Join Ordering of SPARQL Property Path Queries

Julien Aimonier-Davat(✉), Hala Skaf-Molli, Pascal Molli, Minh-Hoang Dang, and Brice Nédelec

LS2N, University of Nantes, 2 Rue de la Houssinière - BP 92208, 44322 Nantes Cedex 3, France
{julien.aimonier-davat,hala.skaf-molli,pascal.molli, minh-hoang.dang,brice.nedelec}@univ-nantes.fr

Abstract. SPARQL property path queries provide a succinct way to write complex navigational queries over RDF knowledge graphs. However, their evaluation remains difficult as they may involve the execution of transitive closures. As a result, many property path queries just timeout when executed on public online RDF knowledge graphs. One solution to speed up their execution is to find optimal join orders. Although the join ordering problem has been extensively studied for traditional SPARQL queries, the presence of property path patterns biases existing approaches. In this paper we focus on $C2RPQ_{UF}$ queries (conjunctive SPARQL property path queries with UNION and FILTER), and we present a query optimizer that is able to capture the cost of $C2RPQ_{UF}$ queries using an appropriate cost model and a sampling-based cardinality estimator. On the latest Wikidata Query Benchmark, we empirically demonstrate that our approach finds significantly better join orders than Virtuoso and BlazeGraph.

Keywords: Join Order · SPARQL Property Path · Random Walks · Sampling

1 Introduction

Context and Motivation: SPARQL 1.1 [30] introduced property paths to add extensive navigational capabilities to the SPARQL query language. Property path queries (PPQs) are intensively used on Wikidata; they account for 38% of the entire query log [5]. Transitive closures are a crucial part of property paths as they allow to match paths of arbitrary length. According to the log of Wikidata, transitive closures are used by 66% of the property path queries. However, transitive closures make the evaluation of property path queries challenging [33], and many of them cannot terminate in less than 60s. For example, the query presented in Fig. 1a searches for road bicycle races in Central America and time-out on Wikidata.

Related Works: Speeding up the processing of PPQs received much attention from the semantic web community [2]. One approach relies on a dedicated

https://doi.org/10.1007/978-3-031-33455-9_3

```
SELECT ?x1 ?x3 WHERE {
  ?x3 wdt:P361 wd:Q27611 .    # tp1 (9)
  ?x1 wdt:P17 ?x3 .           # tp2 (13M)
  ?x1 wdt:P641* wd:Q3609 .    # tp3 (47K)
}
```

(a) $Q_1^{J_1}$: BlazeGraph's join order

```
SELECT ?x1 ?x3 WHERE {
  hint:Query hint:optimizer "None" .
  ?x1 wdt:P641* wd:Q3609 . # tp3
  hint:Prior hint:gearing "reverse" .
  ?x1 wdt:P17 ?x3 .        # tp2
  ?x3 wdt:P361 wd:Q27611 . # tp1
}
```

(b) $Q_1^{J_2}$: Hand-crafted join order

Fig. 1. Query Q_1 comes from the Wikidata Query Benchmark [3] and returns road bicycle races located in Central America. $Q_1^{J_1}$ is the join order decided by BlazeGraph while $Q_1^{J_2}$ is a hand-crafted join order. The "hint:Query" triple pattern in $Q_1^{J_2}$ is used to force the join order in BlazeGraph. Commented numbers are triple patterns cardinality. On the Wikidata Query Service $Q_1^{J_1}$ time-out (>60s) while $Q_1^{J_2}$ terminates in less than 3 s.

index [28] that improves PPQs execution time, but requires the construction and maintenance of the index. Other works propose new dedicated operators to process transitive closures [1,4,27,32,33]. Such approaches are very effective but focus on evaluating property path patterns (PPPs) alone, while most of the time PPPs are just part of a PPQ. For instance, tp_3 is just a pattern among others in the query Q_1 in Fig. 1. In this paper, we propose to improve PPQs execution time by finding better join orders. Compared to previous works, finding better join orders allows us to improve PPQs execution time on existing engines, without new indexes and new operators. To illustrate the impact of finding good join orders, Q_1 has been executed on the Wikidata query service using two different join orders: (1) a join order $J_1 = ((tp1 \bowtie tp2) \bowtie tp_3)$ that has been decided by BlazeGraph, the SPARQL engine behind Wikidata (2) a join order $J_2 = ((tp_3 \bowtie tp2) \bowtie tp1)$ that has been hand-crafted. Following J_1 the query $Q_1^{J_1}$ time-out on Wikidata, i.e. $Q_1^{J_1}$ requires more than 60 s to complete, while $Q_1^{J_2}$ terminates in less than 3 s. Although the join ordering problem has been extensively studied in the context of conjunctive queries with filters [12,13,17], it has been poorly explored when considering property path queries [10,11]. It is currently unclear how current engines consider path patterns, i.e. how the cost of a join order that contains PPPs is computed, and why it should be computed like that.

Approach and Contributions: This paper focuses on the class of conjunctive two-way regular path queries with UNION and FILTER, denoted $C2RPQ_{UF}$ in [5]. For $C2RPQ_{UF}$ queries, we propose a query optimizer that can find efficient join orders without changing existing SPARQL engines. Finding such join orders is challenging; depending on the join order, a path pattern may behave as a transitive closure or a reachability pattern. The changing nature of path patterns is biasing traditional cost models that fail to find efficient join orders. The contributions of this paper are the following:

1. This paper proposes a cost model along with a dynamic programming (DP) algorithm able to capture the cost of evaluating PPQs using traditional PPP operators. Compared to state-of-the-art, the proposed DP algorithm can

rewrite PPPs such that their cost remain observable to existing cost functions as the one defined in [18].
2. Any cost model requires accurate cardinality estimates. However, there is currently no cardinality estimator able to handle property path patterns. Consequently, this paper proposes a cardinality estimator for PPQs based on random walks. Random walks can be computed for cheap thanks to B-Tree indexes, widely used to index RDF data. Compared to state-of-the-art, the proposed cardinality estimator extends the WanderJoin approach [18] to handle property path patterns.
3. The approach is evaluated on BlazeGraph and Virtuoso using the newly proposed Wikidata Query Benchmark [3]. Experimental results demonstrate that our approach significantly improves SPARQL property path queries performance in terms of execution time. Compared to BlazeGraph, the execution time is divided by at least 14.

The rest of the paper is organized as follows: Section 2 presents our approach, preliminaries, and the problem of ordering joins in the presence of property path patterns. Section 3 introduces a cost model for property path queries. Section 4 presents our cardinality estimator for property path queries. Section 5 details our experimental results. Finally, after discussing related works in Sect. 6, we present our conclusions and future work in Sect. 7.

2 Query Optimization of Property Path Queries

This paper follows the traditional query optimizer architecture of the system R [26]. The optimizer takes a property path query as input and returns a physical plan that minimizes a cost function. For each property path pattern, the optimizer also decides in which direction it should be evaluated, i.e. from subjects to objects (*forward navigation*) or from objects to subjects (*backward navigation*). The query optimizer enumerates valid join orders using a dynamic programming algorithm. Based on cardinality estimates, the cost model chooses the cheapest alternative among the valid join orders. The goal of this paper is to target mainstream SPARQL engines, consequently two hypotheses are assumed: (1) There are no indexes dedicated to transitive closures, such as the FERRARI index [28] (2) Property path patterns are evaluated using the traditional ALP procedure (a BFS-style algorithm) defined in the SPARQL specification [30] as it is done in JENA or BlazeGraph, or using transitive closure operators as in Virtuoso.

2.1 Preliminaries

SPARQL Query: This paper follows the notations from [24,25] and considers three disjoint sets I (IRIs), L (literals) and B (blank nodes). Let $T = I \cup L \cup B$ be the set of RDF terms, an RDF triple $(s, p, o) \in (I \cup B) \times I \times T$ connects a subject s through a predicate p to an object o. An RDF graph G is a finite set of RDF triples. Let V be an infinite set of variables, disjoint from the previous sets. A graph pattern P is defined recursively as follows:

1. A tuple from $(I \cup B \cup V) \times (I \cup V) \times (T \cup V)$ is a triple graph pattern.
2. If P_1 and P_2 are graph patterns, then (P_1 AND P_2) and (P_1 UNION P_2) are respectively a conjunctive graph pattern and an union graph pattern.
3. If P is a graph pattern and R is a SPARQL built-in condition, then (P FILTER R) is a filter graph pattern.

Given P a graph pattern, $var(P)$ is the set of variables found in P. The semantics of SPARQL queries is defined in terms of mappings. A mapping μ is a partial function $\mu : V \rightarrow T$. The domain of μ, denoted $dom(\mu)$, is the subset of V on which μ is defined. Given a triple pattern tp, $\mu(tp)$ is the image of tp under μ, i.e. the triple obtained by replacing the variables in tp according to μ. Letting $\mu(P)$ denote the image of P under μ, with respect to the latter definition, the evaluation of a graph pattern P over an RDF graph G is defined as $[\![P]\!]_G = \{ \mu \mid dom(\mu) = var(P) \wedge \mu(P) \subseteq G \}$. To simplify explanations around random walks, this paper assumes that the evaluation of a triple pattern tp over an RDF graph G returns a set of triples, i.e. $[\![tp]\!]_G = \{ \mu(tp) \mid dom(\mu) = var(tp) \wedge \mu(tp) \subseteq G \}$.

SPARQL Property Path Query (PPQ): A property path query is a SPARQL query with at least one property path pattern (PPP). A PPP is a tuple in $(I \cup B \cup V) \times E \times (T \cup V)$ where E is the set of property path expressions. Based on [16], property path expressions are defined by the grammar: $e := a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !a_1, \ldots, a_k \mid !a_1^-, \ldots, a_k^-$ where $a, a_1, ..., a_k \in I$. This paper assumes non-transitive property path expressions to be evaluated using traditional SPARQL algebra operators [30]. Thus, the paper focuses only on the evaluation of transitive property path expressions, i.e. e^+ and e^*. Nested stars, e.g. $(a+)+$, are not considered and will be the subject of future work.

2.2 The Join Ordering Problem with Property Paths

$$C_{mm}(P,G) = \begin{cases} |[\![P]\!]_G| & \textbf{if } P = tp \vee P = \sigma(tp) \\ C_{mm}(P_1) + & \textbf{if } P = P_1 \overset{NLJ}{\bowtie} P_2, \\ \quad |[\![P_1]\!]_G| \times max(\frac{|[\![P_1 \bowtie tp]\!]_G|}{|[\![P_1]\!]_G|}, 1) & (P_2 = tp \vee P_2 = \sigma(tp)) \end{cases}$$

Let us consider the C_{mm} cost function defined above, which is a simplified version of the one presented in [17]. For simplicity, only index-nested-loop joins and left-deep trees are considered, but our proposal holds in the general case. Most cost functions rely on cardinality estimates. For instance, the C_{mm} function defines the cost of evaluating a triple pattern as its cardinality [17]. Indeed, considering traditional indexes SPO, POS, OSP as available, any triple pattern tp can be evaluated over an RDF graph G in $\mathcal{O}(|[\![tp]\!]_G| log(|G|))$. Thus, for conjunctive queries with filters, minimizing a cardinality-based cost function such as the C_{mm} effectively leads to good join orders [7,17]. While the cost of evaluating a triple pattern is correlated with its cardinality, it is not always true for property

```
SELECT ?x1 ?x3 WHERE {
  ?x3 wdt:P361 wd:Q27611 .          # tp1 (9)
  ?x1 wdt:P17 ?x3 .                 # tp2 (13M)
  ?x1 wdt:P641* ?relax .
  FILTER (?relax = wd:Q3609) .      # tp3 (47K)
}
```

(a) $Q_1^{J_1^F}$: Forward relaxation of J_1.

```
SELECT ?x1 ?x3 WHERE {
  ?x3 wdt:P361 wd:Q27611 .          # tp1 (9)
  ?x1 wdt:P17 ?x3 .                 # tp2 (13M)
  ?relax wdt:P641* wd:Q3609 .       # tp3 (47K)
  FILTER (?relax = ?x1) .
}
```

(b) $Q_1^{J_1^B}$: Backward relaxation of J_1.

Fig. 2. Forward and Backward relaxations of tp_3 in J_1 of query Q_1.

path patterns (PPPs). Assuming PPPs are evaluated using ALP, a BFS-style algorithm defined by the standard [30], two cases can be distinguished:

Case 1, Transitive-pattern Let $tp = (s, p, o)$ be a PPP such that at least the subject or the object is a variable, i.e. $s \in V \vee o \in V$. Let o be in V and N be the set of nodes reachable from s. Using the ALP algorithm, the evaluation of tp returns N that can be computed in $\mathcal{O}(|N|log(|G|))$ over an RDF graph G. As $|[\![tp]\!]_G| = |N|$, the cost of evaluating tp is correlated with its cardinality.

Case 2, Reachability-pattern Let $tp = (s, p, o)$ be a PPP such that both the subject and the object are bounded, i.e. $s, o \notin V$. According to [30], the cardinality of a fully bounded PPP is 1 if o can be reached from s, 0 otherwise. However, to check the reachability between s and o, the ALP algorithm first computes the set of nodes N reachable from s, then checks if $o \in N$. Consequently, the cost of evaluating a fully bounded PPP is not correlated with its cardinality.

To illustrate the problem on a concrete example, let us compute the cost of $Q_1^{J_1}$ and $Q_1^{J_2}$ depicted in Fig. 1. Using the true cardinalities[1], we calculate $C_{mm}(Q_1^{J_1}) \approx 91K$ and $C_{mm}(Q_1^{J_2}) \approx 168K$. Despite $Q_1^{J_1}$ being estimated less costly than $Q_1^{J_2}$, $Q_1^{J_1}$ time-out on Wikidata while $Q_1^{J_2}$ completes in less than 3 s. Focusing on property path patterns, tp_3 appears as a transitive-pattern in $Q_1^{J_2}$, while it appears as a reachability-pattern in $Q_1^{J_1}$. Because the cost of computing N is not captured by the cardinality of a reachability-pattern, the cost of $Q_1^{J_1}$ is largely underestimated. Thus, a cost function purely based on cardinalities, such as the C_{mm}, cannot correctly estimate the cost of a PPQ. *The scientific challenge is to define a cost model that, given any join order, can capture the cost of a PPP whether it appears as a transitive-pattern or a reachability-pattern.*

3 Cost-Model for Property Path Queries

Given a join order J, the key idea is to relax fully bounded property path patterns (PPPs) such that J no longer contains reachability-patterns. If all PPPs behave as transitive-patterns, then cardinality-based cost functions are able to correctly estimate the cost of J. Whether a PPP behaves as a transitive-pattern or a reachability-pattern depends on the join order. Therefore, the general approach

[1] True cardinalities were computed using SPARQL COUNT queries on the Wikidata SPARQL endpoint as of December 5, 2022.

is to detect reachability-patterns during enumeration of join orders, and relax them before ordering them.

Definition 1 (Reachability-pattern relaxation). *Let Q be a SPARQL property path query, J a join order, and $tp = (s, p, o) \notin J$ a fully bounded property path pattern with respect to J, i.e. $var(tp) \subseteq var(J)$. A forward relaxation of tp generates a filter graph pattern of the form $((s, p, v)\ FILTER\ (v = o))$ such that $v \notin var(Q)$. A backward relaxation of tp generates a filter graph pattern of the form $((v, p, o)\ FILTER\ (v = s))$ such that $v \notin var(Q)$.*

Using a BFS-style algorithm, a reachability-pattern can be evaluated following two strategies. One can decide to navigate from the subject to the object (forward strategy), another can decide to go from the object to the subject (backward strategy). In this context, the forward and backward relaxations allow to estimate which strategy is the cheapest one. According to [33], the cost of going forward or backward can be drastically different. Selecting the best strategy is therefore important to expect good performance. For instance, BlazeGraph evaluates tp_3 in $Q_1^{J_1}$ starting from the object. Using the forward and backward relaxations, we can rewrite $Q_1^{J_1}$ as $Q_1^{J_1^F}$ and $Q_1^{J_1^B}$ as depicted in Fig. 2. If we use the true cardinalities to compute the cost of both strategies, we get $C_{mm}(Q_1^{J_1^F}) = 95K$ and $C_{mm}(Q_1^{J_1^B}) = 2.6B$. Thus, $Q_1^{J_1}$ time-out on Wikidata because BlazeGraph chose the wrong strategy to evaluate tp_3.

Algorithm 1: Dynamic Programming with Relaxation

```
Require: $Q$: SPARQL Property Path Query, $G$: RDF Graph
Data: $dpTable$: keeps the best join order for a set $S$ of triple/property path patterns
1  for $\forall tp \in Q$ do $dpTable[\{tp\}] = tp$
2  for $\forall n \in 1..|Q| - 1$ do
3      for $\forall S \in dpTable : |S| = n$ do
4          for $\forall tp \in Q : tp \notin S$ do
5              if $var(tp) \cap var(dpTable[S]) = \emptyset$ then continue
6              $S' = S \cup \{tp\}$ ; $C = \emptyset$
7              if $tp$ is a $PPP \wedge var(tp) \subseteq var(dpTable[S])$ then
8                  $C = C \cup \{dpTable[S] \bowtie ForwardRelaxation(Q, tp)\}$
9                  $C = C \cup \{dpTable[S] \bowtie BackwardRelaxation(Q, tp)\}$
10             else
11                 $C = C \cup \{dpTable[S] \bowtie tp\}$
12             for $\forall P \in C$ do
13                 if $S' \notin dpTable \vee C_{mm}(P, G) < C_{mm}(dpTable[S'], G)$ then
14                     $dpTable[S'] = P$
15 return $UndoRelaxation(dpTable[Q])$
```

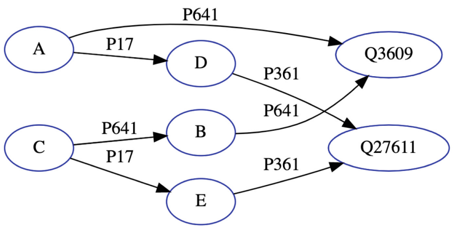
Algorithm 1 is a custom dynamic programming algorithm that integrates relaxation. When the algorithm detects a reachability-pattern tp with respect to a join order $J = dpTable[S]$ (Line 7), it uses relaxation so that the cost function is able to correctly estimate the cost of tp in J. Moreover, to select the

```
SELECT DISTINCT ?x1 ?x3 WHERE {
   ?x1 wdt:P641 wd:Q3609 .    # tp3
   ?x1 wdt:P17 ?x3 .          # tp2
   ?x3 wdt:P361 wd:Q27611 .   # tp1
}
```

(a) $(Q_1^{J_2})^{1..1}$

```
SELECT DISTINCT ?x1 ?x3 WHERE {{
   ?x1 wdt:P641 wd:Q3609 .    # tp3
   ?x1 wdt:P17 ?x3 .          # tp2
   ?x3 wdt:P361 wd:Q27611 .   # tp1
} UNION {
   ?v1 wdt:P641 wd:Q3609 .    # tp3.1
   ?x1 wdt:P641 ?v1           # tp3.2
   ?x1 wdt:P17 ?x3 .          # tp2
   ?x3 wdt:P361 wd:Q27611 .   # tp1
}}
```

(b) $(Q_1^{J_2})^{1..2}$



(c) RDF graph G_1

Fig. 3. RDF graph G_1 and rewrites of the query $Q_1^{J_2}$ used to estimate the cost of J_2 with random walks.

best strategy to evaluate tp both relaxations are used, generating two candidates that are stored in C (Line 8-9). One of them will evaluate tp using the forward strategy, while the other will use the backward strategy. Next, the cost function is used to keep the cheapest alternative (Line 12-14). At the end, the algorithm returns the cheapest join order, with relaxed property path patterns in their original form.

4 Cardinality Estimation of Property Path Queries

This section introduces a new cardinality estimator for property path queries based on random walks. Random walks [19] offer several advantages: (1) They proved to be the best approach for estimating the cardinality of conjunctive SPARQL queries [23] (2) They do not require maintaining statistics [10] (3) They can be efficiently implemented just by relying on traditional SPO, POS, and OSP indexes that are widely available on existing triple stores [18]. Before moving to the contribution, we first recall how to estimate the cardinality of conjunctive SPARQL queries using random walks. Next, we address the case of SPARQL property path queries. For the sake of simplicity, we assume that property path queries are conjunctive queries with a single property path pattern. However, the approach can be generalized to $C2RPQ_{UF}$ queries that contain multiple property path patterns.

4.1 Cardinality Estimates of Conjunctive Queries

Let Q be a conjunctive SPARQL query, and $J = \langle tp_1, ..., tp_n \rangle$ be the join order used to perform random walks. Based on [19] a random walk $\gamma = \langle t_1, ..., t_n \rangle$ is computed over an RDF graph G by randomly picking t_1 in $[\![tp_1]\!]_G$, and each subsequent t_i $(i > 1)$ in $[\![t_{i-1} \bowtie tp_i]\!]_G$. Thus, the probability of sampling γ is $P(\gamma) = |[\![tp_1]\!]_G|^{-1} \prod_{i=2}^{n} |[\![t_{i-1} \bowtie tp_i]\!]_G|^{-1}$. Let $\Gamma = \langle \gamma_1, ..., \gamma_k \rangle$ be a multiset of k random walks, the cardinality of Q is estimated as $card(\Gamma) = |\Gamma|^{-1} \sum_{i=1}^{|\Gamma|} P(\gamma_i)^{-1}$. For instance, let us estimate the cardinality of $(Q_2^{J_2})^{1..1}$ on the RDF graph G_1 with a budget of 2 random walks. Both $(Q_2^{J_2})^{1..1}$ and G_1 are depicted in Fig. 3. Let γ_1 and γ_2 be the two random walks we picked following J_2:

$$\gamma_1 \left| \begin{array}{l|ll} tp_3 & \text{picking } t_1 = (\mathbf{A}, P641, Q3609) & \text{in } [\![(?x1, P641, Q3609)]\!]_{G_1} \\ tp_2 & \text{picking } t_2 = (A, P17, \mathbf{D}) & \text{in } [\![(\mathbf{A}, P17, ?x3)]\!]_{G_1} \\ tp_1 & \text{picking } t_3 = (D, P361, Q27611) & \text{in } [\![(\mathbf{D}, P361, Q27611)]\!]_{G_1} \end{array} \right.$$

$$\gamma_2 \left| tp_3 \right| \text{ picking } t_1 = (\mathbf{B}, P641, Q3609) \text{ in } [\![(?x1, P641, Q3609)]\!]_{G_1}$$

In this example, $P(\gamma_1) = \frac{1}{2} \times \frac{1}{1} \times \frac{1}{1} = \frac{1}{2}$, while $P(\gamma_2) = 0$. Indeed, when it becomes impossible for a random walk γ to sample t_i for some $i \leq n$, e.g. t_2 in γ_2 because $[\![(B, P17, ?x3)]\!]_{G_1} = \emptyset$, γ is classified as invalid, and its probability of being sampled is 0. Thus, the estimated cardinality of $(Q_2^{J_2})^{1..1}$ is $\frac{1}{2} \times (P(\gamma_1)^{-1} + P(\gamma_2)^{-1}) = \frac{2+0}{2} = 1$.

4.2 Cardinality Estimates of Property Path Queries

Definition 2. *Let Q be a conjunctive SPARQL property path query. Let $tp_i \in Q$ be a property path pattern. We denote Q^d the conjunctive SPARQL query obtained by rewriting tp_i into a chain $tp_i^1, ..., tp_i^d$ of triple patterns. If $J = \langle tp_1, ..., tp_i, ..., tp_{|Q|} \rangle$ is a join order associated to Q, we denote $J^d = \langle tp_1, ..., tp_i^1, ..., tp_i^d, ..., tp_{|Q|} \rangle$ the equivalent join order associated to Q^d.*

To estimate the cardinality of a SPARQL property path query Q, the key idea is to rewrite Q into an equivalent query Q' that does not contain property paths. For instance, let us consider the query $Q_1^{J_2}$ depicted in Fig. 1b. Assuming that the diameter d of the relation $P641$ is known, and let $d = 2$, $Q_1^{J_2}$ is equivalent to the query $(Q_1^{J_2})^{1..2}$ described in Fig. 3b, i.e. both return the same result. Knowing d, any PPP can be rewritten as an `UNION` graph pattern with d clauses, each clause matching paths of different lengths from 1 to d. Thus, assuming a budget of k random walks, the cardinality of Q' can be estimated by uniformly distributing random walks over the d clauses of the `UNION`, ending up with d multisets of random walks $\Gamma_1, ..., \Gamma_d$. The cardinality of Q' is then estimated as $\sum_{i=1}^{d} card(\Gamma_i)$. In other words, we consider clauses of the `UNION` as individual queries $Q^1, ..., Q^d$, for which we estimate the cardinality, and the cardinality of Q is the sum of the estimated cardinalities of $Q^1, ..., Q^d$.

According to the SPARQL semantics, PPPs are evaluated following a set-semantics [30]. For instance, no matter how many paths they are between `wd:Q3609` and `A`, evaluating tp_3 over G_1 must return `A` only once. Thus, for the rewriting to be correct, a `DISTINCT` modifier must be introduced in the rewriting of Q into Q'. However, to the best of our knowledge, estimating the cardinality of `DISTINCT` queries using random walks has not been studied. To cope with this issue, the `DISTINCT` modifier is just ignored, aware that the estimator will overestimate cardinalities. On dense graphs, cardinalities can be significantly overestimated, preventing the optimizer from finding good join orders. Nevertheless, we assume that in practice, removing the `DISTINCT` modifier will not prevent the optimizer from finding good join orders.

Algorithm 2: Cardinality Estimation with Property Paths

```
Require: $Q$: SPARQL query where $tp_i$ is a property path pattern, $J$: Join order, $G$: RDF
         graph, $k$: Number of random walks, $dMax$: Depth exploration limit
Data: $d$: Length of the longest path explored, $\Gamma$: Multisets of random walks
1  $d = 1$
2  while $\sum_j |\Gamma_j| < k$ do
3      $d' \sim \mathcal{U}\{1, d\}$ ; $\gamma = \langle t_1, ..., t_n \rangle = randomWalk(Q^{d'}, J^{d'}, G)$
4      if $P(\gamma) > 0 \wedge t_i^1, ..., t_i^{d'} \in \gamma$ are pairwise distinct then
5          $\Gamma_{d'} = \Gamma_{d'} \cup \{\gamma\}$
6      else
7          $\Gamma_{d'} = \Gamma_{d'} \cup \{\gamma'\}$ with $P(\gamma') = 0$
8      end
9      if $|\gamma| \geq i + d - 1 \wedge t_i^1, ..., t_i^d \in \gamma$ are pairwise distinct then
10         $d = min(d + 1, dMax)$
11     end
12 end
13 return $\sum_{j=1}^{d} card(\Gamma_j)$
```

Rewriting a property path pattern tp requires to know the diameter d of the subgraph recognized by tp. To avoid relying on statistics, Algorithm 2 computes d while performing random walks. Given a SPARQL property path query Q where tp_i is a PPP, and a budget k, Algorithm 2 starts with $d = 1$. At each iteration, the algorithm computes a random walk $\gamma = \langle t_1, ..., t_n \rangle$ from $Q^{d'}$, following the join order $J^{d'}$, where d' is drawn uniformly at random between 1 and d. Each time $d' = d$, the algorithm checks if γ has found a path of length d matching tp_i. Given $J^d = \langle tp_1, ..., tp_i^1, ..., tp_i^d, ..., tp_{|Q|} \rangle$, a path of length d has been found if γ matches at least $\langle tp_1, ..., tp_i^1, ..., tp_i^d \rangle$, i.e. if $|\gamma| \geq |\langle tp_1, ..., tp_i^1, ..., tp_i^d \rangle|$ or $|\gamma| \geq i + d - 1$. In this case, it may exist a path of length $d + 1$ matching tp_i, and d is increased by 1.

Algorithm 2 increases d each time a random walk finds a path of length d matching tp_i. However, in the presence of cycles, d may increase forever, significantly impacting the accuracy of estimates. To address this issue, Algorithm 2 enforces a simple path semantics. Under a simple path semantics [20], a random walk can go through a node only once when matching PPPs. In other words, let $\gamma = \langle t_1, ..., t_i^1, ..., t_i^{d'}, ..., t_{|Q|} \rangle$ be a random walk sampled from $Q^{d'}$, with $t_i^1, ..., t_i^{d'}$ matching $tp_i^{d'}$, γ is valid if and only if $t_i^1, ..., t_i^{d'}$ are pairwise distinct. Thus, considering an infinite number of random walks, Algorithm 2 ensures that d

Table 1. Characteristics of the workload used in the experiments

#queries	#joins	#triples	#path patterns	#constants	#join variables
213	1–8	2–9	1–2	1–5	1–6

converges to the size of the longest simple path in the subgraph recognized by tp_i, which is equal to or larger than the diameter of the subgraph.

Even without cycles, d can quickly reach large values. Because the budget k is distributed between queries $Q^1, ..., Q^d$ to sample paths of length 1 to d, with a small budget, the algorithm may end up with too few random walks in each multiset Γ_j to compute accurate estimates. To address this issue, d can be clipped to a maximum value $dMax$. As the computation time of a random walk is proportional to its size, clipping d can also improve optimization times.

5 Experimental Study

The goal of this experimental study is to empirically answer the following questions: (1) Does our approach improve total workload execution time compared to baselines? (2) What is the impact of our approach on each query? (3) How do the budget k and the depth exploration limit $dMax$ impact performance?

5.1 Experimental Setup

Datasets and Queries. Our experiments use the newly proposed Wikidata Graph Query Benchmark (WDBench) [3], which extracts real-world SPARQL queries from the public query logs of the Wikidata SPARQL endpoint. The WDBench provides a RDF dataset of 1,257,169,959 triples built from the dump of Wikidata. To create our workload, all non-property path queries have been filtered out, as well as queries with cross-products. The resulting workload contains 213 queries and is described in Table 1. To ensure that queries return the same result for all engines, we added the DISTINCT modifier to all queries.

Compared Approaches. To demonstrate that random walks have the potential to be used to improve the join order of SPARQL property path queries, we compare our approach with Virtuoso v.OS-7.2.7 [8] (one of the most deployed engine in practice [6], as well as the SPARQL endpoint behind DBpedia), and BlazeGraph v.2.1.4 [31] (used by the Wikidata Query Service [21]). BlazeGraph comes with two different optimizers; a first optimizer based on simple statistics such as the cardinality of triple patterns, and another one, named RTO, that relies on sampling to estimate the cost of join orders. Because RTO supports property paths and is adapted from ROX [15], which is related to our proposal, our approach is compared to both optimizers.

Implementation and Experimental Protocol. We implemented our query optimizer as a standalone Python 3.9 program. Random walks are performed

Table 2. TET=Total Execution Time, T=Timeouts, E=Errors.

Engine	Optimizer	TET [Seconds]	T	E
BlazeGraph	Default	20270	15	9
	RTO	35107	36	64
	Proposal (k=1000, d=5)	1429 ± 130	0	0
Virtuoso	Default	419	0	0
	Proposal (k=1000, d=5)	362 ± 11	0	0

over the WDBench dataset stored in HDT [9]. The generated plans of our query optimizer are translated into SPARQL queries using BlazeGraph and Virtuoso query hints. Query hints allow us to force the join order and the direction in which property path patterns are evaluated by the engine[2][3][4]. Virtuoso and BlazeGraph have been tuned for a system with 64GB, following engine recommendations. Code and configurations can be found online for reproducibility purposes[5] As random walks are not deterministic, the workload is optimized 5 times for each tested configuration, and each query is executed 3 times after a warmup execution. All queries are executed with a timeout of 900 s.

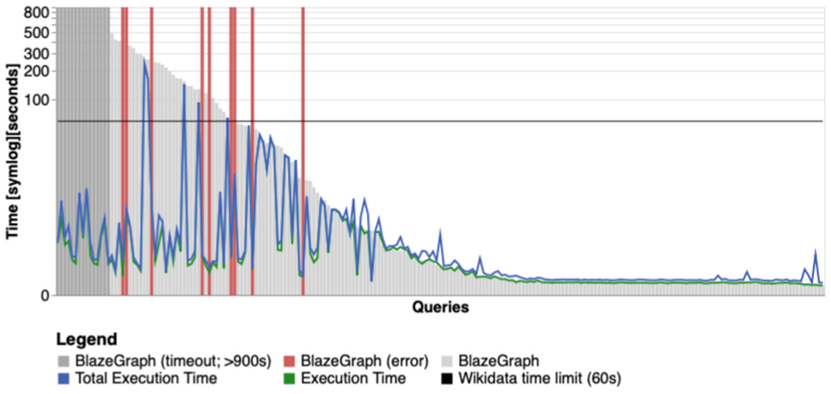
Evaluation Metrics. In our experiments, the following metrics are used: (1) The *Total Execution Time* is the time spent by Virtuoso or BlazeGraph executing a SPARQL query with the Optimization Time. (2) The *Execution Time* is the time spent by Virtuoso or BlazeGraph executing a SPARQL query without the Optimization Time. (3) The *Optimization Time* is the time spent by our query optimizer to optimize a SPARQL query. Each query is executed three times and the metrics are computed on the average of these 3 executions.

Hardware. All experiments ran on a single machine with Ubuntu 20.04.4 LTS, AMD EPYC 7513 32-Core Processor, 64GB of RAM, and a logical volume of 2TB on a remote SSD accessible through the LAN.

5.2 Experimental Results

Does Our Approach Improve Total Workload Execution Time Compared to Baselines? Table 2 presents the total execution time of the workload for all engines. Our proposal is configured with a budget of 1000 random walks (k=1000), and the depth limit for property paths is set to 5 (d=5). As our proposal relies on random walks, the workload has been optimized 5 times. Averages and standard deviations are reported in Table 2.

[2] https://docs.openlinksw.com/virtuoso/rdfsparqlimplementatiotrans.
[3] https://docs.openlinksw.com/virtuoso/rdfperfcost.
[4] https://github.com/blazegraph/database/wiki/QueryHints.
[5] https://github.com/JulienDavat/Join-Ordering-of-SPARQL-Property-Path-Queries.



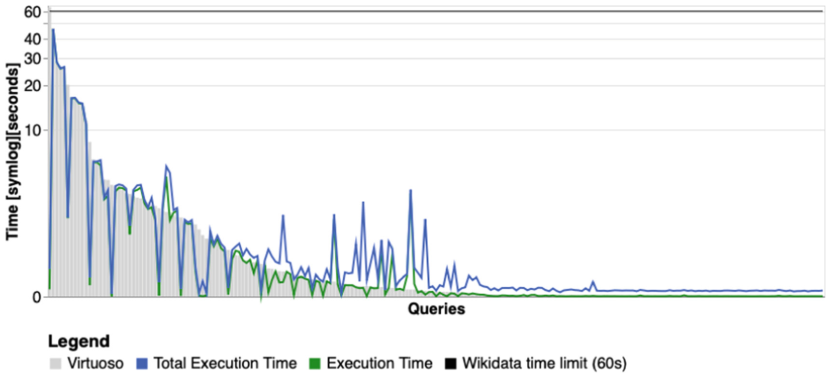


Fig. 4. Average execution time on BlazeGraph (top) and Virtuoso (bottom) with a budget of 1000 random walks and an exploration depth limited to 5.

First, Virtuoso is much faster than BlazeGraph on this workload. While BlazeGraph requires more than 20270 s, Virtuoso only needs 419 s. Moreover, BlazeGraph is not able to execute 24 queries. For the first 15 queries, BlazeGraph reaches the time-out set to 900 s in our experiments. An Out-Of-Memory exception occurs for the nine remaining queries. Note that queries that time-out account for 900 s in the total execution time, while queries that result in an error account for 0. On its side, Virtuoso processes all 213 queries. Despite being recommended when there are join ordering issues, the RTO optimizer delivers the worst results. Given a SPARQL query Q, to estimate the cost of a join order J, RTO samples the first triple pattern in J and executes Q^J on this sample. Unfortunately, evaluating Q^J, even on a small sample, can take a very long time. As a result, many queries time-out or crash because of the optimization step. In the end, our proposal outperforms both BlazeGraph and Virtuoso. The workload total execution time is divided by at least 14 on BlazeGraph. It can be much more depending on the queries that time-out. Compared to Virtuoso, we observe a 14% improvement on the execution time.

Table 3. Global results of our experiments. For each configuration, queries have been optimized 5 times and executed 3 times after a warmup execution. DL=Depth Limit, TET=Total Execution Time (secs), ET=Execution Time (secs), OT=Optimization Time (secs), T=Timeouts, E=Errors.

		Walks	DL	TET	ET	OT	T	E
Blaze Graph	Default			20270			15	9
	RTO			35107			36	64
	Proposal	1000	10	1635 (± 195)	1577 (± 195)	58 (± 2)	0	0
			5	1429 (± 130)	1383 (± 128)	37 (± 14)	0	0
			3	1629 (± 207)	1583 (± 206)	46 (± 4)	0	0
			1	2046 (± 219)	2006 (± 222)	40 (± 13)	0	0
		10000	10	1890 (± 286)	1530 (± 288)	361 (± 4)	0	0
			5	1545 (± 134)	1226 (± 127)	319 (± 14)	0	0
			3	1583 (± 206)	1290 (± 206)	293 (± 8)	0	0
			1	2087 (± 15)	1847 (± 11)	241 (± 6)	0	0
Virtuoso	Default			419	415	4	0	0
	Proposal	1000	10	365 (± 18)	317 (± 18)	48 (± 0)	0	0
			5	362 (± 11)	319 (± 11)	43 (± 0)	0	0
			3	356 (± 29)	317 (± 29)	39 (± 0)	0	0
			1	346 (± 5)	315 (± 5)	31 (± 0)	0	0
		10000	10	613 (± 17)	300 (± 12)	314 (± 9)	0	0
			5	594 (± 15)	303 (± 15)	291 (± 1)	0	0
			3	570 (± 16)	301 (± 35)	269 (± 1)	0	0
			1	528 (± 3)	308 (± 3)	220 (± 0)	0	0

What is the Impact of Our Approach on Each Query? Figure 4 presents a per-query view of the results summarized in Table 2. Queries are ordered on the x-axis according to the total execution time of the baseline. Focusing first on BlazeGraph, the 15 queries that exceed the time limit are depicted in dark gray, while the nine queries that result in an error are in red. As depicted by the blue curve in Fig. 4, our optimizer makes the difference on the long-running queries by finding better join orders. When looking at the optimization time, i.e. the ratio between the blue and green curves, they are irregularities. It comes from the HDT storage that cannot draw a random triple in constant time using the POS index. We can draw the same conclusion on Virtuoso. For short-running queries, the generated join orders are close to Virtuoso. However, long-running queries can benefit from significant improvements. For instance, the longest query on Virtuoso takes 63 s to complete. Using our optimizer we are able to find a join order that reduces the execution time to 110 ms.

How do the Budget k ***and The Depth Exploration Limit*** $dMax$ ***Impact Performance?*** In our approach, two parameters impact performance; the budget k, i.e. the number of random walks used to estimate the cardinality of joins, and the limit $dMax$ on the exploration depth for property paths. To measure the impact of these two parameters, we tested different configurations that are summarized in Table 3.

First, let us focus on the execution time. As expected, given $dMax$, increasing k systematically leads to better performance. Moreover, increasing the budget tends to decrease the variance between measurements, i.e. estimates become more reliable. However, despite multiplying the number of random walks by ten, the gain in terms of execution time is not that large, especially on Virtuoso. When using a bottom-up approach (as a DP algorithm), the quality of join orders mainly depends on the quality of the first joins, as highlighted in [18]. Thus, accuracy on 1-way, 2-way, or 3-way joins is often enough to get good join orders and does not require a large budget.

If increasing the budget always leads to better performance, increasing $dMax$ may negatively impact the execution time. Random walks are uniformly distributed over the interval $1..dMax$. The larger the interval is, the fewer walks remain to estimate each part, i.e. the more inaccurate the estimator is. Consequently, with a small budget, it is better to reduce $dMax$ to have more accurate cardinalities. Even if property path patterns may be underestimated (because they are not fully explored), estimates will be more reliable. However, with a larger budget, it is worth looking for a larger $dMax$ to better capture the real cost of property path patterns. The budget k being the most impacting factor in optimization time, a good strategy to define k and $dMax$ is to define a budget first, and then select $dMax$ by testing different values until the quality of join orders deteriorates. For instance, with a budget of 1000 random walks on BlazeGraph, setting $dMax = 5$ results in good performance, but increasing $dMax$ to 10 starts deteriorating the execution time.

6 Related Works

Different approaches have been proposed to speed up the evaluation of SPARQL property path queries [2]. Some approaches rely on indexes to improve the evaluation of property path patterns. For instance, [28] proposes an index named FERRARI that encodes transitive closures into a compact representation. This index is then used in RDF-3X [11] to evaluate property path queries efficiently. In [4], authors combine a novel index to represent sets of triples with a new algorithm based on the Glushkov automaton. Although using indexes can drastically improve property path patterns execution, maintaining them on large and dynamic knowledge graphs is costly [27]. This paper requires no additional indexes, only those currently used in SPARQL engines.

Other approaches rely on innovative property path pattern operators [1,4,27,32,33]. For instance, to find the best strategy to evaluate a property pattern, Waveguide [33] introduces a new operator that mixes idea from relational and graph databases. To improve the execution time of property path queries, [27] relies on an approximate operator to compute, with a fixed error rate, the reachability between two nodes. All these approaches are really exciting but focus only on evaluating property path patterns alone. This paper focuses on optimizing property path queries, where a property path pattern is just one part of a query. By choosing the proper join orders, queries execution time can be significantly improved without changing the underlying engines.

Closer to our approach, [14] focuses on the optimization of conjunctive regular path queries, i.e. not just property path patterns. In [14], the authors propose a new algebra, with a set of rewriting rules, allowing optimizers to explore query plans that could not be considered before. While their approach consists in enriching the search space of query plans, we propose a solution to accuratly compare the different plans in order to choose the best one. Thus both approaches can be used together.

The importance of finding good join orders for property path queries has already been pointed out in RDF-3X [10,11]. However, to estimate the cardinality of property path patterns, the FERRARI index is used. Other approaches based on synopses or statistics can also be used to estimate cardinalities, such as characteristic sets [22] or SumRDF [29]. However, they suffer from the same problem as index-based approaches; synopses and statistics need to be maintained. A well-known alternative is sampling. Sampling allows gathering information when no indexes or statistics are available. For instance, the RTO engine of BlazeGraph relies on sampling, while Virtuoso commonly uses sampling to estimate the selectivity of filters. One drawback of using sampling to estimate cardinalities is its cost. However, [18] demonstrates that sampling can be both accurate and cheap when using traditional index structures such as existing B-Tree indexes. Following the same basic principles, WanderJoin [19] uses random walks to evaluate aggregate queries, and can be used as a cardinality estimator. As demonstrated in [23], WanderJoin outperforms other cardinality estimators. Compared to [18] and [19], our cardinality estimator handles property path queries.

7 Conclusions and Future Work

This paper introduces a new query optimizer that relies on the relaxation of reachability-patterns and a sampling-based cardinality estimator to find efficient join orders for $C2RPQ_{UF}$ queries. The experimental study demonstrates that our proposal outperforms existing engines. On the newly proposed Wikidata Query Benchmark, the workload execution time is divided by 14 compared to BlazeGraph. This work opens several perspectives. First, optimization times can be significantly improved using a better budget model such as the one defined in [18]. Moreover, computing the confidence interval of estimators [19] may allow us to adapt the budget to each join order, rather than systematically computing k random walks. Generalizing our approach to nested stars is also part of our research agenda. Finally, this paper relies on random walks to estimate the cardinality of $C2RPQ_{UF}$ queries. Another exciting line of research would be to study how random walks can be used to estimate the cardinality of SPARQL queries in the presence of the MINUS, OPTIONAL and FILTER NOT EXISTS operators.

Acknowledgments. This work is supported by the ANR project DeKaloG (Decentralized Knowledge Graphs), ANR-19-CE23-0014, CE23 - Intelligence artificielle, and the CominLabs project MikroLog (The Microdata Knowledge Graph).

References

1. Aimonier-Davat, J., Skaf-Molli, H., Molli, P.: Processing SPARQL property path queries online with web preemption. In: Verborgh, R., et al. (eds.) ESWC 2021. LNCS, vol. 12731, pp. 57–72. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77385-4_4
2. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.C.N.: A survey of RDF stores & SPARQL engines for querying knowledge graphs. VLDB J., 1–26 (2021)
3. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench: a wikidata graph query benchmark. In: Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D., et al. (eds.) The Semantic Web—ISWC 2022. ISWC 2022. Lecture Notes in Computer Science, vol. 13489, pp. 714–731. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19433-7_41
4. Arroyuelo, D., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Time-and space-efficient regular path queries. In: 38th International Conference on Data Engineering (ICDE), pp. 3091–3105. IEEE (2022)
5. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of wikidata query logs. In: The World Wide Web Conference, pp. 127–138 (2019)
6. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.-Y.: SPARQL web-querying infrastructure: ready for action? In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8219, pp. 277–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41338-4_18
7. Cluet, S., Moerkotte, G.: On the complexity of generating optimal left-deep processing trees with cross products. In: Gottlob, G., Vardi, M.Y. (eds.) ICDT 1995. LNCS, vol. 893, pp. 54–67. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-58907-4_6
8. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: n: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) Networked Knowledge - Networked Media. Studies in Computational Intelligence, vol. 221, pp. 7–24. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02184-8_2
9. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). J. Web Seman. **19**, 22–41 (2013)
10. Gubichev, A.: Query processing and optimization in graph databases. Ph.D. thesis, Technische Universität München (2015)
11. Gubichev, A., Bedathur, S.J., Seufert, S.: Sparqling kleene: fast property paths in RDF-3x. In: First International Workshop on Graph Data Management Experiences and Systems, pp. 1–7 (2013)
12. Gubichev, A., Neumann, T.: Exploiting the query structure for efficient join ordering in SPARQL queries. In: 17th International Conference on Extending Database Technology, EDBT (2014)
13. Hertzschuch, A., Hartmann, C., Habich, D., Lehner, W.: Simplicity done right for join ordering. In: CIDR (2021)
14. Jachiet, L., Genevès, P., Gesbert, N., Layaïda, N.: On the optimization of recursive relational queries: application to graph queries. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 681–697 (2020)

15. Kader, R.A., Boncz, P.A., Manegold, S., van Keulen, M.: ROX: run-time optimization of XQueries. In: Çetintemel, U., Zdonik, S.B., Kossmann, D., Tatbul, N. (eds.) International Conference on Management of Data, SIGMOD. ACM (2009)
16. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: SPARQL with property paths. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_1
17. Leis, V., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: How good are query optimizers, really? VLDB Endow. **9**(3), 204–215 (2015)
18. Leis, V., Radke, B., Gubichev, A., Kemper, A., Neumann, T.: Cardinality estimation done right: Index-based join sampling. In: CIDR (2017)
19. Li, F., Wu, B., Yi, K., Zhao, Z.: Wander join and XDB: online aggregation via random walks. ACM Trans. Database Syst. **44**(1), 1–41 (2019). https://doi.org/10.1145/3284551
20. Losemann, K., Martens, W.: The complexity of regular expressions and property paths in SPARQL. ACM Trans. Database Syst. (TODS) **38**(4), 1–39 (2013)
21. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of Wikidata: semantic technology usage in Wikipedia's knowledge graph. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11137, pp. 376–394. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_23
22. Neumann, T., Moerkotte, G.: Characteristic sets: accurate cardinality estimation for RDF queries with multiple joins. In: 27th International Conference on Data Engineering. IEEE (2011)
23. Park, Y., Ko, S., Bhowmick, S.S., Kim, K., Hong, K., Han, W.S.: G-care: a framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In: International Conference on Management of Data (SIGMOD) (2020)
24. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 1–45 (2009)
25. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Database Theory - ICDT 2010, pp. 4–33 (2010)
26. Selingerl, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access path selection in a relational database management system. In: ACM SIGMOD (1979)
27. Sengupta, N., Bagchi, A., Ramanath, M., Bedathur, S.: Arrow: approximating reachability using random walks over web-scale graphs. In: International Conference on Data Engineering (ICDE), pp. 470–481. IEEE (2019)
28. Seufert, S., Anand, A., Bedathur, S., Weikum, G.: Ferrari: flexible and efficient reachability range assignment for graph indexing. In: 29th International Conference on Data Engineering (ICDE), pp. 1009–1020. IEEE (2013)
29. Stefanoni, G., Motik, B., Kostylev, E.V.: Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In: The World Wide Web Conference, pp. 1043–1052 (2018)
30. Steve, H., Andy, S.: SPARQL 1.1 query language. In: Recommendation W3C (2013)
31. Thompson, B., Personick, M., Cutcher, M.: The bigdata® RDF graph database. In: Linked Data Management, pp. 221–266. Chapman and Hall/CRC, Boca Raton (2016)
32. Wadhwa, S., Prasad, A., Ranu, S., Bagchi, A., Bedathur, S.: Efficiently answering regular simple path queries on large labeled networks. In: International Conference on Management of Data, pp. 1463–1480 (2019)
33. Yakovets, N., Godfrey, P., Gryz, J.: Query planning for evaluating SPARQL property paths. In: International Conference on Management of Data, pp. 1875–1889 (2016)