardinality estimation (e.g., the minimum) with $card(tp_1 \bowtie tp_2) = 2$, the planner decides for a bind join strategy as probing the expected 2 tuples requires fewer requests than obtaining all tuples for tp_3. The cardinality of tp_3 is given as $count(tp_3) = 1187$ and thus, the request cost for the access operator in a hash join is $acc(T(tp_3)) = \lceil 1187/100 \rceil = 12$. However, the actual number of tuples produced by $tp_1 \bowtie tp_2$ is 756. Hence, the PBJ switches its operation after probing 13 tuples ($13 > 1 \cdot 12$, Eq. (5)) to a hash join strategy. In this example, the PBJ adapts appropriately to the cardinality estimation error of the planner. Switching to the hash join operator requires less requests than probing the remaining $756 - (13 + 12) = 731$ tuples from $tp_1 \bowtie tp_2$. The adaptivity of the operator results in a smaller number of requests and therefore, reduces the overall query execution time.

Algorithm 2: Polymorphic Bind Join

```
   Input: Plan T1, access operator T2 = A2 = (SE2, c)
1  Ω1 ← eval(T1) = {μ^1_1, ..., μ^1_k, EOF}
2  μ' ← Ω1.next()
3  cnt ← 1
4  while μ' ≠ EOF do
5  |   for μ ∈ eval((μ'(SE2), c)) do
6  |   |   output {μ' ∪ μ | μ ~ μ'}
7  |   if switch_PBJ(cnt, T2) then
8  |   |   break
9  |   else
10 |   |   cnt ← cnt + 1
11 |   |   μ' ← Ω1.next()
12 if μ' = EOF then output EOF
   // Switch to Hash Join Strategy
13 H1 ← HashTable()
14 H2 ← HashTable()
15 Ω2 ← eval(T2) = {μ^2_1, ..., μ^2_l, EOF}
16 μ'' ← null
17 while μ' ≠ EOF ∧ μ'' ≠ EOF do
18 |   μ' ← receive Ω1.next() /* Asynchronous      */
19 |   H1.insert(μ')
20 |   for μ ∈ H2.probe(μ') do
21 |   |   output {μ ∪ μ' | μ ~ μ'}
22 |   μ'' ← receive Ω2.next() /* Asynchronous     */
23 |   H2.insert(μ'')
24 |   for μ ∈ H1.probe(μ'') do
25 |   |   output {μ ∪ μ'' | μ ~ μ''}
26 output EOF
```

Parameters for the PBJ4 The PBJ follows a simple heuristic-based decision rule to decide whether it should switch its join strategy. This is due to the fact that it cannot assess how many tuples in $eval(T_1)$ are still remaining to be received. If the bind join strategy of the operator is sub-optimal with respect to the number of requests, the operator should switch to the hash join strategy after probing the first tuple. Since the operator cannot know whether this is the case, it follows the decision rule in $\texttt{switch}_{\texttt{PBJ}}$ to determine the number of tuples (cnt) that it probes in the bind join phase, before switching to the hash join phase. There is a maximum value for cnt for which switching to the

hash join reduces the number of requests in comparison to not switching. We can split the tuples received from $eval(T_1)$ into two disjoint subsets $\Omega_1 = \Omega_1^{BJ} \cup \Omega_1^{HJ}$, with Ω_1^{BJ} the tuples that are received and probed during the bind join phase and Ω_1^{HJ} the tuples received during the hash join phase. The operator can reduce the number of requests if $\lambda \cdot acc(T_2)$ is set such that the operator switches with $cnt = |\Omega_1^{BJ}|$ and

$$\underbrace{acc((SE_2, c))}_{\text{Requests for hash join}} < \underbrace{\sum_{\mu \in \Omega_1^{HJ}} acc((\mu(SE_2), c))}_{\text{Requests for probing remaining tuples}} .$$

In other words, the PBJ decides correctly if it switches to a hash join operation as long as the access requests for T_2 in the hash join are lower than probing the remaining tuples Ω_1^{HJ} in the bind join phase. This will be illustrated in the following example.

Example 5.2. We consider the query plan from Figure 1a with PBJ operators and focus on the second join operator $T = (tp_1 \bowtie tp_2) \bowtie_{\mathsf{PBJ}} tp_3$ with $\Omega_1 = eval((tp_1 \bowtie tp_2))$. Assume that probing each tuple $\mu \in \Omega_1$ requires a single request, and assume that the operator will switch its strategy at $cnt = 13$ (i.e., $\lambda = 1$). This leads to the following outcomes, depending on the actual cardinality of Ω_1:

1. If $|\Omega_1| < 13$: The operator adapts *correctly* by *not switching* the join strategy; switching to a hash join would require at least the same number of requests.
2. If $|\Omega_1| \in [13, 24]$: The operator adapts *incorrectly* by *switching* to a hash join. With $cnt = 13$, the operator switches to the hash join which requires an additional 12 requests, which is more expensive than probing the at most 24 tuples.
3. If $|\Omega_1| > 25$: The operator adapts *correctly* by *switching* the join strategy. Switching at $cnt = 13$ requires additional 12 requests in the hash join phase. The total number of requests $13 + 12$ is lower than probing all tuple from Ω_1 in the bind join phase.

In the motivating example $|\Omega_1| = 756$ and, therefore, the PBJ would make the right decision.

Correctness of the PBJ4 We show that the Polymorphic Bind Join operator produces correct solution mappings according to the SPARQL set semantics [25].

Theorem 5.1. Given an RDF graph G, an LDF interface c with $ep(c) = G$, a conjunctive SPARQL expression $P = P_1$ And P_2. The Polymorphic Bind Join operators yields the correct set of solution mappings for a query plan $T = T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2)$, that is

$$[\![P]\!]_G = eval(T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2))$$

The intuition of the operator's correctness is as follows. If the operator does not switch its join strategy, it operates as a regular bind join operator producing correct results. In the case that it switches from the bind join strategy to the hash join strategy, there are still tuples from $eval(T_1)$ that have not been considered in the join. These remaining tuples will be processed in the hash join phase (Line 18) and inserted into the hash table H_1. The remaining results of the join are produced in the hash join phase: (i) either when the tuples are probed in the hash table H_2 of $eval(T_2)$ (Line 21), or (ii) when the tuples from $eval(T_2)$ are probed against the tuples in H_1 (Line 25). In order to avoid spurious duplicates, the tuples of $eval(T_1)$ that are probed during the bind join phase are not considered again in the hash join phase. The formal proof of Theorem 5.1 is presented in the following.

Proof. We demonstrate the correctness of the operator $\bowtie_{\mathsf{PBJ}}$ by proving completeness and correctness. We prove these by contradiction in the following.

Completeness: We assume that the $\bowtie_{\mathsf{PBJ}}$ produces incomplete result sets:

$$eval(T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2)) \subset [\![P]\!]_G$$

Let us consider a solution mapping μ, such that $\mu \in [\![P]\!]_G$ and $\mu \notin eval(T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2))$. Without loss of generality, assume that $\mu = \mu_i^1 \cup \mu_j^2$, with $\mu_i^1 \sim \mu_j^2$, $\mu_i^1 \in eval(T(P_1))$ and $\mu_j^2 \in [\![P_2]\!]_{ep(c)}$. Assuming the evaluation of $T(P_1)$ is correct by Proposition 4.1, we distinguish two sub-cases:

Case I: μ_i^1 is processed by the PBJ during the bind join phase (Line 4 to 11). The operator computes $\mu_i^1 \cup \mu_x^2, \forall \mu_x^2 \in eval((\mu_i^1(P_2), c))$ (Line 6). By Def. 4.4, we have that $\mu_x^2 \in [\![\mu_i^1(P_2)]\!]_{ep(c)}$. Since the solution mappings in $[\![\mu_i^1(P_2)]\!]_{ep(c)}$ correspond to the subset of solution mappings in $[\![P_2]\!]_{ep(c)}$ which are compatible with μ_i^1, μ must be produced by the PBJ.

CASE II: μ_i^1 is processed by the PBJ during the hash join phase (Line 17 to 25). In this case, μ_i^1 is inserted into the hash table H_1 and probed with all solution mappings in H_2. If μ_j^2 is in H_2, then μ will be produced by the operator (Line 21). If μ_j^2 is not yet in H_2, it will be processed by the operator (Line 22) as part of $eval(T(P_2))$ and probed in H_1 (where μ_i^1 has been inserted) and the solution mapping μ is produced.

In both cases, the solution mapping μ is produced by the PBJ. This contradicts the assumption that $\mu \notin eval(T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2))$.

Soundness: We assume that the $\bowtie_{\mathsf{PBJ}}$ produces unsound results:

$$eval(T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2)) \supset [\![P]\!]_G$$

To this end, we consider a solution mapping μ, such that $\mu \in eval(T(P_1) \bowtie_{\mathsf{PBJ}} T(P_2))$ and $\mu \notin [\![P]\!]_G$. Without loss of generality, assume that $\mu = \mu_i^1 \cup \mu_j^2$, with $\mu_i^1 \sim \mu_j^2$, $\mu_i^1 \in eval(T(P_1))$ and $\mu_j^2 \notin eval(T(P_1))$. Assuming the evaluation of $T(P_1)$ is correct by Proposition 4.1, we have that $\mu_i^1 \in [\![P_1]\!]_G$. As a result, μ_j^2 is the cause of the unsoundness of μ. Note that μ_j^2 must be produced by an access operator for $T(P_2)$ in either the bind join or the hash join phase. If μ_j^2 is produced by the access operator during the bind join phase, we have by Def. 4.4 that $\mu_j^2 \in [\![\mu_i^1(P_2)]\!]_G$ (Line 5). Which implies that $\mu_j^2 \in [\![P_2]\!]_G$, as $[\![\mu_i^1(P_2)]\!]_G$ is the subset of solution mappings from $[\![P_2]\!]_G$ that are compatible with μ_i^1. Then μ must belong to $[\![P]\!]_G$ which contradicts the assumption. Otherwise, μ_j^2 is produced during the hash join phase by $eval(T(P_2))$ (Line 15). By Def. 4.4 this means that $\mu_j^2 \in [\![P_2]\!]_G$. Then μ must belong to $[\![P]\!]_G$ which contradicts the assumption. □

5.2. Polymorphic Hash Join2

We now introduce the Polymorphic Hash Join (PHJ), which is the complement of the PBJ. The PHJ enables adaptivity by switching its join strategy from a hash join to a bind join during query execution. We denote the PHJ by $\bowtie_{\mathsf{PHJ}}$ and it can be placed in a query plan $T = T_1 \bowtie_{\mathsf{PHJ}} T_2$ when T_2 is an access operator in the plan. The core idea of the operator is that it decides to switch the join strategy after receiving the last tuple from the sub-plan T_1. At this point, the operator estimates whether probing all tuples received from T_1 would require fewer requests than continuing with the hash join by obtaining the remaining tuples from T_2.

The PHJ operator is outlined in Algorithm 3. In the first phase, the operator follows a non-blocking, symmetric hash join strategy as long as the operator receives tuples from T_1 (Line 8 to 23). During the hash join phase, the operator builds two hash tables H_1 and H_2 by inserting the tuples obtained from T_1 and T_2. Tuples obtained from T_1 are then probed in H_2 and vice versa. Moreover, the operator keeps track of the set of solution mapping O that it produces (Line 13 and 21). As soon as the operator receives the `EOF` tuple from T_1 which indicates that all tuples from $eval(T_1)$ have been received and processed, the operator determines whether it is convenient to switch the join strategy. The decision to switch is determined by the `switch` function. The function determines the number of remaining requests with the hash join strategy and compares it to an estimation of the requests for probing all tuples received from T_1 using a bind join. The number of remaining hash join requests acc_Δ is determined by the total number of requests for obtaining the tuples from T_2 and subtracting the number of requests which have already been performed. The operator then estimates whether instantiating all tuples received from T_1 and requesting the resulting expressions would require fewer requests than to continue with the hash join:

$$\mathtt{switch}_{\mathsf{PHJ}}(cnt_1, cnt_2, T) = \begin{cases} \mathtt{True} & \varepsilon \cdot cnt_1 < acc_\Delta(cnt_2, T_2), \\ \mathtt{False} & \text{otherwise.} \end{cases}$$

The parameter $\varepsilon \in (0, \infty)$ allows for weighting the bind join requests and we will show that we can guarantee that the PBJ minimizes the total number of requests during its execution if the parameter ε is set correctly. For a TPF server c with page size p_c, the remaining number of hash join requests is determined as

$$acc_\Delta(cnt_2, T_2) = \underbrace{acc(T_2)}_{\text{Total requests for } T_2} - \underbrace{\left\lceil \frac{cnt_2}{p_c} \right\rceil}_{\text{Performed requests}} .$$

In the case that the PHJ decides to switch to the bind join, it terminates the hash join phase (Line 15). Thereafter, in the bind join phase, the operator probes all tuples from $eval(T_1)$ as they have been inserted in the hash table H_1 (Line 25 to 28). The operator does not know whether there are remaining tuples from $eval(T_2)$ that are compatible with a solution mapping

Algorithm 3: Polymorphic Hash Join

```
   Input: Plan T_1, access operator T_2 = A_2 = (SE_2, c_2)
1  O ← ∅
2  H_1 ← HashTable()
3  H_2 ← HashTable()
4  cnt_1 ← 0, cnt_2 ← 0
5  μ' ← null, μ'' ← null
6  Ω_1 ← eval(T_1) = {μ^1_1, ..., μ^1_k, EOF}
7  Ω_2 ← eval(T_2) = {μ^2_1, ..., μ^2_l, EOF}
8  while μ' ≠ EOF ∧ μ'' ≠ EOF do
9      μ' ← receive Ω_1.next() /* Asynchronous */
10     cnt_1 ← cnt_1 + 1
11     H_1.insert(μ')
12     for μ ∈ H_2.probe(μ') do
13         O ← O ∪ {μ ∪ μ' | μ ~ μ'}
14         output {μ ∪ μ' | μ ~ μ'}
15     if μ' = EOF ∧ switch_PHJ(cnt_1, cnt_2, T) then
16         break
17     μ'' ← receive Ω_2.next() /* Asynchronous */
18     cnt_2 ← cnt_2 + 1
19     H_2.insert(μ'')
20     for μ ∈ H_1.probe(μ'') do
21         O ← O ∪ {μ ∪ μ'' | μ ~ μ''}
22         output {μ ∪ μ'' | μ ~ μ''}
23
   // Switch to bind join strategy
24 if μ'' ≠ EOF then
25     for μ' in H_1 do
26         for μ ∈ eval((μ'(SE_2), c_2)) do
27             if {μ' ∪ μ} ∉ O then
28                 output {μ' ∪ μ | μ ~ μ'}
29 output EOF
```

from $eval(T_1)$, which has not been inserted into H_2 during the hash join phase. Therefore, it probes all tuples obtained from $eval(T_1)$ in the inner plan T_2 (Line 26) during the bind join phase. Following set semantics, the operator does not produce duplicate tuples as it determines whether a tuple has been produced already (Line 27) before producing it.

Optimality Condition for the PHJ4 For the PHJ there is an optimal parameter ε^*, for which the operator minimizes the total number of requests during its execution.

Proposition 5.1. For a query plan $T = T_1 \bowtie_{\mathsf{PHJ}} T_2$ with $T_2 = A_2 = (SE_2, c)$, the Polymorphic Hash Join operator minimizes the total number of request if the parameter ε is set to ε^* with

$$\varepsilon^* = \frac{1}{|eval(T_1)|} \sum_{\mu \in eval(T_1)} acc((\mu(SE_2), c)).$$

In other words, if ε^* is equal to the average number of requests to be performed for probing a tuple from $eval(T_1)$ in T_2, the `switch` function in the operator will evaluate to `True` only if switching requires fewer requests. The proposition follows from the following observations. First, we decide whether to change the strategy once we have received all tuples from the subplan P_1 and therefore, we have that $cnt_1 = |eval(T_1)|$ and consequently, the left side in the decision rule of the `switch` function is $\varepsilon \cdot |eval(T_1)|$. Therefore, the `switch` function evaluates to `True` with ε^*, if we have

$$\underbrace{\sum_{\mu \in eval(T_1)} acc((\mu(SE_2, c_2))}_{\text{Bind join requests}} < \underbrace{acc_\Delta(cnt_2, T_2)}_{\text{Remaining hash join requests}} .$$

The left expression is the sum of requests necessary when probing each individual solution mapping form $eval(T_1)$ in the inner-plan T_2. For each solution mapping μ, the number of requests is given according to the access request cost acc as defined in Equation (1). The right side of the expression, $acc_\Delta(cnt_2, T_2)$, is the number of remaining requests to obtain all solution mappings T_2 according to the current hash join strategy. Consequently, if ε is set accurately to the average number of requests for probing a tuple in the bind join, the operator always chooses the request-minimizing strategy. In the case that $\varepsilon \neq \varepsilon^*$, we cannot guarantee the minimum number of requests:

- $\varepsilon > \varepsilon^*$: The number of requests for probing all solution mappings is overestimated and the operator performs more requests by not switching to the bind join strategy.
- $\varepsilon < \varepsilon^*$: The number of requests for probing all solution mappings is underestimated and the operator switches to the bind join strategy, which ends up requiring more requests.

If we assume that all requests are equal in terms of response time, we have that $\varepsilon \geqslant 1$ because probing a solution mapping requires at least one request. However, different types of requests may yield different response times [27] and, therefore, values for ε below 1 may also be feasible. Determining ε^* requires knowing the number of requests for probing each solution mapping from T_1 which is unlikely to be known in practice. Nonetheless, when setting the parameter ε it should be still considered that (i) depending on the expression SE_2, probing an instantiation of SE_2 may require several requests and (ii) a request for probing may require less time than a request of the hash join due to fewer intermediate results to be transferred.

Correctness of the PHJ4 We now show that the Polymorphic Hind Join operator produces correct solution mappings according to the SPARQL set semantics [25].

Theorem 5.2. Given an RDF graph G, an LDF interface c with $ep(c) = G$, a conjunctive SPARQL expression $P = P_1$ AND P_2. The Polymorphic Bind Join Operators yields the correct set of solution mappings for a query plan $T = T(P_1) \bowtie_{\text{PHJ}} T(P_2)$, that is

$$[\![P]\!]_G = eval(T(P_1) \bowtie_{\text{PHJ}} T(P_2))$$

The intuition of the operator's correctness is as follows. If the operator does not switch its join strategy, it operates as a regular hash join operator producing a correct result set. The operator considers switching from the hash join strategy to the bind join strategy when all tuples from $eval(T_1)$ have been received and inserted into H_1 (Line 15). The hash join strategy is executed only when there are tuples from $eval(T_2)$ that have not been received yet (Line 24). In this case, all tuples in H_1 are probed in T_2 in the bind join phase to produce all results of the join. To avoid spurious duplicates, the operator checks in the bind join phase if a result has been produced during the hash join phase (Line 27). The proof of Theorem 5.2 is as follows.

Proof. We prove the correctness of the operator $\bowtie_{\text{PHJ}}$ by showing completeness and soundness. We prove these by contradiction in the following.

Completeness: The first case is that the $\bowtie_{\text{PHJ}}$ produces incomplete result sets. We assume that

$$eval(T(P_1) \bowtie_{\text{PHJ}} T(P_2)) \subset [\![P]\!]_G$$

Let us consider a solution mapping μ, such that $\mu \in [\![P]\!]_G$ and $\mu \notin eval(T(P_1) \bowtie_{\text{PHJ}} T(P_2))$. Without loss of generality, assume that $\mu = \mu_i^1 \cup \mu_j^2$, with $\mu_i^1 \sim \mu_j^2$, $\mu_i^1 \in eval(T(P_1))$ and $\mu_j^2 \in [\![P_2]\!]_{ep(c)}$. Assuming the evaluation of $T(P_1)$ is correct by Proposition 4.1, we distinguish two sub-cases:

CASE I: The operator does not switch the operation, i.e., $\texttt{switch}(cnt_1, cnt_2, T)$ never evaluates to `True`. As a result, all solution mappings from $[\![P_1]\!]_G$ and $[\![P_2]\!]_G$ (Def. 4.4) are processed in the hash join phase (Line 8 to 23) and the operator finalizes once both `EOF` tuples have been received. In this case, μ_i^1 is processed from $eval(T(P_1))$ (Line 9), inserted into the hash table H_1, and probed with all solution mappings in H_2. If μ_j^2 is in H_2, then μ will be produced by the operator (Line 14). If μ_j^2 is not yet in H_2, it will be processed by the operator as part of $eval(T(P_2))$ (Line 17) and probed in H_1 (where μ_i^1 has already been inserted) and the solution mapping μ is produced.

CASE II: The operator switches the operation to the bind join strategy. This can only be the case if all solution mappings from $[\![P_1]\!]_G$ have been processed in the hash join phase (Line 15) and if not all tuples from $eval(T(P_2))$ have been processed yet: $\mu' = \texttt{EOF}$ and $\mu'' \neq \texttt{EOF}$. As a consequence, all solution mappings from $[\![P_1]\!]_G$ have been inserted into H_1 and probed in H_2. If μ_j^2 was in H_2 when μ_i^1 was probed in H_2, then μ will be produced by the operator (Line 14). In addition, all solution mappings produced during the hash join phase are added to the set O (Line 13 and 21). In the bind join phase (Line 25 to Line 28), the operator processes all tuples in H_1 including μ_i^1 and produces the solution mappings that it has not yet produced: $\mu_i^1 \cup \mu_x^2, \forall \mu_x^2 \in eval((\mu_i^1(P_2), c)) \wedge \mu_i^1 \cup \mu_x^2 \notin O$. By Def. 4.4, we have that $\mu_x^2 \in [\![(\mu_i^1(P_2)]\!]_{ep(c)}$ and the solution mappings in $[\![\mu_i^1(P_2)]\!]_{ep(c)}$ correspond to the subset of solution mappings in $[\![P_2]\!]_{ep(c)}$ which are compatible with μ_i^1. Since $\mu = \mu_i^1 \cup \mu_j^2$ has not been produced yet, we have $\mu \notin O$. As a result, μ must be produced by the PHJ.

In both cases, the solution mapping μ is produced by the PHJ. This contradicts the assumption that $\mu \notin eval(T(P_1) \bowtie_{\text{PHJ}} T(P_2))$.

Soundness: The second case is that the $\bowtie_{\text{PHJ}}$ produces unsound results. We assume that

$$eval(T(P_1) \bowtie_{\text{PHJ}} T(P_2)) \supset [\![P]\!]_G$$

This means there is a solution mapping μ, such that $\mu \in eval(T(P_1) \bowtie_{\text{PHJ}} T(P_2))$ and $\mu \notin [\![P]\!]_G$. Without loss of generality, assume that $\mu = \mu_i^1 \cup \mu_j^2$, with $\mu_i^1 \sim \mu_j^2$, $\mu_i^1 \in eval(T(P_1))$ and $\mu_j^2 \notin eval(T(P_1))$. Since the operator processes the solution mappings in $[\![P_1]\!]_G$ twice in the case that it switches its operation, there are two cases in which the PHJ might produce unsound results: (1) μ_j^2 is the cause of unsoundness; or (2) μ is a spurious duplicate produced by the PHJ.

The first case is that μ_j^2 is the cause of the unsoundness of μ. Note that μ_j^2 must be produced by an access operator for $T(P_2)$. The proof that μ_j^2 is not produced by an access operator during both the hash and the bind join phase is analogous to the soundness proof of the PBJ for Theorem 5.1.

The second case is that μ is produced twice (i.e., a spurious duplicate) when the operator processes μ_i^1 from H_1 a second time in the bind join phase. Furthermore, we assume that μ is produced during the hash join phase and a second time by the `output` of the operator in Line 28. In this case, we have that μ_i^1 in H_1 and the operator obtains the solution mappings $\mu_i^1 \cup \mu_x^2, \forall \mu_x^2 \in eval((\mu_i^1(P_2), c))$ (Line 26). We assume that $\mu_i^1 \cup \mu_j^2$ is in this set of solution mappings. If $\mu_i^1 \cup \mu_j^2$ has already been produced during the hash join phase, we have that $\mu_i^1 \cup \mu_j^2 \in O$ (Line 13 and 21). As the operator does not produce solution mappings which are in O (Line 27), we have that $\mu_i^1 \cup \mu_j^2$ is not produced a second time. As a result, μ is not produced twice, which contradicts the assumption. □

5.3. Summary2

We introduced a new class of adaptive join operators that are able to switch their join strategy during the execution of a query plan. In particular, we presented two instances from this class: the Polymorphic Bind Join (PBJ) and the Polymorphic Hash Join (PHJ). These operators allow for switching their join strategy from a bind join to a hash join and vice versa. While we presented PBJ and PHJ operators probe a single tuple in the bind join phase, both operators can easily be extended to handle several bindings per request (binding block size > 1). Moreover, the proposed operators can be further optimized by leveraging the query planner and the properties of the LDF interface. The query planner could determine the sensitivity of individual operators according to estimated cardinalities and probabilities of estimation errors. In addition, the operators could leverage the properties of the underlying LDF interface. For example, in the case that the solution mappings are sorted (e.g., for a TPF server with an HDT backend), the operators could use the order to reduce the number of requests after switching the join strategy. In addition, the PHJ could be extended in two ways. First, instead of keeping all produced tuples in the set O to avoid producing duplicate solution mappings, the operator could also check during the bind join phase, whether the tuples are in the hash table H_2. If this is the case, they have been processed in the hash join phase already and the corresponding solution mapping has been produced[8]. Second, the PHJ could track response times and estimate the number of requests for probing each tuple during its execution to properly set the ε parameter. In this work, we do not consider these optimizations in the implementation of the operators as we aim to evaluate the concept of switching the join strategy in a more general scenario and for a variety of query planning approaches.

Table 1

Properties of the benchmark queries.

		#	# Answers			# Triple Patterns	
			min	max	median	min	max
WatDiv	C	3	2	417 029	20	6	10
	F	25	0	58	8	6	9
	L	25	0	239	47	2	3
	S	35	0	320	4	3	9
nLDE	BM 1	20	2	45 655 205	7131	4	14
	BM 2	25	2	1768	19	3	5

6. Evaluation1

In this experimental evaluation, we investigate the effectiveness of our two approaches for robust query processing over Linked Data Fragments. As previously mentioned, we evaluate the approaches using the example of Triple Pattern Fragment (TPF) servers. We investigate the effectiveness of the proposed query planning approach CROP and the impact of the adaptive join operators PBJ and PHJ on two different benchmarks with a fine-grained analysis of the results.

6.1. Experimental Setup2

Datasets and Queries4 As the basis for our evaluation, we focus on two different datasets and corresponding benchmark queries that have also been used in previous evaluation for LDF clients [1–6]. First, we use a synthetic RDF graph and benchmark queries from the Waterloo SPARQL Diversity Test Suite (WatDiv) [30]. Specifically, 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

[8]We choose to describe the operator with the set O to provide a more intuitive description and proof of correctness.

from query categories, namely, 25 linear queries (L), 35 star queries (S), 25 snowflake-shaped queries (F) and 3 complex queries (C). The second benchmark is taken from the evaluation of nLDE[9] and is based on the real-world RDF graph of DBpedia 2014 [5]. The benchmark consists of two subsets of queries. The first subset, Benchmark 1 (BM 1), consists of 20 non-selective queries composed of basic graph patterns with up to 14 triple patterns that produce a large number of intermediate results. The second subset, Benchmark 2 (BM 2), consists of 25 more selective queries from the domains *Historical*, *Life Science*, *Movies*, *Music*, and *Sports*. The number of queries per category as well as statistics on the number of answers and number of triple patterns for the queries of benchmarks is provided in Table 1. In addition, to showcase the benefits of combining the cost model with robustness in the query plan optimizer on different RDF graphs, we designed an additional test-set with 10 queries for 3 RDF graphs that include either a `s-o` and or an `o-o` join and 3-4 triple patterns. We used the RDF graphs DBpedia 2014, GeoNames 2012, and DBLP 2017 for which we obtained the HDT files from the RDF-HDT website[10].

Implementation4 We implemented CROP[11] on top of the nLDE client[4], which is implemented in Python 2.7.13. We implemented our cost model, robustness measure, and query plan optimizer, such that the resulting physical query plans can be executed using nLDE. We used the default of 2 eddy operators and do not consider the routing adaptivity features of nLDE, that is, we select no routing policy for the execution. We deployed the TPF server with an HDT backend [31] using the `Server.js` v2.2.3[12] implementation. We set the number of workers for the TPF server to 5. All experiments were executed on a Debian Jessie 64 bit machine with CPU: 2x Intel(R) Xeon(R) CPU E5-2670 2.60GHz (16 physical cores), and 256GB RAM. The queries were executed three times in all experiments. The timeout was set to 900 seconds for the experiments to set the parameters for CROP (δ, k, γ, and ρ). For all the remaining experiments, we set a runtime timeout of 600 seconds.

Evaluation Metrics4 We consider the following query execution metrics:

(i) *Runtime*: Elapsed time in seconds spent by a query engine to complete the evaluation of a query measured in seconds. In the implementation of CROP, we distinguish between the *optimization time* spent by the query planner to obtain a query plan and the *execution time* to execute the plan.
(ii) *Number of Requests*: Total number of requests submitted to the server during the query execution including query planning.
(iii) *Number of Answers*: Total number of answers produced during query execution.
(iv) *Diefficiency*: Continuous efficiency as the answers are produced over time [32].

We provide all results of our experimental study in our supplemental material on Zenodo[13].

6.2. Robust Query Planning2

We start by evaluating the proposed cost model, robustness measure, and corresponding query planner. First, we focus on the height discount factor for the cost model and the block size k for the IDP algorithm. Second, we focus on the cost and robustness threshold values of the query planner. Finally, we compare the resulting parameterization for CROP to the state-of-the-art clients for TPF servers. In order to determine how well the parameter settings generalize, we randomly selected a subset of the queries from the benchmarks to set the parameters but compare our approach to the state of the art for all benchmark queries. From the 88 queries in the WatDiv Benchmark, we randomly selected 2 queries per subgroup (L1-L5, F1-F5, S1-S7, C) resulting in a total of 36 queries. For the nLDE benchmark, we chose 10 queries from BM 1 at random and 2 queries per domain from BM 2, resulting in a total of 20 queries.

Cost Model and IDP Parameters4 We start by investigating how the parameter settings of the cost model affect the efficiency and structure of the query plans obtained by the planner. We begin with the parameter δ which determines the impact of the height discount factor (Eq. (3)). We consider the following settings: $\delta \in \{0, 1, 2, 3, 4, 5, 6, 7\}$. As we only focus on local deployment in our evaluation, we do not investigate different processing cost parameters and set $\phi = 0.001$. Optimizing the value for ϕ should be considered when in different deployment scenarios, where network delays have a stronger impact on the query execution cost.

[9]https://people.aifb.kit.edu/mac/nlde/
[10]http://www.rdfhdt.org/datasets/
[11]https://github.com/Lars-H/crop
[12]https://github.com/LinkedDataFragments/Server.js
[13]https://doi.org/10.5281/zenodo.4639843

Table 2

Cost-Model: Impact of the δ-parameter of the height discount factor for the Bind Join (BJ) on the query plan efficiency (runtime and requests), percentage of BJs and bushy plans. Best efficiency values are indicate in bold.

	WatDiv				nLDE BM			
δ	Runtime (s)	Requests	BJs	Bushy Plans	Runtime (s)	Requests	BJs	Bushy Plans
0	255.0	**24 039**	50 %	2 %	1300.75	**24 702**	34 %	10 %
1	267.01	30 133	61 %	2 %	588.74	24 743	39 %	10 %
2	224.6	30 708	66 %	0 %	330.99	27 184	44 %	10 %
3	222.15	29 907	72 %	0 %	221.27	36 154	54 %	10 %
4	221.26	29 587	74 %	0 %	**182.86**	36 497	59 %	10 %
5	236.55	32 933	74 %	2 %	196.83	40 082	67 %	5 %
6	215.22	32 659	73 %	5 %	200.95	42 587	71 %	5 %
7	**211.72**	32 651	74 %	5 %	202.02	42 586	71 %	5 %

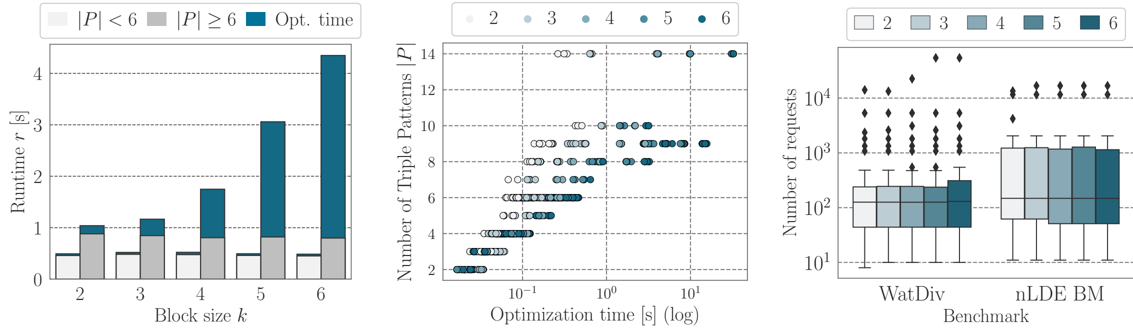
We disable robust plan selection by setting $\rho = 0.00$, set the default block size to $k = 3$, and select the top $t = 5$ plans. Table 2 provides an overview of the query execution performance regarding the mean runtime and the mean number of requests for both benchmarks per run. In addition, to determine the impact of the parameter on the structural properties of the query plans, we report the percentage of Bind Joins (BJ) in the query plans and the percentage of bushy query plans.

Regarding the efficiency of the query plans, we observe that an increase in the δ values yields a reduction of query runtimes. In particular, the lowest runtime for the WatDiv benchmark are observed for $\delta = 7$ and for the nlDE benchmark with $\delta = 4$. For the nLDE benchmark, the runtimes slightly increase with $\delta > 4$. In contrast to the runtimes, the lowest number of requests in both benchmarks are observed without the height discount factor (i.e., $\delta = 0$). Combining this observation with the percentage of BJ operators, the results show that (i) an increasing δ leads the query planner to place more BJs, (ii) more BJs yield a higher number of requests for probing the tuples, and (iii) the discrepancy between runtime and number of requests arises from the fact the requests from the BJ operator typically yield lower response times in comparison to HJ requests, as more variables are instantiated in the triple pattern which leads to fewer intermediate results that need to be transferred. While the request costs of the cost model do not distinguish the type of requests, the height discount factor allows for balancing this effect.

Even though we can observe a similar impact of the δ value on the percentage of BJs in the query plans, the shape of the query plans differs in the two benchmarks. When considering the percentage of bushy plans, we find that in the WatDiv benchmark fewer bushy plans are obtained and the best results in the nLDE benchmark are observed with a higher bushy plan ratio. These results show that, on the one hand, the δ parameter not only affects the type of join operators placed in the plan but also their shape. Based on the previous observations, we set $\delta = 4$ as it yields an appropriate trade-off between runtime efficiency and the number of requests for both benchmarks.

Next, we focus on the parameterization of the IDP algorithm. Specifically, investigate how the block size k impacts both times for the planner to devise these plans (optimization time) and the efficiency of their execution (execution time and the number of requests). We set the number of top plans kept by the IDP to $t = 5$ and set the height discount parameter $\delta = 4$ as suggested by the previous results. We study the block sizes $k \in \{2, 3, 4, 5, 6\}$. Figure 3a shows the median runtimes $\widetilde{r}$ for all queries per block size k with the proportion of the optimization time indicated in blue. The runtimes are separated for small queries ($|P| < 6$) and larger queries ($|P| \geqslant 6$). Furthermore, note that $k = \min\{k, |P|\}$.

The results show that for small queries ($|P| < 6$) the overall runtimes as well as the optimization time proportion, is similar regardless of the block size k. For larger queries ($|P| \geqslant 6$), the runtimes increase with higher k. This increase for larger queries is due to a larger proportion of optimization time to obtaining *ideally* better plans. At the same time, the benefit of investing the additional time and exploring alternative query plans is disproportionately low as the execution times only marginally decrease. Considering just the execution times, we find that the mean lowest execution times are obtained for Watdiv with $k = 2$ (5.41 s) with very similar results for the second-lowest with $k = 4$ (5.79 s). For the nLDE Benchmark, the lowest mean execution times are observed for $k = 3$ (8.98 s).



(a) Share of optimization time on runtime. (b) Optimization time by query size $|P|$. (c) Number of request per benchmark.

Fig. 3. IDP: Impact of the different block sizes in the IDP algorithm on the execution time, optimization time, and number of requests.

Table 3

Mean Runtime, mean number of requests, and the number of robust plans ($|R^*|$) selected by the query plan optimizer for both benchmarks. Best overall runtime and minimum number of requests per configuration and benchmark are indicated in bold.

WatDiv Benchmark

	$\rho = 0.05$			$\rho = 0.10$			$\rho = 0.15$			$\rho = 0.20$			$\rho = 0.25$		
γ	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$
0.1	87.88	**20 796**	6	89.04	**20 796**	6	96.47	21 504	6	97.31	21 643	6	338.30	26 395	8
0.3	**87.79**	**20 796**	6	89.68	**20 796**	6	194.62	30 078	2	196.18	30 217	2	196.45	30 217	2
0.5	195.14	30 078	2	195.15	30 078	2	196.77	30 078	2	197.63	30 217	2	197.45	30 217	2
0.7	194.41	30 078	2	193.61	30 078	2	196.47	30 078	2	194.97	30 085	0	195.29	30 085	0
0.9	195.19	30 078	2	192.97	30 078	2	194.84	30 078	2	195.77	30 085	0	194.49	30 085	0

nLDE Benchmark

	$\rho = 0.05$			$\rho = 0.10$			$\rho = 0.15$			$\rho = 0.20$			$\rho = 0.25$		
γ	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$	Runtime	Requests	$\|R^*\|$
0.1	291.45	**31 465**	1	303.14	49 925	2	303.95	49 925	2	304.31	49 925	2	384.13	58 092	3
0.3	293.19	**31 465**	1	303.29	49 925	2	301.46	49 925	2	299.85	49 925	2	302.20	49 925	2
0.5	290.03	**31 465**	1	196.19	35 783	1	196.01	35 783	1	196.82	35 783	1	**195.37**	35 783	1
0.7	290.84	**31 465**	1	196.67	35 783	1	196.28	35 783	1	198.50	35 783	1	196.26	35 783	1
0.9	289.72	**31 465**	1	195.78	35 783	1	197.67	35 783	1	196.06	35 783	1	197.13	35 783	1

The optimization times (log-scale) with respect to the query size are shown in Figure 3b, where the block size is indicated in color. The results show the impact of the number of triple patterns in the query on the optimization time. Especially, for large block sizes $k > 4$ and queries with many triple patterns this trend is more apparent. For example, for query Q07 with 14 triple pattern, the optimization time is on average more than 100 times higher with $k = 6$ (31.85 s) than with block size $k = 2$ (0.30 s). This is due to the fact that the number of plans that need to be considered grows exponentially with k. The observed impact of the block size on the query optimization time is in line with the results reported by Kossmann and Stocker [29].

Finally, we look at the distribution of the number of requests for the different benchmark and block sizes as shown in Figure 3c. We can observe that for the WatDiv benchmark, the number of outliers increases, especially with $k > 4$,. For the nLDE benchmark, the number of requests can be slightly reduced for some queries with $k > 4$. Based on all previous observations, we set k in a dynamic fashion with: $k = 4$ if $|P| < 6$ and $k = 2$ otherwise. As a result, the planner avoids disproportionate optimization times, especially for large queries, while

still exploring the space of possible plans sufficiently to find efficient alternative query plans.

CROP Query Optimizer Parameters4 The previous experiments focused on determining appropriate parameters for the cost model. We now focus on the parameters for the proposed query optimizer (cf. Algorithm 1), namely the robustness threshold ρ and the cost threshold γ. The robustness threshold ρ defines whether an alternative plan should be considered. The cost threshold γ limits the alternative plans to those which are not considered too expensive with respect to the cheapest plan. We tested all 25 combinations of $\rho \in \{0.05, 0.10, 0.15, 0.20, 0.25\}$ and $\gamma \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. We executed the subset of 56 queries (36 WatDiv and 20 nLDE BM) for each combination three times. Table 3 shows the mean results per benchmark for all runs. The results include the query plan efficiency in terms of runtime and number of requests, as well as the number of queries for which a robust query plan was selected over the cheapest plan as $|R^*|$. The best results are indicated in bold.

For the WatDiv benchmark, the results show that the best performing query plans are obtained for parameter values $\rho \in \{0.05, 0.10\}$ and $\gamma \in \{0.1, 0.3\}$. While the mean runtimes slightly differ, the number of requests indicates that the same query plans are chosen by the planner. Moreover, in 6 out of the 36 queries an alternative robust plan is chosen. In contrast, for the nLDE Benchmark, the results show the best runtimes with $\rho \in [0.10, 0.25]$ and $\gamma \in [0.5, 0.9]$. The query plans that minimize the number of requests are obtained with $\rho = 0.05$ regardless of the cost threshold. The difference can be explained by the single query for which the robust query plan is chosen. The best runtimes are obtained in the case that the robust plan is selected for Q02 (*Movies*) and the minimum number of requests if the robust plan is selected for Q05 (BM 1). These results are in line with the previous observation regarding the height discount factor, i.e. minimizing the number of requests not always yields the best execution times. The results also show that for the nLDE benchmark, robust plans are selected less frequently than for the WatDiv benchmark. In both benchmarks with $\gamma = 0.1$, we observe the impact of an increasing robustness threshold as more alternative robust plans are chosen ($|R^*|$). Furthermore, the results indicate the effectiveness of the cost threshold in limiting the selection of alternative plans. Combining the results from both benchmarks, we choose a trade-off setting for the parameters and

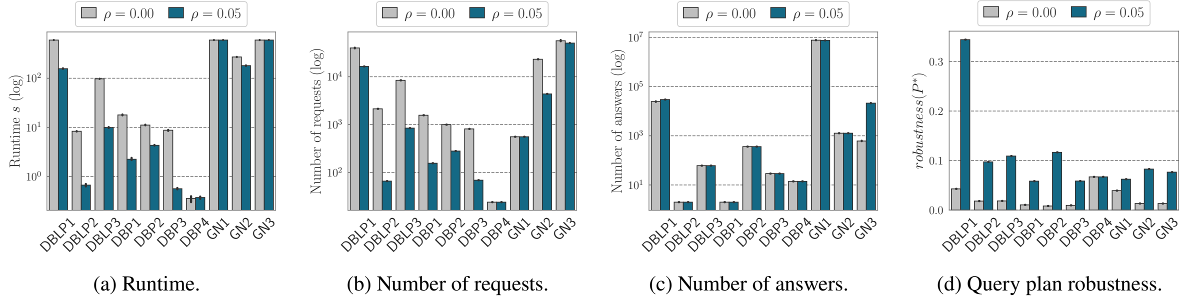
Table 4
Comparison to the state of the art.

		$\bar{r}$	$\tilde{r}$	$\overline{req.}$	$\sum ans.$	Timeout [%]
WatDiv	CROP	**2.12**	**0.68**	**375**	**419 448**	0
	nLDE	9.01	0.84	862	**419 448**	0
	Comunica	6.98	2.76	1572	**419 448**	0
nLDE BM	CROP	**58.14**	0.89	4520	2 969 457	6
	nLDE	77.15	**0.77**	**3897**	**6 616 570**	9
	Comunica	84.28	4.67	9234	423 904	10

set the cost threshold to $\gamma = 0.3$ and the robustness threshold to $\rho = 0.05$ in the following experiments.

Next, we study the effectiveness of our query planning approach in identifying efficient alternative robust plans using the 10 queries from our custom test-set. We keep the parameters from the previous experimental evaluations and compare the configuration of our planner with $\rho = 0.00$ (the cheapest plan is always chosen) to the configuration that selects robust plans with $\rho = 0.05$ (as suggested by the previous experiments). The results are shown in Figure 4. It can be observed that for 7 queries (DBLP1-3, DBP1-3, GN2) the query planner obtains more efficient query plans when enabling the selection of a robust plan. For these queries, the runtime and the total number of requests are lower and, at the same time, the selected robust query plans produce the same number of answers or even more before reaching the timeout. Regarding the remaining queries, we find that for the query DBP4 the planner chooses the same query plan in both configurations as the cheapest plan is also considered to be robust enough (Fig. 4d). For the queries GN1 and GN3, the timeout is reached in all cases. Nonetheless, we can observe that the robust query plans produce more answers with the same number or fewer requests during their execution. The results indicate that CROP allows for devising efficient query plans even for queries where the cost model produces high cardinality estimation errors in the presence of o-o and s-o joins. The robustness values of the cheapest plans, as shown in Fig. 4d, lead the query optimizer to choose more robust plans, which reduces the query execution times as well as the number of requests.

Comparison to the State of the Art4 After the intrinsic evaluation of our approach and determining appropriate parameters for the query optimizer, we now compare CROP to state-of-the-art TPF clients. Specifically, we compare CROP to nLDE [5] and Comunica [6]. In contrast to the previous experiments, we evaluate the



(a) Runtime.

(b) Number of requests.

(c) Number of answers.

(d) Query plan robustness.

Fig. 4. Custom Testset: Runtime, number of requests, number of answers, and query plan robustness for the 10 queries of the custom benchmark. Results compare CROP without robustness ($\rho = 0.00$) and with robustness ($\rho = 0.05$).

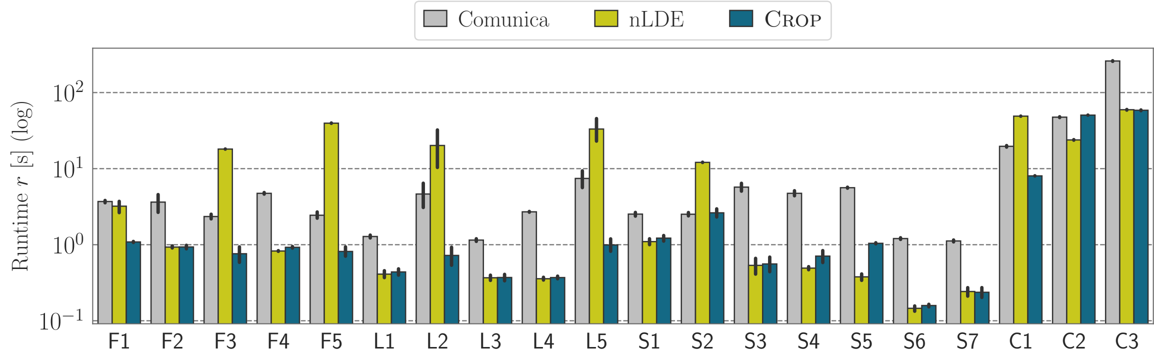
clients using *all* queries from both benchmarks and set the timeout to 600 seconds. Table 4 summarizes the results, where the mean ($\bar{r}$), median ($\widetilde{r}$) runtimes, mean number of requests ($\overline{req.}$), total number of answers ($\sum ans.$), and the percentage of query executions timing out are listed. Moreover, the runtimes of all queries[14] are shown in more detail in Figure 5. In the following, we analyze the results within each benchmark.

Considering the individual runtimes per query group of the WatdDiv benchmark (Fig. 5a), we observe that CROP, in general, exhibits very good performance in terms of runtime in the majority of the cases. Overall, CROP yields the lowest runtimes with the lowest number of requests (Table 4). Combining these observations, the results suggest that the query plans obtained by CROP find a balance between the left-deep plans (i.e., Comunica) and the bushy plans (i.e., nLDE) for the WatDiv benchmark. Finally, none of the clients reach the timeout and consequently, all clients produce complete answers.

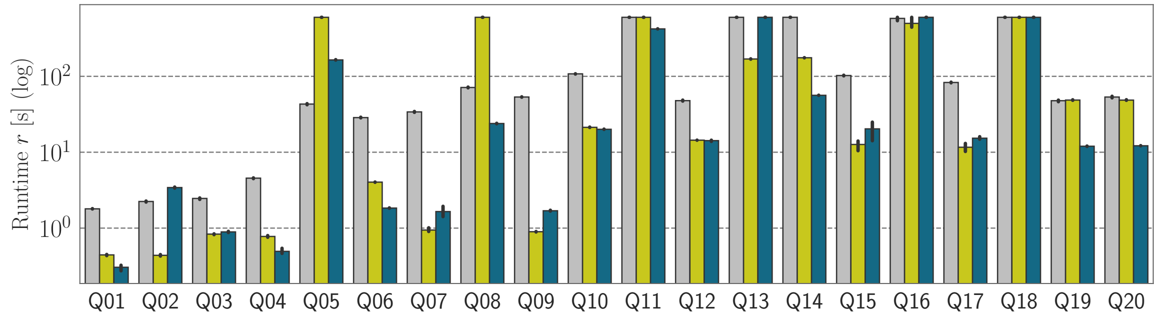
Next, we focus on the nLDE Benchmark, for which the runtimes are detailed in Fig. 5b and Fig. 5c. The runtime results for the nLDE Benchmark 1 reveal a similar performance of CROP as in the WatDiv benchmark. While CROP performs best in 35% of the queries, it also performs well in the remaining queries and is only outperformed by nLDE and Comunica for a single query (Q02). Similar results can be observed for the more selective queries in Benchmark 2, where CROP performs as least as well as the best competitor (nLDE) in the majority of the queries. The previous observation is also reflected in the aggregated results in Table 4. CROP has the lowest mean runtimes. However, nLDE slightly outperforms CROP regarding the median runtimes and the mean number of requests. Moreover, nLDE yields the highest number of answers. Note that, the absolute difference in answers is very high (~3.7M) which is due to the better performance of nLDE in query Q18 which has a total of 45M answers. When considering the number of queries for which all answers are obtained, we find that nLDE obtains complete answers for 90% and CROP 93% of all queries. This is also due to the fact that CROP only reaches the timeout in 6% and nLDE in 9% of query executions. In contrast to the WatDiv benchmark, Comunica is outperformed by nLDE with the highest runtimes and number of requests while obtaining the fewest answers. These results suggest that the bushy query plans of the nLDE query planner are more efficient for the non-selective queries in the nLDE benchmark and out-perform the left-deep plans. Similar to the WatDiv benchmark, the results show that CROP still finds an effective trade-off between these planning paradigms and overall obtains efficient query plans in both benchmarks.

Summarizing the experimental study on our cost- and robustness-based query plan optimizer CROP, we examined how the parameters of the cost model and the planner impact the query plans and compared our approach to state-of-the-art TPF clients. The results show that the height discount factor and the block size of the IDP allow the query planner for obtaining query plans that find a balance of runtime, the number of requests, and optimization time. The cost and robustness thresholds enable the query planner to determine when and which alternative robust plan should be selected, in the case that the cheapest plan is not considered to be robust enough. Finally, comparing CROP to state-of-the-art TPF clients, we found that CROP outperforms the existing approaches in the majority of cases by exploring and devising appropriate query plans without following a fixed heuristic that either builds left-deep plans (Comunica) or bushy plans (nLDE).

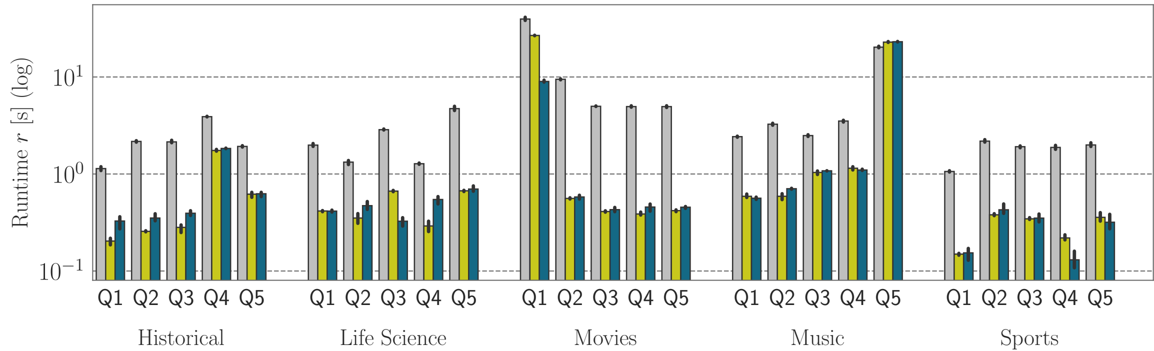
[14]Results are aggregated per query group in WatDiv.



(a) WatDiv Benchmark.



(b) nLDE Benchmark 1.



(c) nLDE Benchmark 2.

Fig. 5. Comparison to the state of the art: Mean query runtimes of Comunica, nLDE, and CROP for the different benchmarks and query categories.

6.3. Polymorphic Join Operators2

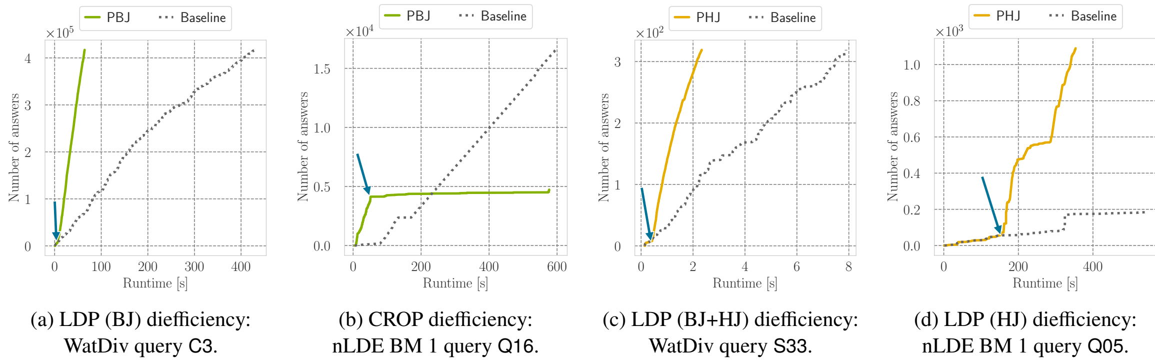
We now evaluate the proposed Polymorphic Join operators and investigate how this novel intra-operator adaptivity affects the query plan executions.

Planning Approaches4 In order to understand the effectiveness of the adaptive PBJ and PHJ, we determine their impact on a variety of query planning approaches. As a result, we are able to examine how different planning methods can benefit from the adaptive operators during the query plan execution. Specifically, we focus on three query planning approaches: a left-deep planner, the nLDE planner, and CROP. Similar to Comunica's *sort* heuristics, the *left-deep planner* (LDP) obtains query plans by sorting the triple patterns by increasing *count* values and builds left-linear plans according to this join order. We consider three different variants of the planning approach, which differ in the join strategies they implement. LDP (HJ) supports hash joins only, LDP (BJ) supports bind joins only, and LDP

Table 5

Overview of the Polymorphic Join Operators for the different benchmarks and planning approaches. Listed are the mean ($\bar{r}$) and median ($\tilde{r}$) runtimes [s], the mean number of requests ($\overline{req.}$) and the sum of answers ($\sum ans.$). Moreover, we indicate the percentage of PBJ and PHJ that switched their strategy during execution by PBJ^{+} and PHJ^{+}, respectively. Best performance values per planner are indicated in bold.

		WatDiv						nLDE BM					
		$\bar{r}$	$\tilde{r}$	$\overline{req.}$	$\sum ans.$	PBJ^{+}	PHJ^{+}	$\bar{r}$	$\tilde{r}$	$\overline{req.}$	$\sum ans.$	PBJ^{+}	PHJ^{+}
LDP (HJ)	Baseline	37.29	15.81	1961	375 163	0 %	0 %	176.46	22.77	2899	2 378 020	0 %	0 %
	PHJ	**10.86**	**1.84**	**651**	16 202	0 %	41 %	**112.92**	**1.6**	**2615**	2 291 301	0 %	22 %
LDP (BJ)	Baseline	12.91	2.95	1859	419 448	0 %	0 %	**80.0**	7.91	9561	1 389 673	0 %	0 %
	PBJ	**2.19**	**0.78**	**298**	419 448	33 %	0 %	87.75	**1.56**	**2683**	2 184 025	66 %	0 %
LDP (BJ+HJ)	Baseline	6.48	1.18	611	419 448	0 %	0 %	62.77	0.87	1478	2 369 838	0 %	0 %
	PBJ	3.95	1.07	395	419 448	6 %	0 %	85.1	0.88	1676	1 785 498	7 %	0 %
	PHJ	4.69	0.71	499	419 448	0 %	16 %	**62.71**	**0.86**	**1477**	2 404 440	0 %	0 %
	PBJ + PHJ	**2.11**	**0.7**	**283**	419 448	6 %	16 %	85.26	0.93	1675	2 421 653	7 %	0 %
nLDE	Baseline	9.01	**0.84**	862	419 448	0 %	0 %	77.15	0.77	3897	6 616 570	0 %	0 %
	PBJ	6.13	0.85	531	419 448	8 %	0 %	104.82	0.91	**3521**	6 272 811	31 %	0 %
	PHJ	8.47	0.86	828	419 448	0 %	6 %	**76.91**	**0.74**	3899	6 574 086	0 %	0 %
	PBJ + PHJ	**5.61**	0.85	**497**	419 448	8 %	6 %	104.81	0.84	3523	6 346 455	31 %	0 %
CROP	Baseline	2.12	0.68	375	419 448	0 %	0 %	58.14	0.89	4520	2 969 457	0 %	0 %
	PBJ	1.99	0.68	272	419 448	5 %	0 %	86.6	1.07	**2381**	1 985 203	45 %	0 %
	PHJ	2.09	**0.64**	373	419 448	0 %	3 %	**58.09**	**0.87**	4515	2 180 385	0 %	0 %
	PBJ + PHJ	**1.97**	0.65	**270**	419 448	5 %	3 %	86.54	1.09	**2381**	2 954 945	45 %	0 %



(a) LDP (BJ) diefficiency: WatDiv query C3.

(b) CROP diefficiency: nLDE BM 1 query Q16.

(c) LDP (BJ+HJ) diefficiency: WatDiv query S33.

(d) LDP (HJ) diefficiency: nLDE BM 1 query Q05.

Fig. 6. Example diefficiency plots with the polymorphic join operators (PBJ / PHJ) and the baseline without polymorphic operators in gray for different planning approaches. Indicated by the blue arrows are the points in the answer traces where the adaptivity impacts the diefficiency.

(BJ+HJ) supports both hash and bind joins. The LDP (BJ+HJ) uses an optimistic join cardinality estimation function (the minimum) and places either a hash join or a bind join based on the resulting estimated number of requests.[15] Additionally, we also use the nLDE planning approach that builds bushy-plans around star-shaped subqueries, and the CROP query planner with the parameter configuration from the previous experiments ($\delta = 4$, dynamic k, $\rho = 0.05$, and $\gamma = 0.3$). In accordance with the previous experiments, we executed all queries from both the WatDiv and nLDE benchmarks three times and set a timeout of 600 seconds. As the baseline, we executed each planning approach without the polymorphic join operators. In addition, (if applicable) we executed the planning approach with either the PBJ or the PHJ enabled as well as with

[15]Similar to Lines 10-15 in Algorithm 1 of [5].

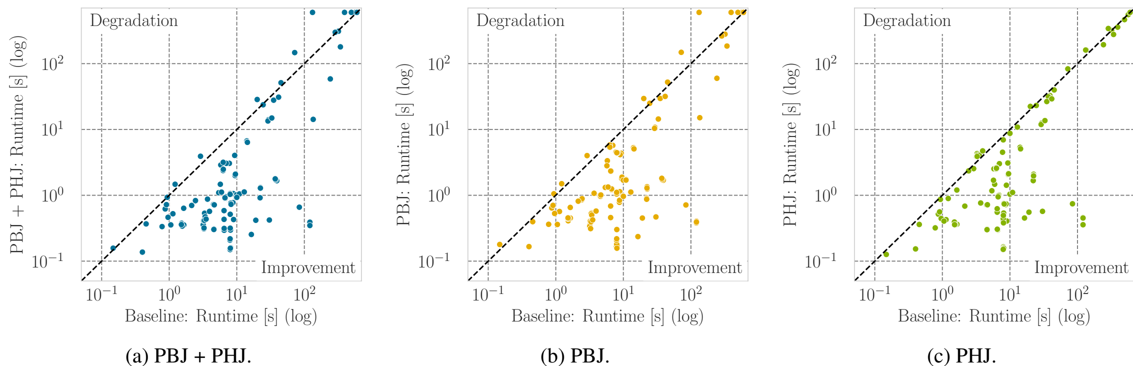
both PBJ + PHJ enabled, resulting in a total of 16 configurations. For the PBJ, we set the λ parameter to $\lambda = 1/\sqrt{height(T_1)}$. Furthermore, we set the parameter to $\varepsilon = 1$ in the PHJ.

Experimental Results4 An overview of the results for all planning approaches per benchmark is provided in Table 5. The table shows the query execution performance according to the mean runtimes, median runtimes, the mean number of requests, and the total answers produced by each approach. Additionally, the percentage of PBJ and PHJ operators that adapted their join strategy during query execution are shown as PBJ^+ and PHJ^+, respectively. Regarding the WatDiv benchmark, the lowest mean runtimes and number of requests are observed when enabling both adaptive join operators (PBJ + PHJ) for all query planning approaches. Only for CROP, the median runtime is slightly lower with just the PHJ enabled. All query planning approaches, except for LDP (HJ), obtain all answers ($\sum ans.$). Intriguingly, LDP (HJ) obtains even fewer results with PHJ enabled. Taking a closer look at the results, we find that these differences are due to the complex query (C3) for which LDP (HJ) reaches the timeout in both configurations. For all other queries, LDP (HJ) obtains all answers without and with the PHJ enabled.

The percentage of operators that adapt their join strategy during query execution (PBJ^+ and PHJ^+) differs for the different planning approaches. It can be observed that the less sophisticated planning approaches, such as the left-deep planner, benefit more from the adaptive join operators than nLDE and CROP. For example, in the query plans of the LDP (BJ+HJ), 6% of the PBJ and 16% of the PHJ operators switch their strategy during the execution, while for CROP only 5% of the PBJ and just 3% of the PHJ switch their strategy. Interestingly, for none of the cases where CROP selects a robust alternative query plan (i.e., R^*), the adaptive join operators adapt their strategy. This indicates that potential cardinality estimation errors that would impact the ideal join strategy were already mitigated by the robust query planner. Overall, the polymorphic join operators improve the runtime efficiency of the query plans for the WatDiv benchmark.

However, considering the results for the nLDE benchmark, the impact of the polymorphic join operators on the query execution differs. For all planning approaches, the results show that enabling the PBJ results in higher mean runtimes. At the same time, fewer answers are produced when enabling the PBJ and PHJ. The only exception is the LDP (BJ) query planner, where enabling the PBJ increases the mean runtime by ~8%, but reduces the median runtime by ~80% and the number of answers produced almost doubles. Comparing the behavior of the PHJ and PBJ for the planning approaches, we find that none of the PHJ operators switch their join strategy, except for the LDP (HJ) planner. As a consequence, the query plans and their execution of the baseline and PHJ configuration are the same and the slight differences in performance are due to runtime conditions (e.g., the exact timing in the execution when the timeout is reached). In summary, only the left-deep query planning approach with just PHJ and just the PBJ operators can benefit from the polymorphic join operators in the nLDE benchmark. For the other query planning approaches, the PBJ switches its strategy inappropriately increasing the query execution performance while the PHJ does not adapt its strategy at all. These results highlight the importance of investigating different benchmarks as the properties of the RDF graphs as well as the queries affect the effectiveness of the proposed operators. Understanding these differences allows to appropriately choose the operators according to the use case and to further improve them in future work.

Next, we investigate the impact of the polymorphic join operators on the continuous production of answers by means of the diefficiency. To this end, we visualize the cumulative number of answers produced with respect to the query runtime for four example queries in Figure 6. The first two queries (Fig. 6a and Fig. 6b) compare the baseline to the configuration with the PBJ enabled. For query C3 from the WatDiv benchmark and the LDP (BJ) planner, we can observe that after obtaining a few hundred tuples from the initial access operator of the query, the PBJ operators switch to the hash join strategy. Only the last PBJ in the query plan does not switch its strategy due to the large number of matching triples for the last triple pattern in the join order: $count($`?v0 :friendOf ?v2`$) = 4\,479\,991$. As a result, the adaptivity of the PBJ in the initial query execution phase allows for producing the answers continuously faster. However, as previously mentioned, the PBJ may also impede the query execution efficiency as shown for the CROP query plan for Q16 in Fig. 6b. In this case, the PBJ operators also switch their join strategy improving the diefficiency at the beginning of the execution. Since the PBJ considers its height in the query, the PBJs located higher in the query plan tend to switch after already probing a larger number of tuples. In the query plan for Q16, the last join operator is more likely to switch its strategy even though it is produc-



(a) PBJ + PHJ. (b) PBJ. (c) PHJ.

Fig. 7. Scatter plots of the runtimes (log-log scale) for all planners with and without adaptive join operators. Value above the diagonal indicate performance degradation with adaptivity and values below performance improvement.

ing answers efficiently in the bind join phase. In this case, switching is not the appropriate strategy as the diefficiency plot shows and the query plan ends up producing fewer answers following the hash join strategy. This is due to the fact that the query execution reaches the timeout. This example illustrates the challenge of determining a feasible approach for switching the join strategy in the PBJ operator.

The second two plots (Fig. 6c and Fig. 6d) compare the baseline to the configuration with the PHJ enabled. While we can observe the potentially detrimental effect of the PBJ in the case that it switches its join strategy too early, the PHJ yields consistent improvements in diefficiency. As shown for example for query S33 in Fig. 6c and query Q05 in Fig. 6d, the PHJ adapts its join strategy appropriately in both cases. The adaptivity substantially improves the diefficiency and reduces the total number of requests of the query plan execution. Due to the higher selectivity of the first join operator in the query plan for S33, the PHJ switches its join strategy rather quickly, while in Q05 the operator switches later during the execution as more tuples are produced by the preceding joins.

As a summary, we compare the runtimes of all planners for each query with and without adaptivity in Figure 7. Each dot represents the runtimes of a query with and without adaptivity for a specific planner. Dots below the diagonal line represent an improvement in the execution performance (lower runtime is better). Overall, for the combination of PBJ and PHJ (Fig. 7a), we observe that, for the majority of queries, the polymorphic join operators allow for improving the robustness of query plan execution by adapting their join strategy. For just the PBJ (Fig. 7b), we observe a few queries where the performance substantially degrades while the majority yield a performance improvement. As expected, for the PHJ (Fig. 7c), we see that the runtime of the majority of queries improves. Just a few queries are slightly above the diagonal line which is likely to be attributed to runtime conditions. Combining the theoretical properties and the empirical evaluation of the PHJ, our findings show that the opportunity of improving the query execution efficiency with PHJ outweighs the potential risk of performance degradation. For the PBJ, we find that (with the current parameters) in many cases there is a higher risk of performance degradation. Therefore, future work should further investigate and improve the PBJ operator and its parameterization to reduce this risk.

7. Conclusion1

In this work, we investigated different robust query processing techniques for Linked Data Fragments. In particular, we proposed two approaches to overcome the challenges of join cardinality estimation errors to devise efficient query planning and execution.

Our first approach, a cost- and robustness-based query plan optimizer (CROP), devises plans that are robust with respect to cardinality estimation errors. We propose a cost model to estimate the cost of query plans that combines local processing and network costs. We assess the robustness of a query plan using the *cost ratio robustness* measure, which compares the query plan's best-case cost to its average-case cost (*RQ 1*). While the best-case cost assumes optimistic cardinality estimations, the average-case cost accounts for potential estimation errors by incorporating alternative, less optimistic cardinality estimations. Our query planner

combines the cost model and robustness measure to devise efficient query plans. The results of our experimental study provide the following insights regarding *RQ 2*: (1) Including the robustness measure improves the query plan efficiency for queries with s-o and o-o joins. (2) In the case that the cheapest plan is not robust enough, an alternative robust plan that is not too expensive should be chosen. (3) CROP overall outperforms state of the art TPF clients.

In our second approach, we focus on query execution robustness by adapting the operators to estimation errors. To this end, we propose a new class of adaptive join operators that are able to switch their join strategy at runtime. We propose a Polymorphic Bind Join (PBJ) and a Polymorphic Hash Join (PHJ), which are able to switch their join strategy from a bind to hash join and vice versa. Our theoretical analysis proves the correctness of both operators under set semantics. In our empirical evaluation, we investigate the impact of PBJ and PHJ on the query execution efficiency for different planning approaches. The results show that especially the left-deep query planning approaches benefit from the adaptivity of the operators. In addition, we find that the gains in robustness during query execution depend on the operator and the type of queries. In particular, the PHJ consistently enables more robust query execution, while the PBJ yields better results for the more selective queries from the WatDiv benchmark (*RQ 3*).

Concluding, we found that robust query processing approaches for Linked Data Fragments enable more efficient query execution. Robust query planning approaches help to devise efficient query plans that are less prone to potential cardinality estimation errors. Adaptive join operators can enhance the robustness during query execution by reducing the impact of suboptimal query planning decisions. Future work may continue in both directions. Our query planning approach should be extended and evaluated for additional LDF interfaces. Moreover, existing clients for LDF interfaces, such as Comunica or smart-KG, could benefit from implementing the cost model and the robustness measure. In the area of adaptive join operators, future work should investigate alternative switch rules for the polymorphic join operators, e.g., by considering information from the query planner similar to [21] and [24]. In addition, other types of polymorphic join operators should be studied. For instance, operators that are able to switch their strategies several times. Lastly, the operators can be extended to further leverage the properties of the LDF interfaces, such as sorted tuples or the support of several bindings in the expressions.

References

[1] R. Verborgh, M.V. Sande, O. Hartig, J.V. Herwegen, L.D. Vocht, B.D. Meester, G. Haesendonck and P. Colpaert, Triple Pattern Fragments: A low-cost knowledge graph interface for the Web, *J. Web Semant.* **37-38** (2016), 184–206. doi:10.1016/j.websem.2016.03.003.

[2] O. Hartig and C.B. Aranda, Bindings-Restricted Triple Pattern Fragments, in: *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*, C. Debruyne, H. Panetto, R. Meersman, T.S. Dillon, eva Kühn, D. O'Sullivan and C.A. Ardagna, eds, Lecture Notes in Computer Science, Vol. 10033, 2016, pp. 762–779. doi:10.1007/978-3-319-48472-3_48.

[3] A. Azzam, J.D. Fernández, M. Acosta, M. Beno and A. Polleres, SMART-KG: Hybrid Shipping for SPARQL Querying on the Web, in: *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Y. Huang, I. King, T. Liu and M. van Steen, eds, ACM / IW3C2, 2020, pp. 984–994. doi:10.1145/3366423.3380177.

[4] T. Minier, H. Skaf-Molli and P. Molli, SaGe: Web Preemption for Public SPARQL Query Services, in: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R.W. White, A. Mantrach, F. Silvestri, J.J. McAuley, R. Baeza-Yates and L. Zia, eds, ACM, 2019, pp. 1268–1278. doi:10.1145/3308558.3313652.

[5] M. Acosta and M. Vidal, Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data, in: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan and S. Staab, eds, Lecture Notes in Computer Science, Vol. 9366, Springer, 2015, pp. 111–127. doi:10.1007/978-3-319-25007-6_7.

[6] R. Taelman, J.V. Herwegen, M.V. Sande and R. Verborgh, Comunica: A Modular SPARQL Query Engine for the Web, in: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, D. Vrandecic, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee and E. Simperl, eds, Lecture Notes in Computer Science, Vol. 11137, Springer, 2018, pp. 239–255. doi:10.1007/978-3-030-00668-6_15.

[7] S. Yin, A. Hameurlain and F. Morvan, Robust Query Optimization Methods With Respect to Estimation Errors: A Survey, *SIGMOD Rec.* **44**(3) (2015), 25–36. doi:10.1145/2854006.2854012.

[8] J.L. Wiener, H.A. Kuno and G. Graefe, Benchmarking Query Execution Robustness, in: *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, R.O. Nambiar and M. Poess, eds, Lecture Notes in Computer Science, Vol. 5895, Springer, 2009, pp. 153–166. doi:10.1007/978-3-642-10424-4_12.

[9] L. Heling and M. Acosta, Cost- and Robustness-Based Query Optimization for Linked Data Fragments, in: *The Semantic Web - ISWC 2020 - 19th International Semantic Web Confer-*

ence, Athens, Greece, November 2-6, 2020, Proceedings, Part I, J.Z. Pan, V.A.M. Tamma, C. d'Amato, K. Janowicz, B. Fu, A. Polleres, O. Seneviratne and L. Kagal, eds, Lecture Notes in Computer Science, Vol. 12506, Springer, 2020, pp. 238–257. doi:10.1007/978-3-030-62419-4_14.

[10] A. Deshpande, Z.G. Ives and V. Raman, Adaptive Query Processing, *Found. Trends Databases* **1**(1) (2007), 1–140. doi:10.1561/1900000001.

[11] B. Quilitz and U. Leser, Querying Distributed RDF Data Sources with SPARQL, in: *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, S. Bechhofer, M. Hauswirth, J. Hoffmann and M. Koubarakis, eds, Lecture Notes in Computer Science, Vol. 5021, Springer, 2008, pp. 524–538. doi:10.1007/978-3-540-68234-9_39.

[12] O. Görlitz and S. Staab, SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions, in: *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, O. Hartig, A. Harth and J.F. Sequeda, eds, CEUR Workshop Proceedings, Vol. 782, CEUR-WS.org, 2011. http://ceur-ws.org/Vol-782/GoerlitzAndStaab_COLD2011.pdf.

[13] A. Charalambidis, A. Troumpoukis and S. Konstantopoulos, SemaGrow: optimizing federated SPARQL queries, in: *Proceedings of the 11th International Conference on Semantic Systems, SEMANTICS 2015, Vienna, Austria, September 15-17, 2015*, A. Polleres, T. Pellegrini, S. Hellmann and J.X. Parreira, eds, ACM, 2015, pp. 121–128. doi:10.1145/2814864.2814886.

[14] G. Montoya, H. Skaf-Molli and K. Hose, The Odyssey Approach for Optimizing Federated SPARQL Queries, in: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, C. d'Amato, M. Fernández, V.A.M. Tamma, F. Lécué, P. Cudré-Mauroux, J.F. Sequeda, C. Lange and J. Heflin, eds, Lecture Notes in Computer Science, Vol. 10587, Springer, 2017, pp. 471–489. doi:10.1007/978-3-319-68288-4_28.

[15] M. Saleem, A. Potocki, T. Soru, O. Hartig and A.N. Ngomo, CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation, in: *Proceedings of the 14th International Conference on Semantic Systems, SEMANTICS 2018, Vienna, Austria, September 10-13, 2018*, A. Fensel, V. de Boer, T. Pellegrini, E. Kiesling, B. Haslhofer, L. Hollink and A. Schindler, eds, Procedia Computer Science, Vol. 137, Elsevier, 2018, pp. 163–174. doi:10.1016/j.procs.2018.09.016.

[16] M. Acosta, M. Vidal, T. Lampo, J. Castillo and E. Ruckhaus, ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints, in: *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N.F. Noy and E. Blomqvist, eds, Lecture Notes in Computer Science, Vol. 7031, Springer, 2011, pp. 18–34. doi:10.1007/978-3-642-25073-6_2.

[17] S.J. Lynden, I. Kojima, A. Matono and Y. Tanimura, ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints, in: *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part II*, R. Meersman, T.S. Dillon, P. Herrero, A. Kumar, M. Reichert, L. Qing, B.C. Ooi, E. Damiani, D.C. Schmidt, J. White, M. Hauswirth, P. Hitzler and M.K. Mohania, eds, Lecture Notes in Computer Science, Vol. 7045, Springer, 2011, pp. 808–817. doi:10.1007/978-3-642-25106-1_28.

[18] O. Hartig, I. Letter and J. Pérez, A Formal Framework for Comparing Linked Data Fragments, in: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, C. d'Amato, M. Fernández, V.A.M. Tamma, F. Lécué, P. Cudré-Mauroux, J.F. Sequeda, C. Lange and J. Heflin, eds, Lecture Notes in Computer Science, Vol. 10587, Springer, 2017, pp. 364–382. doi:10.1007/978-3-319-68288-4_22.

[19] L. Heling and M. Acosta, A Framework for Federated SPARQL Query Processing over Heterogeneous Linked Data Fragments, *CoRR* **abs/2102.03269** (2021). https://arxiv.org/abs/2102.03269.

[20] B. Babcock and S. Chaudhuri, Towards a Robust Query Optimizer: A Principled and Practical Approach, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, ed., ACM, 2005, pp. 119–130. doi:10.1145/1066157.1066172.

[21] S. Babu, P. Bizarro and D.J. DeWitt, Proactive Re-optimization, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, ed., ACM, 2005, pp. 107–118. doi:10.1145/1066157.1066171.

[22] F. Wolf, M. Brendle, N. May, P.R. Willems, K. Sattler and M. Grossniklaus, Robustness Metrics for Relational Query Execution Plans, *Proc. VLDB Endow.* **11**(11) (2018), 1360–1372. doi:10.14778/3236187.3236191. http://www.vldb.org/pvldb/vol11/p1360-wolf.pdf.

[23] A. Gounaris, N.W. Paton, A.A.A. Fernandes and R. Sakellariou, Adaptive Query Processing: A Survey, in: *Advances in Databases, 19th British National Conference on Databases, BNCOD 19, Sheffield, UK, July 17-19, 2002, Proceedings*, B. Eaglestone, S. North and A. Poulovassilis, eds, Lecture Notes in Computer Science, Vol. 2405, Springer, 2002, pp. 11–25. doi:10.1007/3-540-45495-0_2.

[24] V. Markl, V. Raman, D.E. Simmen, G.M. Lohman and H. Pirahesh, Robust Query Processing through Progressive Optimization, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A.C. König and S. Deßloch, eds, ACM, 2004, pp. 659–670. doi:10.1145/1007568.1007642.

[25] M. Schmidt, M. Meier and G. Lausen, Foundations of SPARQL query optimization, in: *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, L. Segoufin, ed., ACM International Conference Proceeding Series, ACM, 2010, pp. 4–33. doi:10.1145/1804669.1804675.

[26] C.B. Aranda, M. Arenas and Ó. Corcho, Semantics and Optimization of the SPARQL 1.1 Federation Extension, in: *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, G. Antoniou, M. Grobelnik, E.P.B. Simperl, B. Parsia, D. Plexousakis, P.D. Leenheer and J.Z. Pan, eds, Lecture Notes in Computer Science, Vol. 6644, Springer, 2011, pp. 1–15. doi:10.1007/978-3-642-21064-8_1.

[27] L. Heling, M. Acosta, M. Maleshkova and Y. Sure-Vetter, Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study, in: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, D. Vrandecic, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee and E. Simperl, eds, Lecture Notes in Computer Science, Vol. 11137, Springer, 2018, pp. 86–102. doi:10.1007/978-3-030-00668-6_6.

[28] E. Wössner, C. Qin, J. Fernández and M. Acosta, Triple Pattern Join Cardinality Estimations over HDT with Enhanced Metadata, in: *Proceedings of the Posters and Demo Track of the 15th International Conference on Semantic Systems co-located with 15th International Conference on Semantic Systems (SEMANTiCS 2019), Karlsruhe, Germany, September 9th - to - 12th, 2019*, M. Alam, R. Usbeck, T. Pellegrini, H. Sack and Y. Sure-Vetter, eds, CEUR Workshop Proceedings, Vol. 2451, CEUR-WS.org, 2019. http://ceur-ws.org/Vol-2451/paper-31.pdf.

[29] D. Kossmann and K. Stocker, Iterative dynamic programming: a new class of query optimization algorithms, *ACM Trans. Database Syst.* **25**(1) (2000), 43–82. doi:10.1145/352958.352982.

[30] G. Aluç, O. Hartig, M.T. Özsu and K. Daudjee, Diversified Stress Testing of RDF Data Management Systems, in: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C.A. Knoblock, D. Vrandecic, P. Groth, N.F. Noy, K. Janowicz and C.A. Goble, eds, Lecture Notes in Computer Science, Vol. 8796, Springer, 2014, pp. 197–212. doi:10.1007/978-3-319-11964-9_13.

[31] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres and M. Arias, Binary RDF representation for publication and exchange (HDT), *J. Web Semant.* **19** (2013), 22–41. doi:10.1016/j.websem.2013.01.002.

[32] M. Acosta, M. Vidal and Y. Sure-Vetter, Diefficiency Metrics: Measuring the Continuous Efficiency of Query Processing Approaches, in: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II*, C. d'Amato, M. Fernández, V.A.M. Tamma, F. Lécué, P. Cudré-Mauroux, J.F. Sequeda, C. Lange and J. Heflin, eds, Lecture Notes in Computer Science, Vol. 10588, Springer, 2017, pp. 3–19. doi:10.1007/978-3-319-68204-4_1.