NPCS: Native Provenance Computation for SPARQL

Zubaria Asma
asma@ics.forth.gr
FORTH-ICS and University of Crete
Heraklion, Crete, Greece

Daniel Hernández
daniel.hernandez@ki.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Luis Galárraga
luis.galarraga@inria.fr
Inria
Rennes, France

Giorgos Flouris
fgeo@ics.forth.gr
FORTH-ICS
Heraklion, Crete, Greece

Irini Fundulaki
fundul@ics.forth.gr
FORTH-ICS
Heraklion, Crete, Greece

Katja Hose
katja.hose@tuwien.ac.at
TU Wien
Wien, Austria

ABSTRACT

The popularity of Knowledge Graphs (KGs) both in industry and academia owes credit to their flexible data model, suitable for data integration from multiple sources. Several KG-based applications such as trust assessment or view maintenance on dynamic data rely on the ability to compute provenance explanations for query results. The how-provenance of a query result is an expression that encodes the records (triples or facts) that explain its inclusion in the result set. This article proposes NPCS, a Native Provenance Computation approach for SPARQL queries. NPCS annotates query results with their how-provenance. By building upon spm-provenance semirings, NPCS supports both monotonic and non-monotonic SPARQL queries. Thanks to its reliance on query rewriting techniques, the approach is directly applicable to already deployed SPARQL engines using different reification schemes – including RDF-star. Our experimental evaluation on two popular SPARQL engines (GraphDB and Stardog) shows that our novel query rewriting brings a significant runtime improvement over existing query rewriting solutions, scaling to RDF graphs with billions of triples.

CCS CONCEPTS

• **Theory of computation** → **Data provenance**; • **Information systems** → **World Wide Web**; **Database query processing**.

KEYWORDS

Knowledge graphs, SPARQL, RDF, how-provenance, data provenance

ACM Reference Format:
Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irini Fundulaki, and Katja Hose. 2024. NPCS: Native Provenance Computation for SPARQL. In *Proceedings of the ACM Web Conference 2024 (WWW '24), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3589334.3645557

WWW '24, May 13–17, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0171-9/24/05.
https://doi.org/10.1145/3589334.3645557

1 INTRODUCTION

Thanks to the continuous advances in information extraction and knowledge graph construction, the Web nowadays enjoys from a plethora of machine-readable data, structured in large RDF knowledge graphs (KGs). These KGs, often queried via SPARQL endpoints, allow computers to "understand" the real world. They do so by encoding knowledge as collections of triples or statements (s, p, o) with subject s, predicate p, and object o, e.g., $(UK, capital, London)$. A triple (s, p, o) is also a directed p-labeled edge from node s to node o. This data model serves as the foundation for multiple applications such as question answering, Web search, and smart assistants.

Given the heterogeneity of data sources that contribute to modern KGs, the problem of identifying the *provenance* of query results is central for many KG-based tasks. The provenance of a query result is an expression that encodes the lineage of data transformations and statements that contributed to that result. Provenance is of great value for KG providers because it streamlines maintenance tasks such as source selection and view maintenance. For data consumers, query provenance serves as an explanation for answers. This can be pivotal in use cases that need to assess data reliability, or manage access control, trustworthiness, and data quality.

Among the existing formalisms to model provenance, *how-provenance* is the most expressive [15]. In this model, the provenance of a query result is an algebraic expression in a *provenance semiring*. Consider, for instance, the following KG,

$$\{u_1\colon (UK, capital, London), u_2\colon (London, in, UK), u_3\colon (London, a, City)\},$$

and the SPARQL query

$$\text{SELECT } ?x \text{ WHERE } \{\ \{UK\ capital\ ?x\}\ \text{UNION}\ \{?x\ in\ UK; a\ City\}\ \}.$$

The solution to this query is *London*. How-provenance explains the presence of *London* in the result set with the polynomial expression $u_1 \oplus (u_2 \otimes u_3)$. This polynomial tells us that there are two ways to get London as a solution: either via u_1, or via the conjunction of u_2 and u_3. There exist different algebraic structures for provenance in the literature [8, 12, 15], but this paper considers spm-semirings [12] because they are designed for the semantics of SPARQL, including its non-monotonic fragment. We highlight that query provenance assumes the availability of identifiers for triples, in other words, it assumes that the KG has been reified using some scheme. Examples of reification schemes are RDF-star and named graphs.

There are essentially two main strategies to compute how-provenance for query results. Methods, such as TripleProv [22], opt for customized engines designed to compute provenance along

query evaluation. Since provenance support is embedded in the engine, such solutions allow for advanced optimizations. On the downside, customized engines are not applicable to already deployed SPARQL endpoints. The other alternative is query rewriting [18]. By this logic, SPARQL queries are rewritten so that the new query retrieves both the query solutions and the polynomials describing their provenance, potentially with some post-processing. While rewriting the query induces a runtime overhead, this design provides the flexibility to be applied to any SPARQL endpoint on the Web.

With this in mind, we propose NPCS, a new query rewriting method for how-provenance computation in RDF/SPARQL engines. Unlike previous approaches [18], NPCS does not require any post-processing to compute query solutions with how-provenance annotations. This makes NPCS the first fully native SPARQL solution for how-provenance. Moreover, NPCS supports different data reification schemes, including RDF-star. Our experimental evaluation on both synthetic and real data suggests that our fully native SPARQL rewriting (a) incurs a reasonable runtime overhead, and (b) it is consistently faster than SPARQLprov [18], the state of the art in how-provenance in SPARQL.

The rest of the paper is structured as follows. In Sections 2 and 3, we provide the basic background for this work in terms of related work and preliminaries, respectively. In Section 4 we describe our rewriting method in detail, whereas Section 5 provides a comprehensive evaluation of the performance of our approach. We wrap up the paper and discuss future work in Section 6.

2 RELATED WORK

We survey the literature on provenance for query results along two axis: provenance models (Section 2.1), and provenance support for RDF/SPARQL engines (Section 2.2).

2.1 Semirings and Provenance Models

Semirings were first used to model query provenance in the groundbreaking work by Green et al. [15]. This work proposed *commutative semirings* to annotate query results for selection, projection, join, and union queries for Datalog and the positive fragment of relational algebra. Commutative semirings cannot model provenance for non-monotonic operators such as the left-outer join and the difference [15], hence the algebraic structures were expanded to include a monus operator[1] that accounts for the relational difference [11]. Commutative semirings and their extensions model provenance as polynomial expressions. These expressions, called *how-provenance*, encode both the sources and the data transformations required to obtain (and sometimes exclude) a particular query answer. How-provenance is more expressive than other provenance models such as lineage [7] or why-provenance [6].

Damasio et al. [8] showed that by rewriting SPARQL queries into relational algebra, we can provide provenance annotations for SPARQL queries using *m-semirings*. However, Geerts et al. [12] show that these can yield very long and complex provenance expressions, and thus developed the *spm-semirings* formalism (spm stands for SPARQL Minus) to overcome these limitations. Spm-semirings guarantee more compact explanations and offer native support for non-monotonic SPARQL operators such as OPTIONAL and MINUS. Since how-provenance polynomials are abstract annotations, they are useful to a handful of metadata management applications [9, 16] via the notion of commutation with homomorphisms.

[1]Do not confuse monus with minus (the SPARQL operator). A monus operator is an operator on certain commutative monoids that are not groups (see [11]).

2.2 Provenance-supported SPARQL engines

Wylot et al. [22] introduced TripleProv, a system to compute provenance annotations in the commutative semiring framework for queries with basic graph patterns, union, and the OPTIONAL operator. Since its reliance on commutative semirings, TripleProv cannot guarantee commutation with homomorphisms for queries involving the non-monotonic OPTIONAL operator. Additionally, TripleProv uses a customized engine that organizes data into molecules—sort of indexes for star patterns—, and thus it cannot be used on already deployed SPARQL engines. This is why our approach resorts to query rewriting for how-provenance computation.

But approaches based on query rewriting are not rare at all. Perm [14] and GProM [3] are two examples. Such approaches are tailored for relational databases. Hence, they are not applicable to SPARQL queries out of the box. Similarly to TripleProv, none of these methods can properly support non-monotonic SPARQL queries because Perm is based on the lineage model, and GProM relies on commutative semirings.

While the work of Geerts et al. [12] was the first to study provenance for the non-monotonic fragment of SPARQL, the first concrete method to compute how-provenance under the spm-semiring formalism was proposed by Hernandez et al. [18]. They introduced SPARQLprov, a method based on query rewriting that can annotate query results with how-provenance polynomials for both monotonic and non-monotonic queries. Contrary to NPCS, SPARQLprov is not a 100% SPARQL solution because it relies on a subsequent decoding phase to compute the final provenance annotations from the results of the rewritten query. As our experimental evaluation shows, this decoding phase can incur prohibitive runtime overheads for non-selective queries.

3 PRELIMINARIES

3.1 RDF-star and SPARQL-star

The following definition follows the W3C Community Group Draft [17]. We assume the existence of three (pairwise disjoint) countably infinite sets: the set of IRIs I, the set of blank nodes B, and the set of literals L. An RDF triple $t = (s, p, o) \in T$, where $T = (I \cup B) \times I \times (I \cup B \cup L)$, is a statement that consists of a *subject* s, a *predicate* p, and an *object* o. An RDF graph G is a set of RDF triples. The RDF-star data model extends RDF by allowing arbitrarily deep nesting of triples as subject or object arguments:

Definition 3.1 (RDF-star). An RDF-star triple is a 3-tuple defined recursively as follows:

(1) Any RDF triple $t \in T$ is an RDF-star triple; and
(2) Given RDF-star triples t and t', and RDF terms $s \in (I \cup B)$, $p \in I$, and $o \in (I \cup B \cup L)$, then the triples (t, p, o), (s, p, t), and (t, p, t') are also RDF-star triples.

An RDF-star graph G is a set of RDF-star triples.

In a nutshell, RDF-star allows us to "say things" about statements, which endows RDF with native *reification* capabilities. This is crucial when computing how-provenance for query results because query provenance builds upon identifiers for triples in the graph.

RDF graphs can be queried using the SPARQL language. The fundamental building blocks of SPARQL queries are basic graph patterns (BGPs). They are sets of tuples of the form $(s, p, o) \in (V \cup I \cup B) \times (V \cup I) \times (V \cup I \cup B \cup L)$, where V is a countably infinite set of variables—prefixed by the character '?' and disjoint with $I \cup B \cup L$. Analogously to RDF-star, SPARQL-star extends BGPs by allowing nested patterns. BGPs in SPARQL queries are combined with operators such as AND, UNION or SELECT. A (solution) *mapping* is a partial function $\mu : V \to (T \cup I \cup B \cup L)$ where the domain of μ, denoted by $\mathrm{dom}(\mu)$, is a finite set of variables. We write $\mathrm{inScope}(Q)$ for the set of variables, called *in-scope*, that can occur in Q answers, and variables that always occur are called *strongly bound* [4]. The details of SPARQL evaluation semantics are detailed in [2, 21]. In short, the evaluation of a SPARQL query Q on an RDF graph G is defined as a function $[\![Q]\!]_G$, which returns a multiset of mappings. Our goal is to annotate those mappings with their how-provenance.

3.2 How-provenance in SPARQL

3.2.1 Semirings. A *commutative monoid* $\mathcal{M}$ is an algebraic structure $(M, +_{\mathcal{M}}, 0_{\mathcal{M}})$ such that $M \neq \emptyset$ is a set closed under a commutative and associate binary operation $+_{\mathcal{M}}$. The element $0_{\mathcal{M}}$ is the identity operand for $+_{\mathcal{M}}$. Given two commutative monoids $(K, +_{\mathcal{K}}, 0_{\mathcal{K}})$ and $(K, \times_{\mathcal{K}}, 1_{\mathcal{K}})$ such that $\times_{\mathcal{K}}$ is distributive over $+_{\mathcal{K}}$, and $0_{\mathcal{K}} \times_{\mathcal{K}} x = 0_{\mathcal{K}}$ (for every $x \in K$), we call the structure $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ a *commutative semiring*. An *spm-semiring* $(K, +_{\mathcal{K}}, \times_{\mathcal{K}}, -_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ extends a commutative semiring with a minus operation $-_{\mathcal{K}}$. This operator follows a set of axioms that allows us to model non-monotonic operations such as the relational difference. For more details about spm-semirings, we refer the reader to [13]. The algebraic expressions within an spm-semiring are used to annotate query solutions. To see how, we need to introduce the concepts of $\mathcal{K}$-relations and $\mathcal{K}$-graphs.

3.2.2 $\mathcal{K}$-relations and $\mathcal{K}$-graphs. Given a set A, an spm-semiring $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, -_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$, and a function $f : A \to K$ with a finite support set $\mathrm{supp}(f) = \{\mu \in \mathbf{M} \mid f(\mu) \neq 0_{\mathcal{K}}\}$ is called a *$\mathcal{K}$-set* over A. We call $f(\mu)$ the *$\mathcal{K}$-value* of $\mu \in U$. A $\mathcal{K}$-set Ω over the set of all possible SPARQL mappings is called a $\mathcal{K}$-relation, and a $\mathcal{K}$-set Γ over all possible RDF triples is called *$\mathcal{K}$-graph*.

Example 3.2. Let $\mathcal{K} = (K, \oplus, \otimes, \ominus, 0, 1)$ be an spm-semiring with $K = \{u_1, u_2, u_3\}$, and let G and Q be the following RDF-star graph and query:

$$G = \{\ ((Alice, likes, pasta), wasDerivedFrom, u_1),\ ((Alice, likes, pasta), wasDerivedFrom, u_2),\ ((Alice, livesIn, Italy), wasDerivedFrom, u_3)\ \},$$

$$Q = (?x, likes, pasta)\ \text{AND}\ (?x, livesIn, Italy).$$

It is easy to see that the predicate *wasDerivedFrom* defines a $\mathcal{K}$-graph Γ where the first and last triples are associated to $u_1 \oplus u_2$ and u_3, and that $\mu = \{?x \to Alice\}$ is a solution mapping for Q. The set $\{\mu \to (u_1 \oplus u_2) \otimes u_3\}$ is a $\mathcal{K}$-relation that associates Q's solution to a provenance polynomial. This how-provenance annotation tells us that *Alice* is a query solution for Q as long as the triple identified by u_3 is present in conjunction with either the triples u_1 or u_2. We call those provenance identifiers the *sources*.

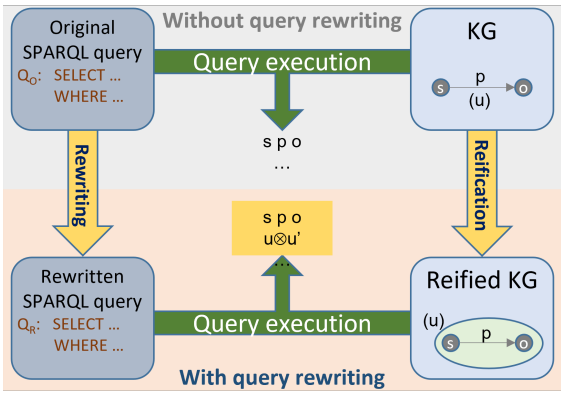

Figure 1: The query rewriting process for NPCS

Definition 3.3. Given an spm-semiring $\mathcal{K}$, a SPARQL query Q consisting of a combination of triple patterns with the operators AND, UNION, DIFF, FILTER, OPTIONAL, and SELECT, and a $\mathcal{K}$-graph Γ, we write $(\!|Q|\!)_\Gamma$ to denote the $\mathcal{K}$-relation defined recursively as follows:

$$(\!|(s,p,o)|\!)_\Gamma(\mu) = \Gamma(\mu(s,p,o)),$$
$$(\!|\text{SELECT } W \text{ WHERE } Q|\!)_\Gamma(\mu) = \textstyle\sum_{\mu' : \mu'|_W = \mu} (\!|Q|\!)_\Gamma(\mu'),$$
$$(\!|Q \text{ FILTER } \varphi|\!)_\Gamma(\mu) = (\!|Q|\!)_\Gamma(\mu) \times_{\mathcal{K}} 1_{\mu \models \varphi},$$
$$(\!|Q_1 \text{ UNION } Q_2|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) +_{\mathcal{K}} (\!|Q_2|\!)_\Gamma(\mu),$$
$$(\!|Q_1 \text{ AND } Q_2|\!)_\Gamma(\mu) = \textstyle\sum_{\mu = \mu_1 \cup \mu_2} ((\!|Q_1|\!)_\Gamma(\mu_1) \times_{\mathcal{K}} (\!|Q_2|\!)_\Gamma(\mu_2)),$$
$$(\!|Q_1 \text{ DIFF } Q_2|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) -_{\mathcal{K}} (\textstyle\sum_{\mu' \sim \mu} (\!|Q_2|\!)_\Gamma(\mu')),$$
$$(\!|Q_1 \text{ OPTIONAL } Q_2|\!)_\Gamma(\mu) = (\!|Q_1 \text{ AND } Q_2|\!)_\Gamma(\mu) +_{\mathcal{K}} (\!|Q_1 \text{ DIFF } Q_2|\!)_\Gamma(\mu),$$

where $\sum$ denotes sums using the operation $+_{\mathcal{K}}$, $\sim$ denotes mapping compatibility, and $\mu'|_W$ is the projection of mapping μ' on the variables in W. Two mappings μ and μ' are compatible if $\mu(?x) = \mu'(?x)$ for every variable $?x \in \mathrm{dom}(\mu) \cap \mathrm{dom}(\mu')$.

4 PROVENANCE COMPUTATION WITH NPCS

In the following, we explain our method to annotate SPARQL query solutions with how-provenance annotations as in Example 3.2. Given a $\mathcal{K}$-graph Γ, its corresponding RDF-star graph G, and a SPARQL query Q, NPCS rewrites Q into Q' so that $[\![Q']\!]_G$ is a set of mappings with an additional variable encoding the provenance polynomials. Thus, the output of Q' represents the $\mathbb{N}[K]$-relation $[\![Q]\!]_\Gamma$. Those polynomials lie in an spm-semiring $(\mathbb{N}[K], \oplus, \otimes, \ominus, 0, 1)$, where K is the set of triple identifiers in G. We highlight that computing provenance assumes that the triples in the graph are reified, i.e., identified. RDF-star is a natural way to do it, but as shown later, our approach supports any reification scheme for RDF data. The NPCS architecture is depicted in Figure 1.

4.1 Our Query Rewriting in a Nutshell

Consider the graphs Γ and G and query Q from Example 3.2. For pedagogical reasons, we rewrite our query Q as P_1 AND P_2. We recall

that our goal is to return the following result set:

$$\left[\begin{array}{c|c} ?x & ?z \\ \hline Alice & (u_1 \oplus u_2) \otimes u_3 \end{array}\right].$$

The column labeled $?z$ stores the how-provenance of each of the query solutions. To compute such an expression, our strategy must rewrite the query such that the rewritten query retrieves the identifiers of the triples that match each of the triple patterns. However, this is not enough. For instance, the triple pattern $P_1 = (?x, \mathit{likes}, \mathit{pasta})$ has two matches, i.e., u_1 and u_2, that must be *grouped* into the expression $(u_1 \oplus u_2)$. This term tells us that the presence of at least one of those sources guarantees the inclusion of *Alice* in the result set. Finally, the groups extracted from each of the triple patterns must be combined with the $\otimes$ operator that explains the semantics of AND. We argue that to obtain the left-hand term of the product we can rewrite P_1 into the following sub-query:

$$\begin{array}{lll} P_1' = (\text{SELECT} & ?x \ (\text{ProvAggSum}(?z{\oplus}{\otimes}1{\oplus}{\odot}) \ \text{AS} \ ?z{\oplus}{\otimes}1) \\ \quad \text{WHERE} & \text{Reify}((?x, \mathit{likes}, \mathit{pasta}), ?z{\oplus}{\otimes}1{\oplus}{\odot}) \\ \quad \text{GROUP BY} & ?x), \end{array}$$

The Reify function rewrites a triple pattern so that it matches the reification scheme used to encode the $\mathcal{K}$-graph Γ as an RDF-star graph G. In our running example, Reify is a shortcut for

$$((s, p, o), \mathit{wasDerivedFrom}, ?z{\oplus}{\otimes}1{\oplus}{\odot}).$$

The intermediate variable $?z{\oplus}{\otimes}1{\oplus}{\odot}$ is introduced to capture the sources that match P_1, whereas $?z{\oplus}{\otimes}1$ groups all the sources associated to a query solution, which explains the group clause on $?x$. The function ProvAggSum, later explained, combines the different sources into a summation with the operator $\oplus$.

The signs $\oplus$, $\otimes$, and $\odot$ in the intermediate variable names are strings used to produce new variable names that do not clash with the original query variables. The names of the variables encode the different steps of the construction of the annotations. For instance, the sign $\odot$ at the end of a variable name tells us that the variable's bindings are triple identifiers. If this is followed by a $\oplus$ sign, then those bindings will eventually be grouped into a summation by a subsequent step. Those results will be stored in a variable with the same prefix but without the suffix $\oplus\odot$. Furthermore, the $\otimes 1$ sign tells us that our results correspond to the first operand of a join operation, namely AND in SPARQL. If we apply the same logic to P_2 our rewriting for Q takes the following form:

$$\begin{array}{lll} Q' = (\ \text{SELECT} & ?x \ (\text{ProvAggSum}(?z{\oplus}) \ \text{AS} \ ?z) \\ \quad \text{WHERE} & ((P_1' \ \text{AND} \ P_2') \\ & \quad \text{BIND} \ (\text{ProvProd}(?z{\oplus}{\otimes}1, ?z{\oplus}{\otimes}2) \ \text{AS} \ ?z{\oplus})) \\ \quad \text{GROUP BY} & ?x\ ). \end{array}$$

The operation ProvProd combines the expressions derived from the product's operands. We resort again to ProvAggSum to sum up all the ways to produce a solution mapping from a join operation. The operators ProvAggSum and ProvProd are defined in terms of the built-in SPARQL functions as follows.

Definition 4.1. Let $?x$, $?x_1$,…, $?x_n$ be variables. Then, we define the following SPARQL operators[2]:

$\text{ProvAggSum}(?x) = \texttt{concat}(\text{“ (}{\oplus}\text{)”}, \texttt{aggregate\_concat}(?x), \text{“)”})$,
$\text{ProvProd}(?x_1, \ldots, ?x_n) = \texttt{concat}(\text{“ (}{\otimes}\text{)”}, ?x_1, \ldots, ?x_n, \text{“)”})$,
$\text{ProvDiff}(?x_1, ?x_2) = \texttt{concat}(\text{“ (}{\ominus}\text{)”}, ?x_1, ?x_2, \text{“)”})$.

4.2 Base Rewriting Rules

Having provided the intuition behind our query rewriting in Section 4.1, we now introduce our rewriting rules for arbitrary SPARQL queries.

Definition 4.2 (Base SPARQL-star query rewriting). Let Q be a SPARQL query, $?z$ a variable, Reify a reification scheme, and $\text{inScope}(Q)$ a function that returns the variables that can be in the solution mappings of query Q after execution[3]. Then, the rewritten query for Q and variable $?z$ over scheme Reify, denoted $\beta(Q, ?z)$, is defined recursively as follows:

(1) If Q is an empty basic graph pattern, then $\beta(Q, z)$ is the query $\{\}$ BIND $(1$ AS $?z)$.
(2) If Q is a triple pattern (s, p, o), then $\beta(Q, z)$ is the query
(SELECT inScope(Q) (ProvAggSum($?z{\oplus}{\odot}$) AS $?z$)
WHERE Reify($(s, p, o), ?z{\oplus}{\odot}$)
GROUP BY inScope(Q)).
(3) If Q is $(Q_1$ AND $Q_2)$, then $\beta(Q, ?z)$ is the query
(SELECT inScope(Q) (ProvAggSum($?z{\oplus}$) AS $?z$)
WHERE $(\beta(Q_1, ?z{\oplus}{\otimes}1)$ AND $\beta(Q_2, ?z{\oplus}{\otimes}2))$
BIND (ProvProd($?z{\oplus}{\otimes}1, ?z{\oplus}{\otimes}2$) AS $?z{\oplus}$)
GROUP BY inScope(Q)).
(4) If Q is $(P_1$ UNION $P_2)$, then $\beta(Q, ?z)$ is the query
(SELECT inScope(Q) (ProvAggSum($?z{\oplus}$) AS $?z$)
WHERE $(\beta(Q_1, ?z{\oplus})$ UNION $\beta(Q_2, ?z{\oplus}))$
GROUP BY inScope(Q)).
(5) If Q is $(Q_1$ DIFF $Q_2)$, then let ν a variable substitution that substitutes with fresh variables the variables in $\text{dom}(Q_1) \cap \text{dom}(Q_2)$ that are not strongly bound in Q_1. Then, $\beta(Q, ?z)$ is the query computed as follows.
(SELECT inScope(Q)
(ProvDiff($z{\ominus}1$, ProvAggSum($z{\ominus}2{\oplus}$)) AS $?z$)
WHERE $(\beta(Q_1, ?z{\ominus}1)$ OPTIONAL$_{C_\nu}$ $\beta(\nu(Q_2), ?z{\ominus}2{\oplus}))$
GROUP BY inScope(Q) $\cup$ $\{z{\ominus}1\}$).
(6) If Q is (SELECT W WHERE Q'), then $\beta(Q, ?z)$ is the query
(SELECT W (ProvAggSum($?z{\oplus}$) AS $?z$)
WHERE $\beta(Q', ?z{\oplus})$
GROUP BY W).
(7) If Q is $(Q'$ FILTER $\varphi)$, then $\beta(Q, ?z)$ is the query $(\beta(Q', ?z)$ FILTER $\varphi)$.
(8) If Q is $(Q'$ BIND $(E$ AS $?x))$, then $\beta(Q, ?z)$ is the query $(\beta(Q', ?z)$ BIND $(E$ AS $?x))$.

We omit the rewriting rule for the OPTIONAL operator because this operator can be written in terms of AND, UNION and DIFF, to be precise, P_1 OPTIONAL $P_2 \equiv (P_1$ DIFF $P_2)$ UNION $(P_1$ AND $P_2)$.

Since the DIFF operation requires tracking the provenance of both operands, DIFF translates to an OPTIONAL operation, more precisely, to an OPTIONAL$_{C_v}$ operation, which extends OPTIONAL by renaming variables in the optional pattern with fresh ones. This

[2]The operators concat and aggregate_concat are built-in SPARQL functions.
[3]https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#variableScope

renaming discards undesired bindings produced in the subtrahend while tracking the provenance. For example, consider the patterns $P_1(?x, ?y)$ and $P_2(?x, ?y, ?u)$, whose in-scope variables are indicated in the parenthesis, and let $\mu_1 = \{?x \mapsto a\}$ and $\mu_2 = \{?x \mapsto a, ?y \mapsto b, ?u \mapsto c\}$ be two solutions for them. If instead of OPTIONAL_{C_v}, the rule for DIFF (rule 5) used OPTIONAL, the query would return the provenance for the mapping $\{?x \mapsto a, ?y \mapsto b\}$ instead of the mapping μ_1. If $?x$ is strongly bound for P_1 (i.e., always bound in its answers) and $?y$ is not, then the operation $P_1 \text{ OPTIONAL}_{C_v} P_2$ is:

$$((P(?x, ?y) \text{ OPTIONAL } P(?x, v(?y), ?u)) \\ \text{FILTER } (\neg \text{bound}(?y) \vee \neg \text{bound}(v(?y)) \vee ?y = v(?y))).$$

To define correctness of the query rewriting described in Definition 4.2 we need the notions of *soundness* and *completeness*. A query rewriting is sound when the rewritten query returns right polynomial expressions, and complete if it returns polynomial expressions for all mappings with non-zero polynomials.

Definition 4.3 (Soundness and Completeness). Let Reify be a function that implements a reification scheme, and γ be a function that receives a SPARQL query Q and a variable $?z \notin \text{inScope}(Q)$, and returns a SPARQL query $\gamma(Q, z)$ with $\text{inScope}(\gamma(Q, ?z)) = \text{inScope}(Q) \cup \{?z\}$. Let Γ be a $\mathcal{K}$-graph, and $G = \text{Reify}(\Gamma)$ the RDF-star graph resulting from applying the Reify function to each triple in Γ in order to encode Γ's triple annotations.

(1) γ is called *sound* for Reify if, for every answer of the rewritten query $\mu \cup \{?z \mapsto e\} \in [\![\gamma(Q, ?z)]\!]_G$, e is an expression for the polynomial $(\!|Q|\!)_\Gamma(\mu)$.
(2) γ is called *complete* for Reify if, for every mapping μ such that $(\!|Q|\!)_\Gamma(\mu)$ is a non-zero polynomial, there exists an expression e such that $\mu \cup \{?z \mapsto e\} \in [\![\gamma(Q, ?z)]\!]_G$.

THEOREM 4.4. *Let* Reify *be a reification scheme, and β be the function described in Definition 4.2. Then, function β is sound and complete for the reification scheme* Reify.

PROOF. It can be shown by induction on the query structure. □

We highlight that for queries of the form Q_1 DIFF Q_2, NPCS also returns why-not provenance explanations of the form $k_1 \ominus k_2$ (k_1 and k_2 are polynomials) for the bindings that match both Q_1 and Q_2. The polynomial k_2 tells us which sources must be removed from the graph so that the corresponding binding becomes a query solution.

4.3 Query Rewriting Optimization

If we look at our example rewritten query Q' described in Section 4.1, we can notice that this query includes a GROUP BY clause, and two subqueries, namely P'_1 and P'_2, each of which also includes a GROUP BY clause. In Definition 4.6 we describe an alternative query rewriting that produces equivalent polynomial expressions, but reduces the number of aggregate operations.

Definition 4.5. A *sum-query* Q is a query such that the query rewriting β described in Definition 4.2 returns a query of the form:

$$\begin{array}{lll} \beta(Q, ?z) = (\ \text{SELECT} & \text{inScope}(Q)\ (\text{ProvAggSum}(?z{\oplus}) \text{ AS } ?z) \\ \text{WHERE} & T \\ \text{GROUP BY} & \text{inScope}(Q)\ ). \end{array}$$

We call query T the *pattern* of $\beta(Q, ?z)$.

Note that, according to Definition 4.2, sum-queries are all queries that match the rules 2, 3, 4, and 6.

Definition 4.6. Let Q be a SPARQL query, $?z$ be a variable, and Reify a reification scheme. Then, the rewritten query for Q and variable $?z$ over scheme Reify, denoted $\beta(Q, ?z)$, is defined recursively as is specified in Definition 4.2, but the following rules are applied when possible:

(1) If Q is $(Q_1 \text{ AND} \cdots \text{AND } Q_n)$, and $Q_1, \ldots, Q_n$ are sum-queries, such that for $1 \leq i \leq n$, the pattern of $\beta(Q_i, ?z{\oplus}{\otimes}i)$ is T_i, then $\beta(Q, ?z)$ is the query
$$\begin{array}{lll} (\ \text{SELECT} & \text{inScope}(Q)\ (\text{ProvAggSum}(?z{\oplus}{\otimes}) \text{ AS } ?z) \\ \text{WHERE} & ((\ T_1 \text{ AND} \cdots \text{AND } T_n) \\ & \quad \text{BIND } (\text{ProvProd}(?z{\oplus}{\otimes}1, \ldots, ?z{\oplus}{\otimes}n) \\ & \qquad \text{AS } z{\oplus}{\otimes}) \\ \text{GROUP BY} & \text{inScope}(Q)\ ). \end{array}$$
(2) If Q is $(Q_1 \text{ UNION} \cdots \text{UNION } Q_n)$, where $Q_1, \ldots, Q_n$ are sum-queries, and for $1 \leq i \leq n$, the pattern of $\beta(Q_i, ?z{\oplus})$ is T_i, then $\beta(Q, ?z)$ is the query
$$\begin{array}{lll} (\ \text{SELECT} & \text{inScope}(Q)\ (\text{ProvAggSum}(?z{\oplus}) \text{ AS } ?z) \\ \text{WHERE} & (T_i \text{ UNION} \cdots \text{UNION } T_n) \\ \text{GROUP BY} & \text{inScope}(Q)\ ). \end{array}$$
(3) If Q is $(Q_1 \text{ DIFF } Q_n)$, and ν is a variable substitution that substitutes with fresh variables the variables in $\text{dom}(Q_1) \cap \text{dom}(Q_2)$ that are not strongly bound in Q_1, and Q_2 is a sum-query where the pattern of $\beta(\nu(Q_2), ?z{\ominus}2)$ is T_2, then $\beta(Q, ?z)$ is the query
$$\begin{array}{lll} (\ \text{SELECT} & \text{inScope}(Q) \\ & (\text{ProvDiff}(z{\ominus}1, \text{ProvAggSum}(z{\ominus}2{\oplus})) \text{ AS } ?z) \\ \text{WHERE} & (\beta(Q_1, ?z{\ominus}1) \text{ OPTIONAL}_{C_v} T_2) \\ \text{GROUP BY} & \text{inScope}(Q) \cup \{z{\ominus}1\}\ ). \end{array}$$
(4) If Q is $(\text{SELECT } W \text{ WHERE } Q')$, where Q' is a sum-query such that the pattern of $\beta(Q', ?z{\oplus})$ is T, then $\beta(Q, ?z)$ is
$$\begin{array}{lll} (\ \text{SELECT} & W\ (\text{ProvAggSum}(?z{\oplus}) \text{ AS } ?z) \\ \text{WHERE} & T \\ \text{GROUP BY} & W\ ). \end{array}$$

Note: In rules 1 and 2 of Definition 4.6, we omitted the parenthesis for sequences of operations AND and UNION, because these operators are associative. Intuitively, the associativity of these operators allows considering the binary operation as a single variadic operation with a single GROUP BY clause.

Example 4.7. Consider the query Q from Example 3.2. Then, according to the query rewriting described in Definition 4.6 the rewritten query of Q is:

$$\begin{array}{lll} \beta(Q, ?z) = (\ \text{SELECT} & ?x\ (\text{ProvAggSum}(?z{\oplus}{\otimes}) \text{ AS } ?z) \\ \text{WHERE} & ((\ \text{Reify}(?x, \textit{likes}, \textit{pasta}, ?z{\oplus}{\otimes}1{\oplus}) \text{ AND} \\ & \quad \text{Reify}(?x, \textit{likes}, \textit{pasta}, ?z{\oplus}{\otimes}2{\oplus})) \\ & \quad \text{BIND } (\text{ProvProd}(?z{\oplus}{\otimes}1{\oplus}, ?z{\oplus}{\otimes}2{\oplus}) \\ & \qquad \text{AS } ?z{\oplus}{\otimes})) \\ \text{GROUP BY} & ?x\ ). \end{array}$$

This query has only one GROUP BY clause whereas the query Q' at the end of Section 4.1 (generated using the rewriting of Definition 4.2) has three.

THEOREM 4.8. *Let* Reify *be a reification scheme, and β be the function described in Definition 4.6. Then, function β is sound and complete for the reification scheme* Reify.

Proof. It can be shown by induction on the query structure, that the expressions resulting from the rules in Definition 4.6 produce the same results that the rewriting in Definition 4.2. □

5 EVALUATION

We conducted an extensive evaluation of NPCS's viability for computing how-provenance by assessing the runtime overhead incurred by the rewritten queries. This is measured by comparing the runtime between the original queries without provenance annotations and the queries obtained with our approach.

5.1 Experimental Setup

5.1.1 Environment. NPCS was implemented in Java, using the Java Development Kit (JDK) version 11. All the experiments were conducted on a computer with an AMD EPYC 7281 16-core processor, 256GB of RAM, and an 8 TB HDD disk running Ubuntu 18.04.6 LTS. We evaluated NPCS on two widely used RDF/SPARQL engines with support for RDF-star, namely GraphDB[4] (version 10.2.0) and Stardog[5] (version 9.1.0). Throughout all experiments, we set a timeout of 350 sec. for individual query executions to ensure consistent results and reported the average response time of the queries over five executions in a cold setting, i.e., after clearing the disk cache.

5.1.2 Competitor. We compare NPCS with SPARQLprov [18], a state-of-the-art solution for how-provenance in SPARQL, which is also based on query rewriting. We used the implementation provided by the paper [18], and extended it to support the RDF-star reification scheme. SPARQLprov and NPCS compute the same provenance polynomials since they both rely on spm-semirings.

5.1.3 Synthetic Workload. We employed the Watdiv performance benchmark specifically designed for RDF/SPARQL engines. Watdiv provides a data generator that can produce synthetic datasets of varying sizes. Additionally, WatDiv includes 20 SELECT query templates, each comprising 10 instantiated queries. The query templates are categorized into four types: linear queries (L), star queries (S), snowflake-shaped queries (F), and complex queries (C). They are all monotonic queries. We therefore introduced five additional non-monotonic query templates (O) as proposed by [18]. These non-monotonic queries were created by enclosing one of the triple patterns in the linear queries with an OPTIONAL clause.

We evaluated NPCS on the 10M-triple and 100M-triple Watdiv datasets, that we reified using the RDF-star and named graphs reification schemes. We excluded the standard reification from the evaluation as it exhibits the worst performance according to [18]. Moreover, we created a 200M-triple dataset by duplicating every triple of the 100M-triple dataset and assigning a second provenance identifier to the duplicates. This dataset simulates a challenging case where triples have been extracted from more than one source.

5.1.4 Real Workload. We tested NPCS and SPARQLprov on the WDBench benchmark [1], which provides real-world data. The benchmark uses data from 15.2 billion triples encoded using the Wikidata reification scheme in a 2023 Wikidata dump. The benchmark provides more than 800 queries consisting of simple BGPs, some of them with OPTIONAL clauses. We took a sample of 150 queries consisting of 50 single-triple-pattern queries, 50 non-monotonic queries (with OPTIONAL), and 50 monotonic queries with more than on triple pattern. The queries were randomly chosen.

[4]https://graphdb.ontotext.com/
[5]https://www.stardog.com/

5.2 Results

5.2.1 Synthetic Workload. Figure 2 compares the execution times of the original query with those of the rewritten queries produced by NPCS and our competitor SPARQLprov on RDF-star data when using GraphDB and Stardog. We mesure the runtimes on the 10M and 100M Watdiv datasets. We first notice that in all cases, rewriting the query to compute how-provenance incurs a performance overhead. Not surprisingly, the overhead increases with data size, but its behaviour also depends on the query engine. For instance, NPCS's overhead ranges from 20% to 30% in GraphDB, and from 25% to 50% in Stardog. Figure 2 also reveals that, on average, GraphDB is more sensitive to data scaling than Stardog, even though runtime across query templates exhibits higher variability in Stardog. Regardless of the data size and the query engine, templates C3 and F5 are by far the most challenging, which makes our competitor SPARQLprov timeout on both GraphDB and Stardog. The complexity of C3 is explained by its large number of intermediate results, whereas for F5 it is caused by the large number of solutions.

When we compare the query rewriting strategies, we notice the NPCS consistently outperforms SPARQLprov in 98 out of our 100 studied cases. Also, NPCS is on average 25 times faster than SPARQLprov. One can explain this performance difference by the fact that NPCS is a fully native SPARQL solution, whereas SPARQLprov relies on a post-hoc decoding phase to compute the provenance polynomials. Like NPCS, SPARQLprov rewrites the query to extract provenance information. Unlike our approach, SPARQLprov encodes the structure of the how-provenance annotations in additional columns in the result set. Those additional columns can be numerous and encode the structure of the provenance polynomials. Decoding that information requires running additional group and aggregation operations. Hence, the runtime of this decoding phase is proportional to the number of query solutions times the maximal depth of the operator trees of the provenance annotations. That explains why SPARQLprov times out for query template F5, which is by far the template with the highest number of query solutions (173.6K solutions on average). NPCS, in contrast, carries out the grouping operations during query evaluation, which not only leverages the engine optimizations for grouping, but also makes it easier to deploy in real-world settings.

Despite NPCS's clear runtime advantage, SPARQLprov can exhibit comparable or better performance on very selective queries. This is demonstrated by the runtimes for queries O1, O2, and O5. In cases such as query templates O2 and O5 on GraphDB, NPCS's strategy of evaluating grouping operations in the source engine does not pay off. This is so because the queries and their constituent triple patterns are very selective.

Figure 3 shows the results for the rewritten queries on the 200M dataset on GraphDB. We observe similar trends as in the 100M scenario, except that SPARQLprov also times out on query templates C2 and F4. We omit the results for Stardog as they exhibit similar behavior as in the 100M dataset.

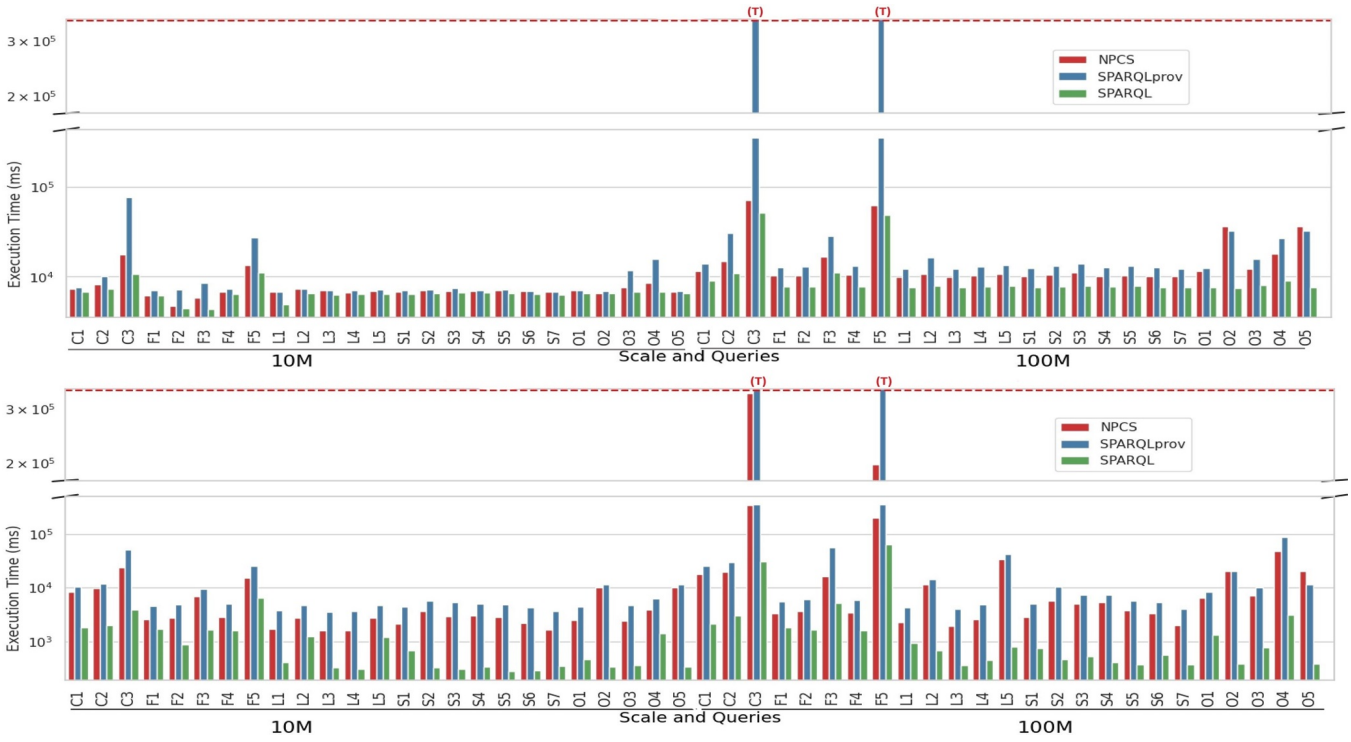

Figure 2: Query execution times on Watdiv 10M and 100M reified with RDF-star on GraphDB (top) and Stardog (bottom)

Finally, we evaluate NPCS on a different reification scheme, namely the popular named graphs strategy. The results are depicted in Figure 4 for the 10M and 100M datasets on GraphDB. We observe the same trends as for the RDF-star reification, that is, NPCS outperforms SPARQLprov consistently in 48 out of 50 studied cases. This shows that our approach is insensitive to the data reification scheme, which makes it applicable to any standard RDF/SPARQL engine. Similar results are observed for Stardog.

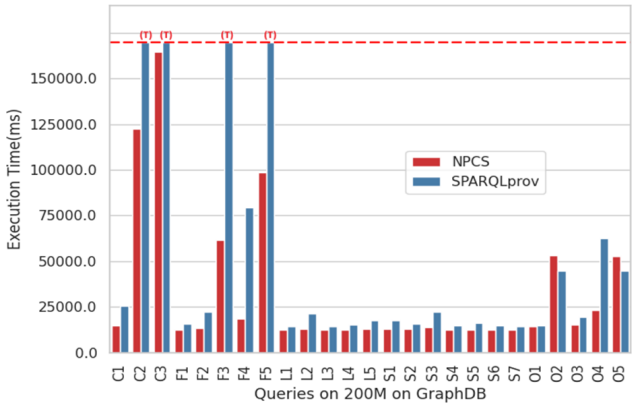

Figure 3: Query execution times on Watdiv 200M reified with RDF-star on GraphDB

5.2.2 Real Workload. We evaluate NPCS on the WDBench. Figure 5 shows the results for GraphDB and Stardog. Each dot in the plot represents the execution of a query, either the original query or a rewritten query by NPCS or by SPARQLprov. Queries are plotted on the x-axis by increasing number of solutions, and the y-axis represents the execution time. We verify the same trend for both engines, namely that SPARQLprov's query rewriting induces a much larger overhead than NPCS's. While the overhead increases with the number of query results for both methods, it is more pronounced for SPARQLprov. This makes SPARQLprov time out when the number of results is above 700K. We also observe that for queries with a few thousand results executed on Stardog, NPCS's overhead can be minimal.

6 CONCLUSIONS

We have proposed NPCS, a novel query rewriting method to compute how-provenance annotations for SPARQL query results. To the best of our knowledge, NPCS is the first 100% SPARQL-based solution for how-provenance. Our approach can be easily applied to standard and already deployed RDF/SPARQL engines, without the need for customized extensions or post-processing steps. Our experimental evaluation on synthetic and real data shows that NPCS's native SPARQL rewriting outperforms the state of the art in how-provenance for SPARQL queries. The performance gains provided by our method allow us to compute provenance annotations for millions of query results on knowledge graphs with billions of triples. This makes NPCS attractive for ETL processes on large volumes of data—a common scenario for multi-source KG construction and OLAP for KGs [10, 19, 20].

As future work we intend to work on lazy approaches for how-provenance computation, that is, approaches where provenance is computed for a user-specified set of solutions. This avoids the

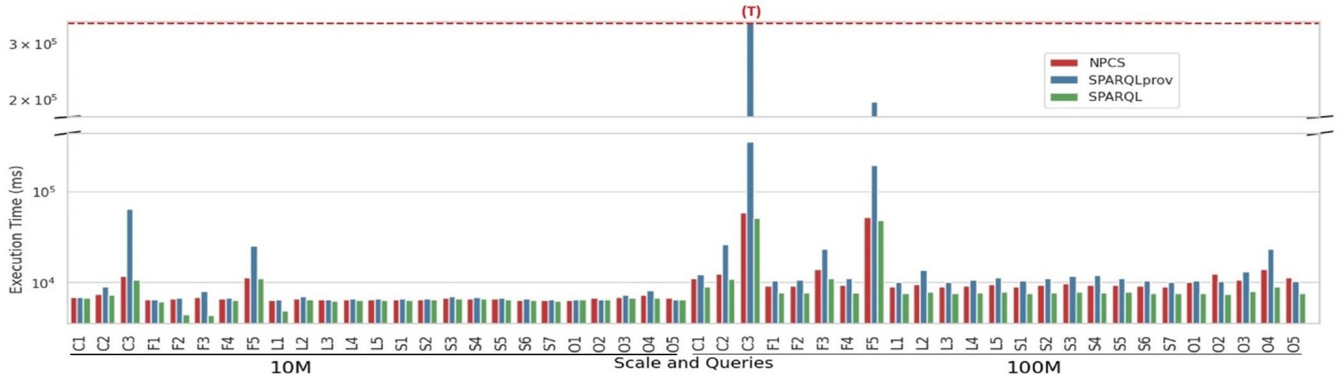

Figure 4: Query execution times on Watdiv 10M and 100M reified as named graphs on GraphDB

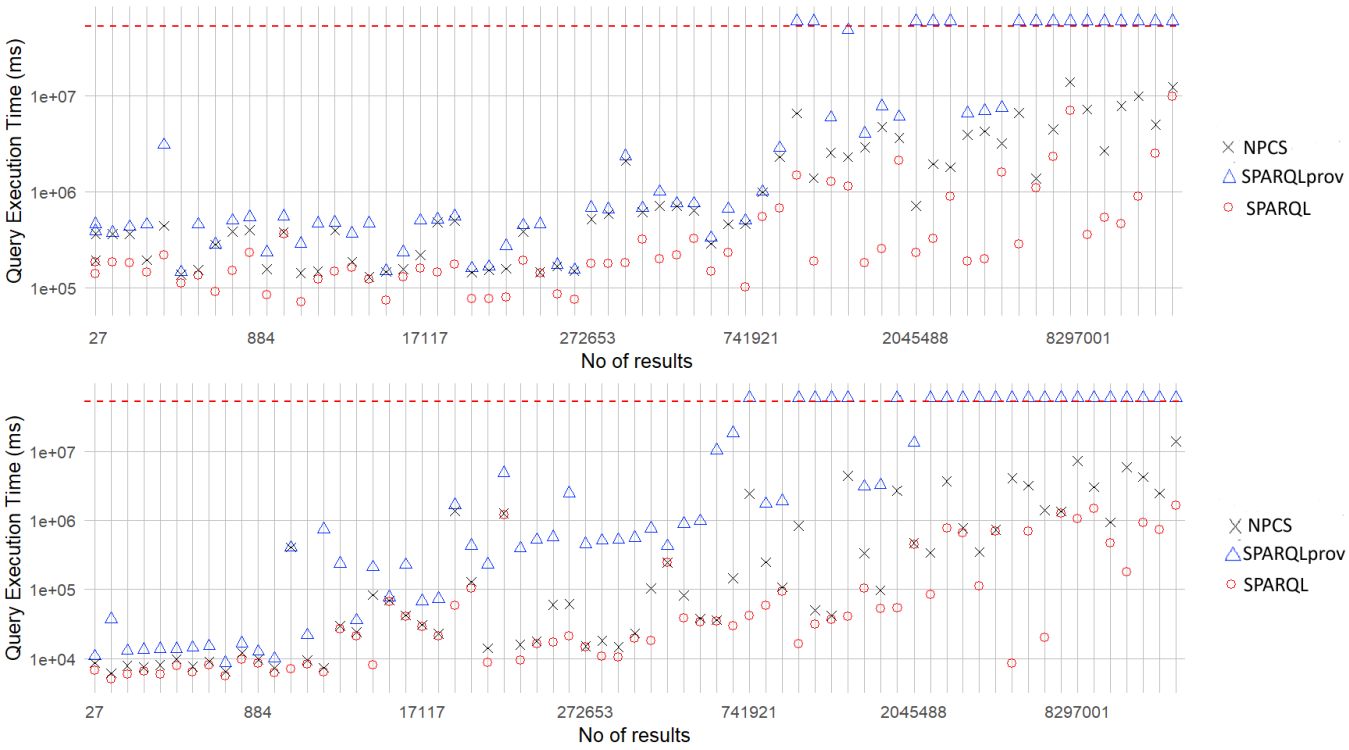

Figure 5: Number of results vs. query execution time for the WDBench queries run on Wikidata, stored in GraphDB (top) and Stardog (bottom), using the Wikidata reification scheme

execution of expensive queries for results that are not of interest of the user. We have also envisioned to tackle the problem of computing how-provenance annotations for non-reified data, and add the support for CONSTRUCT queries.

SUPPLEMENTARY MATERIAL STATEMENT

NPCS's source code[6], the scripts to recreate the experimental setup, all required libraries, queries, and results can be found in [5].

ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 R&I Marie Skłodowska-Curie program, project KnowGraphs – 860801; the TAILOR Network (EU Horizon 2020 R&I program under GA 952215); the COST Action CA19134; the Deutsche Forschungsgemeinschaft (DFG): Germany's Excellence Strategy – EXC 2120/1 – 390831618; and the DFG: SPP 1921 – 318363223 (COFFEE STA 572_15-2).

[6]URL: https://github.com/ZubariaForthAcc/NPCS

REFERENCES

[1] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoc. 2022. WDBench: A Wikidata Graph Query Benchmark. In *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13489)*. Springer International Publishing, Cham, 714–731. https://doi.org/10.1007/978-3-031-19433-7_41

[2] Renzo Angles and Claudio Gutierrez. 2016. The Multiset Semantics of SPARQL Patterns. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9981)*. Springer International Publishing, Cham, 20–36. https://doi.org/10.1007/978-3-319-46523-4_2

[3] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Engineering Bulletin* 41, 1 (2018), 51–62. http://sites.computer.org/debull/A18mar/p51.pdf

[4] Carlos Buil Aranda, Marcelo Arenas, Óscar Corcho, and Axel Polleres. 2013. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Semant.* 18, 1 (2013), 1–17. https://doi.org/10.1016/j.websem.2012.10.001

[5] Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irini Fundulaki, and Katja Hose. 2024. Code and benchmark for NPCS, a Native Provenance Computation for SPARQL. https://doi.org/10.18419/darus-3973

[6] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory — ICDT 2001*, Jan Van den Bussche and Victor Vianu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 316–330.

[7] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. Database Syst.* 25, 2 (jun 2000), 179–227. https://doi.org/10.1145/357775.357777

[8] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. 2012. Provenance for SPARQL Queries. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 7649)*. Springer International Publishing, Cham, 625–640. https://doi.org/10.1007/978-3-642-35176-1_39

[9] Renata Queiroz Dividino, Sergej Sizov, Steffen Staab, and Bernhard Schueler. 2009. Querying for Provenance, Trust, Uncertainty and other Meta Knowledge in RDF. *Journal of Web Semantics* 7, 3 (2009), 204–219.

[10] Luis Galárraga, Kim Ahlstrøm Jakobsen, Katja Hose, and Torben Bach Pedersen. 2018. Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets. In *ISWC (1) (Lecture Notes in Computer Science, Vol. 11136)*. Springer, 547–565.

[11] Floris Geerts and Antonella Poggi. 2010. On Database Query Languages for K-relations. *Journal of Applied Logic* 8, 2 (2010), 173–185.

[12] Floris Geerts, Thomas Unger, Grigoris Karvounarakis, Irini Fundulaki, and Vassilis Christophides. 2016. Algebraic Structures for Capturing the Provenance of SPARQL Queries. *J. ACM* 63, 1 (2016), 7:1–7:63.

[13] F. Geerts, T. Unger, G. Karvounarakis, I. Fundulaki, and V. Christophides. 2016. Algebraic Structures for Capturing the Provenance of SPARQL Queries. *J. ACM* 63, 1 (2016), 7:1–7:63.

[14] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. IEEE Computer Society, Washington, DC, USA, 174–185. https://doi.org/10.1109/ICDE.2009.15

[15] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

[16] Olaf Hartig. 2009. Querying Trust in RDF Data with tSPARQL. In *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5554)*. Springer International Publishing, Cham, 5–20. https://doi.org/10.1007/978-3-642-02121-3_5

[17] Olaf Hartig, Pierre-Antoine Champin, Gregg Kellogg, and Andy Seaborne. 2021. *RDF-star and SPARQL-star*. Technical Report. W3C Community Group Draft Report. https://w3c.github.io/rdf-star/cg-spec/2021-12-17.html

[18] Daniel Hernández, Luis Galárraga, and Katja Hose. 2021. Computing How-Provenance for SPARQL Queries via Query Rewriting. *Proc. VLDB Endow.* 14, 13 (2021), 3389–3401. https://doi.org/10.14778/3484224.3484235

[19] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2016. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In *ISWC (1) (Lecture Notes in Computer Science, Vol. 9981)*. 341–359.

[20] Matteo Lissandrini, Katja Hose, and Torben Bach Pedersen. 2023. Example-Driven Exploratory Analytics over Knowledge Graphs. In *EDBT*. OpenProceedings.org, 105–117.

[21] J. Pérez, M. Arenas, and C. Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[22] Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. 2014. TripleProv: efficient processing of lineage queries in a native RDF store. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*. ACM, New York, NY, USA, 455–466. https://doi.org/10.1145/2566486.2568014