Native Provenance Computation for Federated and Non-Federated SPARQL Queries

Zubaria Asma
FORTH-ICS, Heraklion, Crete, Greece
University of Crete, Heraklion, Crete, Greece

Daniel Hernández
University of Stuttgart, Stuttgart, Germany

Luis Galárraga
Inria, Rennes, France

Giorgos Flouris
FORTH-ICS, Heraklion, Crete, Greece

Irini Fundulaki
FORTH-ICS, Heraklion, Crete, Greece

Katja Hose
TU Wien, Wien, Austria

Abstract

The popularity of knowledge graphs (KGs) owes credit to their flexible data model, which is suitable for data integration from multiple sources. Several KG-based applications, such as trust assessment, view maintenance, or data valuation on dynamic data, rely on the ability to compute provenance explanations for query results. This need becomes more urgent in federated query processing systems, which allow the online consumption of heterogeneous and decentralized Web data. However, the problem of computing and interacting with provenance has received little attention, especially in the federated setting. On those grounds, this paper introduces the NPCS (Native Provenance Computation for SPARQL) approach, and its federated variant Fed-NPCS, that compute provenance for SPARQL query results. Both approaches build upon spm-provenance semirings to annotate the results of monotonic and non-monotonic SPARQL queries with their provenance. Due to their reliance on query rewriting techniques, the approaches are directly applicable to already deployed SPARQL engines and federations using different reification schemes, including RDF-star. Our experimental evaluation shows that our novel query rewriting approach brings significant run-time improvements w.r.t. the state-of-the-art across both centralized and federated settings. In centralized settings, our tests on two popular SPARQL engines (GraphDB and Stardog) reveal substantial runtime gains over existing query rewriting solutions, enabling scalability to RDF graphs with billions of triples. In federated settings, our experiments on the FedShop benchmark with GraphDB show the viability of Fed-NPCS for federations with up to 200 sources.

2012 ACM Subject Classification Information systems → Data provenance; Information systems → Resource Description Framework (RDF)

Keywords and phrases native provenance computation, federated SPARQL queries, data provenance, NPCS, Fed-NPCS

Digital Object Identifier 10.4230/TGDK.1.1.42

Acknowledgements This work was funded by the EU H2020 R&I Marie Skłodowska-Curie program, project KnowGraphs – 860801; the TAILOR project (EU Horizon 2020 R&I program, GA 952215); the COST Action CA19134; and the German Research Foundation (DFG) research project SFB 1574-471687386.

Received Date of submission **Accepted** Date of acceptance **Published** Date of publishing

Editor TGDK section area editor

1 Introduction

The fast advances in information extraction and knowledge graph (KG) construction have enriched the Web with vast amounts of structured, machine-readable data. These KGs, represented as RDF triples and queried through SPARQL endpoints, form the foundation of numerous intelligent applications such as semantic search, question answering, and digital assistants. By encoding real-world knowledge as triples in the form (s, p, o), knowledge graphs allow machines to "understand" and process complex information, driving many data-driven systems.

KGs usually aggregate data from independent and heterogeneous sources. One common approach for KG construction is to apply information extraction techniques, e.g., on the Web, to filter, cleanse, and centralize knowledge so that it can be queried on a single endpoint. Although this strategy guarantees full control over the data, it requires a lot of effort to keep the data up-to-date. On the other hand, federated query systems [2, 3, 13, 24, 29, 36, 39, 41, 42] allow users to query knowledge in a fully decentralized way, giving the impression of a large centralized KG. Federated approaches provide an abstraction layer that hides the complexity of fully distributed processing including tasks such as source selection and efficient query planning. This capability is essential for various downstream applications such as data integration platforms, search engines, or smart assistants, which require real-time access to fresh knowledge. Federated architectures are also crucial when third parties must hold control of the data for the sake of data privacy [44] or for legal reasons.

Given the heterogeneity of the data sources that contribute to modern KGs, the problem of identifying the *provenance* of query results is central – especially in the federated setting. The provenance of a query result is an expression that encodes the lineage of the 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 [19]. 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 and privacy [37], data valuation, trustworthiness, auditing [40], or data quality.

Among the existing formalisms to model query provenance, *how-provenance* is the most expressive [25]. In this model, the provenance of a query result is an algebraic expression in a *provenance semiring*. Consider, for instance, the following KG,

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

and the SPARQL query

$$\textsc{select}\ ?x\ \textsc{where}\ \{\ \{\ \mathit{UK}, \mathit{capital}, ?x\ \}\ \textsc{union}\ \{\ ?x\ \mathit{in}\ \mathit{UK}\ ;\ \mathit{a}\ \mathit{City}\ \}\ \}.$$

The answer 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 [15, 22, 25], but this paper considers spm-semirings [22] 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.

In the centralized setting, there are essentially two main strategies to compute how-provenance for SPARQL queries. Methods such as TripleProv [45] opt for customized engines that compute provenance alongside 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 [10, 30].

In this approach, 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. This design provides the flexibility to be applied to any SPARQL endpoint on the Web at the price of a runtime overhead. Both of these approaches have been applied for the centralized setting; however, to the best of our knowledge, no system exists that allows the computation of how-provenance for query results in the federated setting.

In this paper, we first describe the *NPCS*[1] approach [10], a method to compute provenance for SPARQL queries via query rewriting. NPCS is the first fully native SPARQL solution for how-provenance computation, as previous approaches [30] required some sort of post-processing. Moreover, NPCS supports different data reification schemes, including RDF-star. NPCS is designed for centralized KGs; in this paper we introduce an extension of the method, called *Fed-NPCS*, which is applicable for queries run against federated systems. This new work includes an extension of the how-provenance formalism for federated settings, as well as a runtime evaluation of Fed-NPCS on FedShop [16], a challenging benchmark for SPARQL federated systems. To the best of our knowledge, this is the first work that addresses the computation of how-provenance explanations for SPARQL queries on federations.

Our experimental evaluation demonstrates that NPCS is consistently faster than SPARQL-prov [30], the state of the art in how-provenance in SPARQL for centralized KGs. Moreover, NPCS and Fed-NPCS provide provenance explanations with reasonable runtime overhead (around 10%) while scaling efficiently to large datasets with billions of triples and federations of up to 200 endpoints.

The rest of this paper is structured as follows. Section 2 provides an overview of related work in the areas of provenance and federated query processing, whereas Section 3 introduces some preliminary concepts that will be useful throughout the paper. Section 4 presents our query rewriting method and Section 5 describes its extension to federated SPARQL query processing systems. In Section 6, we report on our experimental evaluation. Finally, Section 7 concludes with a discussion of the implications of our work and an outline of directions for future research.

2 Related work

We survey the literature on provenance for query results along two axes: provenance models (Section 2.1) and provenance support for RDF/SPARQL engines (Section 2.2). For a survey on federated SPARQL systems and related benchmarks we refer the reader to [16, 26]. We highlight that, so far, no other work has addressed the problem of computing how-provenance for SPARQL federated queries.

2.1 Semirings and Provenance Models

Semirings were first used to model query provenance in the groundbreaking work by Green et al. [25]. 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 [25], hence the algebraic structures were expanded to include a monus operator[2] that accounts for the relational difference [21]. Commutative semirings and their extensions model provenance as polynomial expressions. These expressions, called *how-provenance*,

[1] Native Provenance Computation for SPARQL

[2] Do not confuse monus with minus (the SPARQL operator). A monus operator is an operator on certain commutative monoids that are not groups (see [21]).

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 [14] or why-provenance [12].

Damasio et al. [15] showed that by rewriting SPARQL queries into relational algebra, we can provide provenance annotations for SPARQL queries using *m-semirings*. However, Geerts et al. [22] showed 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 [17, 27] via the notion of commutation with homomorphisms.

2.2 Provenance-supported SPARQL engines

Wylot et al. [45] introduced TripleProv, a system to compute provenance annotations in the commutative semiring framework for queries with basic graph patterns, union, and the OPTIONAL operator. Due to 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 [23] and GProM [8] are two examples. Such approaches are, however, 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. [22] 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 Hernández et al. [30]. 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, establishing query rewriting as a practical approach for computing SPARQL how-provenance. 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

The following presentation follows the W3C Community Group Draft [28]. We assume the existence of three (pairwise disjoint) countably infinite sets: the set of IRIs $\mathbf{I}$, the set of blank nodes $\mathbf{B}$, and the set of literals $\mathbf{L}$. An RDF triple $t = (s, p, o) \in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ 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 1** (RDF-star). *An RDF-star triple is a 3-tuple defined recursively as follows:*

- *Every RDF triple t is an RDF-star triple; and*

- *Given RDF-star triples t and t', and RDF terms $s \in (\mathbf{I} \cup \mathbf{B})$, $p \in \mathbf{I}$, and $o \in (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$, then the triples (t, p, o), (s, p, t), and (t, p, t') are also RDF-star triples.*

We write $\mathbf{T}$ to denote the set of all RDF-star triples. An RDF-star graph G is a finite set of RDF-star triples (i.e., $G \subseteq \mathbf{T}$). We write $\mathbf{G}$ to denote the set of all RDF-star graphs.

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.

▶ **Definition 2** (Federated Dataset). *Given a finite set $Y \subseteq \mathbf{I}$, called the set of* graph names*, a* federated dataset *over Y is a function $D : Y \to \mathbf{G}$.*

Intuitively, a federated dataset D associates IRIs with RDF-star graphs.

3.2 SPARQL-star

RDF-star graphs can be queried using the SPARQL-star language. The base of SPARQL-star queries is a countably infinite set $\mathbf{V}$ of variables—prefixed by the character '?' and disjoint with $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. A triple pattern is defined recursively as a follows:

- A triple $(S, P, O) \in (\mathbf{V} \cup \mathbf{I} \cup \mathbf{B}) \times (\mathbf{V} \cup \mathbf{I}) \times (\mathbf{V} \cup \mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ is a triple pattern.
- Given two triple patterns T_1 and T_2, the triples (T_1, P, O), (S, P, T_2), and (T_1, P, T_2) are SPARQL-star triple patterns.

Triple patterns in SPARQL-star queries are combined with operators (e.g., AND, UNION, SELECT) to define SPARQL-star queries.

▶ **Definition 3** (SPARQL-star Syntax). *The syntax of the* SPARQL-star algebra*, whose expressions are called queries, is defined recursively as follows. For each query we also define whether it is* local*,* remote*, or* fully remote*.*

- *An empty basic graph pattern $\{\}$ is a local query.*
- *A triple pattern T is a local query.*
- *Given an IRI $y \in Y$, and a local query Q', the expression $Q =$ (SERVICE y Q') is a fully remote query.*
- *Given two queries Q_1 and Q_2, the expressions (Q_1 AND Q_2), (Q_1 UNION Q_2), (Q_1 DIFF Q_2), and (Q_1 OPTIONAL Q_2) are queries. If Q_1 and Q_2 are local, then these four queries are said to be local. If Q_1 and Q_2 are fully remote, then these four queries are said to be fully remote. Otherwise, if these queries combine a local and a fully remote query, they are said to be remote.*
- *A selection formula is consists of a combination atoms of the form $A = B$ and* bound(`?x`)*, where $A, B \in \mathbf{I} \cup \mathbf{L}$ and `?x` $\in \mathbf{V}$, with the logical connectives $\wedge$, $\vee$, and $\neg$. Given a query Q' and a selection formula φ, the expression $Q = (Q'$ FILTER $\varphi)$ is a query, which is local if Q' is local, fully remote if Q' is fully remote, and remote if Q' is remote.*
- *Given a query Q' and a finite set of variables W, the expression $Q =$ (SELECT W WHERE Q') is a query, which is local if Q' is local, fully remote if Q' is fully remote, and remote if Q' is remote.*

▶ Note 4. Notice that this definition does not allow a SERVICE clause to include another SERVICE clause. We follow this design because according to the SPARQL 1.1. specification, a service clause can contain a single endpoint.

▶ **Example 5.** Let T_1 and T_2 be two triple patterns, and consider the following three queries:

- $Q_1 = (T_1 \text{ AND } T_2)$
- $Q_2 = (T_1 \text{ AND } (\text{SERVICE } y_2\ T_2))$
- $Q_3 = ((\text{SERVICE } y_1\ T_1) \text{ AND } (\text{SERVICE } y_2\ T_2))$

Query Q_1 is local; query Q_2 is remote but not fully-remote because the triple patterns T_1 does not occur in a remote subquery; and query Q_3 is fully-remote because both triple patterns occur in remote subqueries.

▶ Note 6. The syntax described in Definition 3 follows the syntax introduced by Perez et al. [38], and extended in Geerts et al. [22] with the operator SELECT, and by Buil-Aranda [9] with the operator SERVICE. Unlike them [22, 38], we call the difference operator DIFF because it differs from the standard MINUS operator [6, 34]. The operator DIFF is part of the standard SPARQL 1.1 algebra, and the operator MINUS is part of the standard SPARQL 1.1 query language syntax. Also, the operator MINUS is expressible with the other operators we include in Definition 3 [34]. An additional difference compared to the above works is that we syntactically classify queries in *local*, *remote* and *fully-remote* depending on where the triple patterns are evaluated. In this paper we build upon the how-provenance theoretical framework by Geerts et al. [22] who used the operator DIFF in their annotated SPARQL algebra (but called it MINUS).

▶ Note 7. The syntax of the SERVICE operator that we presented in Definition 3 does not allow variables as service names. We introduce this simplification because this feature is not needed to express execution plans for queries over federations.

A (solution) *mapping* is a partial function $\mu : \mathbf{V} \to (\mathbf{T} \cup \mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ where the domain of μ, denoted by $\text{dom}(\mu)$, is a finite set of variables. We write $\mu_\emptyset$ to denote the solution mapping with an empty domain, called the *empty solution mapping*. Two mappings μ_1 and μ_2 are said to be compatible, denoted $\mu_1 \sim \mu_2$, if $\mu_1(\texttt{?x}) = \mu_2(\texttt{?x})$ for each variable $\texttt{?x} \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. Given a mapping μ, and a set of variables $W \subseteq \text{dom}(\mu)$, we write $\mu|_W$ to denote the mapping such that $\text{dom}(\mu|_W) = W$ and $\mu|_W \sim \mu$. Given a triple pattern T, and a mapping μ whose domain $\text{dom}(\mu)$ consists of all variables occurring in T, we write $\mu(T)$ to denote the RDF-star triple t resulting from replacing every variable ?x in T with $\mu(\texttt{?x})$. In short, the evaluation of a SPARQL query Q on an RDF dataset D and an RDF-star graph G, is defined as a function $[\![Q]\!]_{D,G}$ that returns a set of mappings. The details of SPARQL evaluation semantics are detailed in [7, 38]. We next present a definition of the semantics of SPARQL-star.

▶ **Definition 8** (SPARQL-star Semantics). *Given a SPARQL query Q, a federated dataset D, and an RDF-star graph G, we write $[\![Q]\!]_{D,G}$ to denote the set of mappings obtained from evaluating Q in D and G, which is defined recursively as follows:*

- *If $\{\}$ is an empty basic graph pattern, then $[\![\{\}]\!]_{D,G}$ is the set with a single mapping $\mu_\emptyset$ such that* $\text{dom}(\mu) = \emptyset$.
- *If T is a triple pattern, then $[\![T]\!]_{D,G}$ is the set of mappings μ such that* $\text{dom}(\mu)$ *is the set of variables occurring in T and $\mu(T) \in G$.*
- $[\![(\text{SERVICE } y\ Q)]\!]_{D,G} = [\![Q]\!]_{D,D(y)}$.
- *$[\![Q_1 \text{ AND } Q_2]\!]_{D,G}$ is the set of all mappings μ such that $\mu_1 \in [\![Q_1]\!]_{D,G}$ and $\mu_2 \in [\![Q_2]\!]_{D,G}$, $\mu_1 \sim \mu_2$, and $\mu = \mu_1 \cup \mu_2$.*
- *$[\![Q_1 \text{ UNION } Q_2]\!]_{D,G}$ is the set of all mappings μ such that $\mu \in [\![Q_1]\!]_{D,G}$ or $\mu \in [\![Q_2]\!]_{D,G}$.*
- *$[\![Q_1 \text{ DIFF } Q_2]\!]_{D,G}$ is the set of all mappings μ_1 such that $\mu_1 \in [\![Q_1]\!]_{D,G}$ and every mapping $\mu_2 \in [\![Q_2]\!]_{D,G}$ is incompatible with μ_1.*
- *$[\![Q_1 \text{ OPTIONAL } Q_2]\!]_{D,G}$ is the set of all mappings μ such that $\mu \in [\![Q_1 \text{ AND } Q_2]\!]_{D,G}$ or $\mu \in [\![Q_1 \text{ DIFF } Q_2]\!]_{D,G}$.*

- $[\![Q \text{ FILTER } \varphi]\!]_{D,G}$ *is the set of all mappings* μ *such that* $\mu \in [\![Q]\!]_{D,G}$ *and* φ *is a true formula according to mapping* μ*, denoted* $\mu \models \varphi$.
  We write $\mu \models \varphi$ *if the value of the formula* φ *according to* μ*, denoted* $\varphi[\mu]$ *is 1. The value* $\varphi[\mu]$ *is one the values in* $\{0, \frac{1}{2}, 1\}$ *defined recursively as follows:*
  - *Let* φ *be the a formula of the form* $A = B$*. If there is a variable in* φ *that is not in* $\text{dom}(\mu)$*, then* $\varphi[\mu] = \frac{1}{2}$*. Otherwise, the formula* $\mu(\varphi)$ *resulting from replacing the variables* `?x` *in* φ *with* $\mu(\texttt{?x})$ *can have the forms* $a = a$ *or* $a = b$*, where* a *and* b *are two different values in the set* $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ *(e.g., if* φ *is the formula* $\texttt{?x} = a$ *and* $\mu(\texttt{?x}) = a$ *then the* $\mu(\varphi)$ *is* $a = a$*). If* $\mu(\varphi)$ *is* $a = a$ *then* $\varphi[\mu] = 1$*. Otherwise, if* $\mu(\varphi)$ *is* $a = b$ *then* $\varphi[\mu] = 0$.
  - *If* φ *has the form* $\text{bound}(\texttt{?x})$*, then* $\varphi[\mu] = 1$ *if* $\texttt{?x} \in \text{dom}(\mu)$*. Otherwise,* $\varphi[\mu] = 0$.
  - $(\neg\varphi)[\mu] = 1 - \varphi[\mu]$, $(\varphi \wedge \psi)[\mu] = \min(\varphi[\mu], \psi[\mu])$, *and* $(\varphi \vee \psi)[\mu] = \max(\varphi[\mu], \psi[\mu])$.
- $[\![\text{SELECT } W \text{ WHERE } Q]\!]_{D,G}$ *is the set of all mappings* μ *such that there exists a mapping* $\mu' \in [\![Q]\!]_{D,G}$ *with* $\mu = \mu'|_W$.

▶ Note 9. It is not difficult to see that if a query Q is fully-remote, then the set $[\![Q]\!]_{D,G}$ does not depend on the graph G. Similarly, if query Q is local, then the set $[\![Q]\!]_{D,G}$ does not depend on the federated dataset D. Following this observation, we will abbreviate the aforementioned set as $[\![Q]\!]_D$ if Q is fully-remote, and as $[\![Q]\!]_G$ if Q is local.

Not all variables occurring in a query necessarily appear on the solutions. The problem of determining which variables can appear and which variables cannot appear is in general undecidable [9], but an approximation can be defined syntactically. A variable is said *in-scope* when it can appear in the solutions and *strongly bound* when it necessarily appears in all the solutions.

▶ **Definition 10** (SPARQL-star in-scope and strongly bound variables). *Given a* SPARQL-star *query* Q*, the sets of variables that are* in-scope *and* strongly bound*, denoted respectively* $\text{inScope}(Q)$ *and* $\text{stronglyBound}(Q)$*, are recursively defined as follows.*

- $\text{inScope}(\{\}) = \emptyset$ *and* $\text{stronglyBound}(\{\}) = \emptyset$.
- *Given a triple pattern* T*. The sets* $\text{inScope}(T)$ *and* $\text{stronglyBound}(T)$ *are equal to the set of variables occurring in* T.
- *Given the queries* Q_1 *and* Q_2*, the following identities define the in-scope variables in compound queries:*
  - $\text{inScope}(\text{SERVICE } y\ Q_1) = \text{inScope}(Q_1)$,
  - $\text{inScope}(Q_1 \text{ AND } Q_2) = \text{inScope}(Q_1) \cup \text{inScope}(Q_2)$,
  - $\text{inScope}(Q_1 \text{ UNION } Q_2) = \text{inScope}(Q_1) \cup \text{inScope}(Q_2)$,
  - $\text{inScope}(Q_1 \text{ OPTIONAL } Q_2) = \text{inScope}(Q_1) \cup \text{inScope}(Q_2)$,
  - $\text{inScope}(Q_1 \text{ DIFF } Q_2) = \text{inScope}(Q_1)$,
  - $\text{inScope}(Q_1 \text{ FILTER } \varphi) = \text{inScope}(Q_1)$,
  - $\text{inScope}(\text{SELECT } W \text{ WHERE } Q_1) = W \cap \text{inScope}(Q_1)$.
- *Given the queries* Q_1 *and* Q_2*, the following identities define the strongly bound variables in compound queries:*
  - $\text{stronglyBound}(\text{SERVICE } y\ Q_1) = \text{stronglyBound}(Q_1)$
  - $\text{stronglyBound}(Q_1 \text{ AND } Q_2) = \text{stronglyBound}(Q_1) \cap \text{stronglyBound}(Q_2)$,
  - $\text{stronglyBound}(Q_1 \text{ UNION } Q_2) = \text{stronglyBound}(Q_1) \cap \text{stronglyBound}(Q_2)$,
  - $\text{stronglyBound}(Q_1 \text{ OPTIONAL } Q_2) = \text{stronglyBound}(Q_1) \cap \text{stronglyBound}(Q_2)$,
  - $\text{stronglyBound}(Q_1 \text{ DIFF } Q_2) = \text{stronglyBound}(Q_1)$,
  - $\text{stronglyBound}(Q_1 \text{ FILTER } \varphi) = \text{stronglyBound}(Q_1)$,
  - ~~$\text{stronglyBound}(\text{SELECT } W \text{ WHERE } Q_1) = W \cap \text{stronglyBound}(Q')$.~~

- $\text{stronglyBound}(\text{SELECT}\, W\ \text{WHERE}\ Q_1) = W \cap \text{stronglyBound}(Q_1)$.

▶ Note 11. One may think that ?x ∈ inScope(Q) implies that there exists one graph where there is a mapping $\mu \in [\![Q]\!]_G$ such that ?x ∈ dom(μ). However, this is not true. Indeed, the query

$$Q = (\{\}\ \text{OPTIONAL}\ ((a, p, \texttt{?x})\ \text{DIFF}\ (a, p, \texttt{?x})))$$

includes the variable ?x in inScope(Q) and for every graph G, each $\mu \in [\![Q]\!]_G$ holds that ?x $\notin$ dom(μ). One may think something similar regarding the variables in stronglyBound(Q). That is, thinking that every variable ?x that occurs in the domain of all solution mappings of a query Q in every graph G must be a strongly bound. This happens to variable ?x in the query

$$Q' = ((a, p, \texttt{?x})\ \text{UNION}\ (\{\}\ \text{FILTER}\ \neg(a = a))).$$

However, variable ?x $\notin$ stronglyBound(Q'). The problem of determining if a variable can occur in the domain of all or some of the mappings is in general undecidable [9]. Hence, the notions of inScope and stronglyBound are approximations of these more complex notions. The variables in inScope(Q) are a superset of the variables that occur in some mappings $\mu \in [\![Q]\!]_G$ for every graph G. The variables in stronglyBound(Q) are a subset of the variables that occur in all mappings $\mu \in [\![Q]\!]_G$ for every graph G.

3.3 Federated Query Processing in RDF/SPARQL

Federated querying consists of executing queries across multiple, potentially heterogeneous, RDF data sources, so that they are treated as a single and unified dataset.

Formally [1], a federation F is a pair (E, d) where E is the set of all data sources, and d is function that associates every data source $e \in E$ with an RDF graph. A second evaluation function $[\![\cdot]\!]_F$ is defined in terms of the evaluation function over graphs $[\![\cdot]\!]_G$. For a local SPARQL query Q, $[\![Q]\!]_F = [\![Q]\!]_G$ where the graph G, called the *union graph*, is defined as follows:

$$G = [\![Q]\!]_{\bigcup_{e \in E} d(e)}.$$

This ensures that relationships spanning multiple datasets are considered during evaluation, enabling integrated query processing across distributed datasets.

Notice that query Q is local. That is, Q does not contain SERVICE clauses. However, the federated query processing of Q consists of execution Q over a set of graphs from remote services. Since the SERVICE clause allows for remote query execution, one may think that a federated query processing can be implemented by rewriting the local query Q as a fully remote query. Indeed, the following subsection describes how federated SPARQL query engines decompose the local query Q into multiple queries that are remotely executed by rewriting Q as a single fully remote query that uses the SERVICE for the remote execution.

3.3.1 Data Integration and Query Decomposition

In federated querying, data integration requires handling schema heterogeneity across different RDF graphs. Query decomposition techniques decompose the original SPARQL query into subqueries. Each subquery is evaluated on a subset of the federated endpoints, and the results are merged based on join conditions, compatibility mappings, and downstream algebraic operators.

Queries can be decomposed in different ways. The simpler way is decomposing basic graph patterns into triple patterns to compute unions first and then joins. For each triple pattern in a basic graph pattern, the federated engine generates an intermediate relation that consists of the union of the results obtained from evaluating the triple pattern in all the data sources. These relations are then joined to obtain the results of the triple pattern.

▶ **Example 12.** Let Q be the following query asking for the cities of Germany that are not crossed by the Rhine river:

$$Q = (\text{SELECT } \{\texttt{?city}\} \text{ WHERE } (((\texttt{?city}, \mathsf{a}, \mathsf{City}) \text{ AND } (\texttt{?city}, \mathsf{country}, \mathsf{Germany})) \text{ DIFF } (\mathsf{Rhine}, \mathsf{crosses}, \texttt{?city}))).$$

To answer this query over a federated dataset $D = \{y_1 \mapsto G_1, y_2 \mapsto G_2\}$, we can first answer the following triple patterns over the federation:

$T_1 = (\texttt{?city}, \mathsf{a}, \mathsf{City})$,
$T_2 = (\texttt{?city}, \mathsf{country}, \mathsf{Germany})$,
$T_3 = (\mathsf{Rhine}, \mathsf{crosses}, \texttt{?city})$.

The federated computation of a triple pattern can be done by computing the triple on each SPARQL endpoint, and then computing the union of the results. For example,

$$[\![T_1]\!]_{G_1 \cup G_2} = [\![T_1]\!]_{G_1} \cup [\![T_1]\!]_{G_2}.$$

This union can be expressed as a query execution over the RDF-star dataset D with the SPARQL-star operator SERVICE and UNION.

$$[\![T_1]\!]_{G_1 \cup G_2} = [\![((\text{SERVICE } y_2\ T_1)) \text{ UNION } ((\text{SERVICE } y_1\ T_1))]\!]_D.$$

Following this approach, the whole federated execution of the query can be done by evaluating another query Q' over the federated dataset D (i.e., $[\![Q]\!]_{G_1 \cup G_2} = [\![Q']\!]_D$). Such a query Q' is the following:

$$Q' = (\text{SELECT } \{\texttt{?city}\} \text{ WHERE } (((((\text{SERVICE } y_2\ T_1)) \text{ UNION } ((\text{SERVICE } y_1\ T_1))) \text{ AND } (((\text{SERVICE } y_2\ T_2)) \text{ UNION } ((\text{SERVICE } y_1\ T_2)))) \text{ DIFF } (((\text{SERVICE } y_2\ T_3)) \text{ UNION } ((\text{SERVICE } y_1\ T_3))))).$$

Schwarte et al. [43] proposed a query decomposition that checks if two or more triple patterns are exclusive to one of the sources in the federation. If this is the case, they evaluate the basic graph pattern on that specific data source. The evaluation of a basic graph pattern in a single data source reduces the data transfer across the network required by the simple unions-first-joins-later approach.

▶ **Example 13.** Consider the queries Q and Q', and the federated dataset D described in Example 12. If we know that $[\![T_1]\!]_{G_2}$, $[\![T_2]\!]_{G_2}$, and $[\![T_3]\!]_{G_1}$ are empty, then the result of evaluating Q' in the federation will be equal to the result of evaluating the following query Q'' on D:

$$Q'' = (\text{SELECT } \{\texttt{?city}\} \text{ WHERE } (((\text{SERVICE } y_1\ (T_1 \text{ AND } T_2))) \text{ DIFF } ((\text{SERVICE } y_2\ T_3)))).$$

Aimonier-Davat et al. [3] proposed a joins-first-unions-later query decomposition. They noticed that, often, only a few combinations of sources return results. Following this observation, they decompose queries using into joins of answers to triple patterns. The union of these joins is then computed to obtain the answers of a basic graph pattern. Although, in general, the efficiency depends on the federation, Aimonier-Davat et al. showed that their decomposition outperforms others in some large-scale federations.

3.4 How-provenance in SPARQL

3.4.1 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 [20]. 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.

3.4.2 $\mathcal{K}$-relations and $\mathcal{K}$-graphs

Given a set A and an spm-semiring $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, -_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$, a function $f : A \to K$ is called a *$\mathcal{K}$-set* over A. The set $\text{supp}(f) = \{a \in A \mid f(a) \neq 0_{\mathcal{K}}\}$ is called the support of f. For every element $a \in A$, we call $f(a)$ the *$\mathcal{K}$-value* of a in f. If f has a finite support $\{a_1, \ldots, a_n\}$, we write $f = \{\!\!\{a_1 \mapsto k_1, \ldots, a_n \mapsto k_n\}\!\!\}$ to denote that $f(a_i) = k_i$, for $1 \leq i \leq n$.

Intuitively, $\mathcal{K}$-sets annotate the elements of a set A with values in the structure. By assuming that $\mathcal{K}$ is the structure of provenance annotations, this formalism is used to represent provenance of answer to queries or statements in an RDF graph. 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 a *$\mathcal{K}$-graph*.

▶ Note 14. Note that a $\mathcal{K}$-set is a total function $f : A \to K$. The double braces in the notation $f = \{\!\!\{a_1 \mapsto k_1, \ldots, a_n \mapsto k_n\}\!\!\}$ tell us that the function f is not only defined over the set $\{a_1, \ldots, a_n\}$, which is the support of f, but over the whole set A. This notation omits the elements that are not in the support of f because we already know their values are $0_{\mathcal{K}}$.

▶ **Example 15.** 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:

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

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

It is easy to see that the predicate *wasDerivedFrom* defines a $\mathcal{K}$-graph Γ where the first and last triples are associated to the elements $u_1 \oplus u_2$ and u_3, and that $\mu = \{\mathsf{?x} \mapsto \mathsf{Alice}\}$ is a solution mapping for Q. The $\mathcal{K}$-relation $\Omega = \{\!\!\{\mu \mapsto (u_1 \oplus u_2) \otimes u_3\}\!\!\}$ associates Q's solutions to provenance annotation. Even when the domain of Ω is a infinite set of solution mappings, only the non-zero element for μ is needed to express the function Ω. 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.

▶ **Definition 16.** *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 obtained from evaluating Q in Γ, which is defined recursively for an arbitrary mapping μ as follows:*

- $(\!|\{\}|\!)_\Gamma(\mu) = 1$ *if* $\mathrm{dom}(\mu) = \emptyset$, *and* $(\!|\{\}|\!)_\Gamma(\mu) = 0$ *if* $\mathrm{dom}(\mu) \neq \emptyset$,
- $(\!|(s,p,o)|\!)_\Gamma(\mu) = \Gamma(\mu(s,p,o))$,
- $(\!|\textsc{select}\, W \textsc{ where } Q|\!)_\Gamma(\mu) = \sum_{\mu':\mu'|_W=\mu} (\!|Q|\!)_\Gamma(\mu')$,
- $(\!|Q \textsc{ filter } \varphi|\!)_\Gamma(\mu) = (\!|Q|\!)_\Gamma(\mu) \times_\mathcal{K} 1_{\mu\models\varphi}$, *where* $1_{\mu\models\varphi} = 1$ *if* $\mu \models \varphi$ *and* $1_{\mu\models\varphi} = 0$, *otherwise,*
- $(\!|Q_1 \textsc{ union } Q_2|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) +_\mathcal{K} (\!|Q_2|\!)_\Gamma(\mu)$,
- $(\!|Q_1 \textsc{ and } Q_2|\!)_\Gamma(\mu) = \sum_{\mu=\mu_1\cup\mu_2}((\!|Q_1|\!)_\Gamma(\mu_1) \times_\mathcal{K} (\!|Q_2|\!)_\Gamma(\mu_2))$,
- $(\!|Q_1 \textsc{ diff } Q_2|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) -_\mathcal{K} (\sum_{\mu'\sim\mu} (\!|Q_2|\!)_\Gamma(\mu'))$,
- $(\!|Q_1 \textsc{ optional } Q_2|\!)_\Gamma(\mu) = (\!|Q_1 \textsc{ and } Q_2|\!)_\Gamma(\mu) +_\mathcal{K} (\!|Q_1 \textsc{ diff } Q_2|\!)_\Gamma(\mu)$,

where $\sum$ *denotes sums using the operation* $+_\mathcal{K}$.

3.4.3 How-provenance polynomials

In Example 15 we presented how-provenance annotations. We next describe how a universal spm-semiring *generalizes* all the other spm-semirings for the annotated SPARQL algebra. To understand what is meant by the term "generalize", we next introduce the notions of *spm-homomorphism* and *commutation with homomorphisms.*

▶ **Definition 17** (spm-homomorphism). *An* spm-homomorphism *between two spm-semirings* $\mathcal{K}_1 = (K, +_{\mathcal{K}_1}, \times_{\mathcal{K}_1}, -_{\mathcal{K}_1}, 0_{\mathcal{K}_1}, 1_{\mathcal{K}_1})$ *and* $\mathcal{K}_2 = (K, +_{\mathcal{K}_2}, \times_{\mathcal{K}_2}, -_{\mathcal{K}_2}, 0_{\mathcal{K}_2}, 1_{\mathcal{K}_2})$, *is a function* $h : K_1 \to K_2$ *that preserves the identities and the operations. That is:*

- $h(0_{\mathcal{K}_1}) = 0_{\mathcal{K}_2}$,
- $h(1_{\mathcal{K}_1}) = 1_{\mathcal{K}_2}$,
- $h(a +_{\mathcal{K}_1} b) = h(a) +_{\mathcal{K}_2} h(b)$,
- $h(a \times_{\mathcal{K}_1} b) = h(a) \times_{\mathcal{K}_2} h(b)$,
- $h(a -_{\mathcal{K}_1} b) = h(a) -_{\mathcal{K}_2} h(b)$.

▶ **Example 18.** Consider the following spm-semirings:

- The spm-semiring of natural numbers $(\mathbb{N}, +, \cdot, -, 0, 1)$, where $+$ and $\cdot$ are the usual sum and product of natural numbers, and $a - b = a$ if $b = 0$ and $a - b = 0$ if $b \neq 0$.
- The spm-smiring of boolean values $(\mathbb{B}, \vee, \wedge, \nrightarrow, 0, 1)$, where $\vee$, $\wedge$, and $\nrightarrow$ are the Boolean disjunction, conjunction and material nonimplication.

The function $h : \mathbb{N} \to \mathbb{B}$, defined as $h(k) = \min(1, k)$, is an spm-homomorphism.

So far, we have described when an spm-semiring is more general than another spm-semiring. Geerts et al. [22] constructed the most general semiring, called the universal spm-semiring. To define the universal spm-semiring, let $X = \{x_1, \ldots, x_n\}$ be a set of variables, and $\mathrm{mon}(X)$ be the set of monomials over X. Then, let B_X and $\bar{B}_X$ be two sets disjoint from X, defined as follows: $B_X = \{b_\mu \mid \mu \in \mathrm{mon}(X)\}$ and $\bar{B}_X = \{\bar{b}_\mu \mid \mu \in \mathrm{mon}(X)\}$. We abbreviate the set $X \cup B_X \cup \bar{B}_X$ as X_B, and we write $\mathbb{N}[X_B]$ for the set of polynomials with coefficients in $\mathbb{N}$ and variables X_B.

We write $\cong$ to denote the congruence relation between polynomials in $\mathbb{N}[X_B]$. That is, if $p[X_B] \cong q[X_B]$ and $p'[X_B] \cong q'[X_B]$, then $p[X_B] + p'[X_B] \cong q[X_B] + q'[X_B]$ and $p[X_B] \cdot p'[X_B] \cong q[X_B] \cdot q'[X_B]$. In addition, $\cong$ is assumed to be the minimum congruence such that for all monomials $\mu \in \mathrm{mon}(X)$, $b_\mu \cdot \bar{b}_\mu \cong 0$, $b_\mu + \bar{b}_\mu \cong 1$, $b_\mu + b_\mu \cong b_\mu$ and $\bar{b}_\mu + \bar{b}_\mu \cong \bar{b}_\mu$, $\bar{b}_m \cdot \mu \cong 0$, and $b_\mu \cdot b_\nu \cong b_\mu$ whenever $\mu = \nu \cdot \nu'$ for some monomial ν'. Since these last entities characterize the elements of B_X and $\bar{B}_X$ as Booleans, the quotient semiring of $\mathbb{N}[X_B]$ with respect to $\cong$ is called the *semiring of boolynomials.* Abusing of notation, we write $(\mathbb{N}_\cong[X_B], +, \cdot, [0], [1])$ for this quotient semiring,

where the elements of $\mathbb{N}_{\cong}[X_B]$ are the equivalence classes $[p[X_B]] = \{p'[X_B] \mid p[X_B] \cong p'[X_B]\}$, and the operations are $[p[X_B]] + [q[X_B]] = [p[X_B] + q[X_B]]$ and $[p[X_B]] \cdot [q[X_B]] = [p[X_B] \cdot q[X_B]]$.

The semiring of boolynomials lacks of the difference operator. Geerts et al. [22] propose the following difference: $[p[X_B]] - [q[X_B]] = [p[X_B]] \cdot [p_q]$. To see the formal definition of boolynomial p_q see Geerts et al. [22] work. Intuitively, the boolynomial p_q should contain monomials $\bar{b}_\mu$ such that the product eliminates the monomials μ from boolynomial $p[X_B]$. Geerts et al. [22] proved that such a structure $(\mathbb{N}_{\cong}[X_B], +, \cdot, -, [0], [1])$ is the universal spm-semiring.

In what follows, we write $\text{ProvExp}(X)$ to denote the set of all the expressions generated with set of variables X, the operator symbols $\oplus$, $\otimes$, and $\ominus$, and the constants 0 and 1. The semantics of these expressions is given by associating every expression in $\text{ProvExp}(X)$ to an element in $\mathbb{N}_{\cong}[X_B]$ by mapping every variable $x \in X$ to an equivalence class $[X]$, the constants 0 and 1 with the equivalence classes [0] and [1], and the operators $\oplus$, $\otimes$, and $\ominus$ with the operators $+$, $\cdot$, and $-$ of the universal spm-semiring. The expressions computed by the methods proposed by Hernández et al. [30] and Asma et al. [10] represent thus elements in the universal spm-semiring.

We write $\text{ProvExp}_{\cong}(X)$ for the set of equivalence classes $[e]$ where $e \in \text{ProvExp}(X)$ and $e' \in [e]$ if and only if e and e' are associated to the same element in $\mathbb{N}_{\cong}[X_B]$. The structure $(\text{ProvExp}_{\cong}(X), \oplus, \otimes, \ominus, [0], [1])$ whose operators are $[a] \oplus [b] = [a \oplus b]$, $[a] \otimes [b] = [a \otimes b]$, and $[a] \ominus [b] = [a \ominus b]$ is thus an spm-semiring of equivalence classes of expressions representing elements in the universal spm-semiring.

▶ **Example 19.** Consider the spm-semiring $\text{ProvExp}_{\cong}(X)$, an arbitrary spm-semiring $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, -_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$, and a function $g : X \to K$. Let $h : \text{ProvExp}_{\cong}(X) \to \mathcal{K}$ be the function defined recursively as follows:

- $h([x]) = g(x)$ if $x \in X$.
- $h([0]) = 0_{\mathcal{K}}$ and $h([1]) = 1_{\mathcal{K}}$.
- $h([a \oplus b]) = h([a]) +_{\mathcal{K}} h([b])$, $h([a \otimes b]) = h([a]) \times_{\mathcal{K}} h([b])$, and $h([a \ominus b]) = h([a]) -_{\mathcal{K}} h([b])$.

The function h is an spm-homomorphism.

For readability, in what follows we will omit the brackets for the elements of $\text{ProvExp}_{\cong}(X)$. For example, we will write that a triple in a $\text{ProvExp}_{\cong}(X)$-graph is annotated with a variable x to indicate that it is annotated with the element $[x]$ in the spm-semiring over set $\text{ProvExp}_{\cong}(X)$.

▶ **Definition 20** (Commutation with homomorphisms)**.** *Given two spm-semirings $\mathcal{K}_1$ and $\mathcal{K}_2$. We say that the $\mathcal{K}_1$- and $\mathcal{K}_2$-annotated SPARQL algebras* commute with homomorphisms *if and only if for every spm-homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$, every SPARQL query Q and every $\mathcal{K}_1$-graph Γ_1, it holds that $(\!|Q|\!)_{\Gamma_1} \circ h = (\!|Q|\!)_{\Gamma_1 \circ h}$, where $\circ$ denotes the composition of functions.*

▶ **Example 21.** Let Γ_1 be the $\text{ProvExp}_{\cong}(X)$-graph whose support has two triples annotated as follows: $\{\!|(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) \mapsto x_1, (\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}) \mapsto x_2|\!\}$, where x_1 and x_2 are elements in set X. Let $g : X \to \mathbb{N}$ be the function such that $g(x_1) = 3$ and $g(x_2) = 5$, and $h : \text{ProvExp}_{\cong}(X) \to \mathbb{N}$ be the spm-homomorphism defined on top of function g as it is described in Example 19. Then, $\Gamma_2 = \Gamma_1 \circ h$ is the $\mathbb{N}$-graph $\{\!|(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) \mapsto 3, (\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}) \mapsto 5|\!\}$. Let Q be the query (SELECT `?river` WHERE $((\texttt{?river}, \mathsf{crosses}, \texttt{?city}))$), asking for rivers crossing cities. Then, $(\!|Q|\!)_{\Gamma_1}$ is the $\mathbb{N}$-relation $\{\!|\mathsf{Danube} \mapsto x_1 \oplus x_2|\!\}$. Then $(\!|Q|\!)_{\Gamma_1} \circ h$ is the $\mathbb{N}$-relation $\{\!|\mathsf{Danube} \mapsto 8|\!\}$, which is equal to $(\!|Q|\!)_{\Gamma_2}$.

▶ Remark 22. Example 21 illustrates the generality of how-provenance polynomials. It is well-known that every pair of SPARQL algebras commute with homomorphisms. Also, by using the

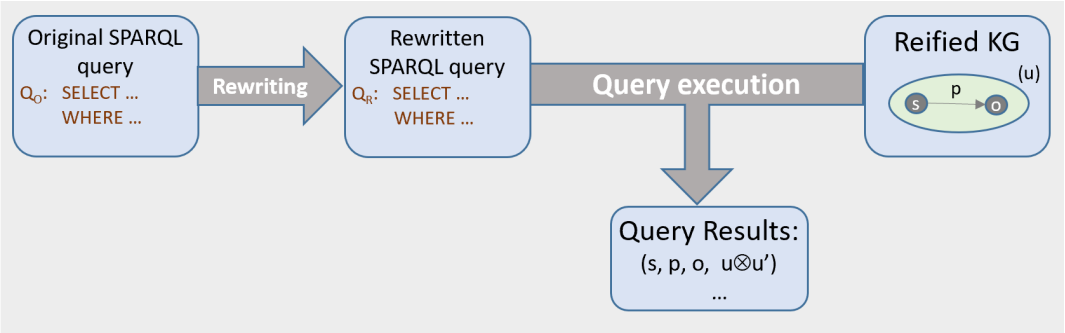

Figure 1 The query rewriting process for NPCS

spm-homomorphism like the one described in Example 19, we can use how-provenance polynomials to symbolically operate elements of other spm-semirings. Thus, the how-provenance polynomials generated with the method we propose in the next section, NPCS, can be used in downstream tasks over other spm-semirings.

4 Provenance computation with NPCS

Having introduced all the preliminary concepts in the previous section, we now introduce our solution to compute how-provenance annotations for query solutions delivered by a single source. Section 5 will show how to port this method to a federated setting.

Given a finite set $X \subset \mathbf{I}$, an X-graph Γ, its corresponding RDF-star graph G, and a SPARQL query Q_O, NPCS rewrites Q_O into Q_R so that $[\![Q_R]\!]_G$ is a set of mappings with an additional variable that encodes the provenance polynomials. Thus, the output of Q_R represents the $\mathrm{ProvExp}_{\cong}(X)$-relation $[\![Q_O]\!]_\Gamma$ that maps each solution to a how-provenance polynomial explanation. Those polynomials lie in an spm-semiring $(\mathrm{ProvExp}_{\cong}(X), \oplus, \otimes, \ominus, 0, 1)$, where X 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 query rewriting process of NPCS is depicted in Figure 1.

4.1 Our Query Rewriting in a Nutshell

Consider the graphs Γ and G and a query Q asking for people who like pasta and live in Italy (see Example 15). For pedagogical reasons, we rewrite our query Q as $(P_1 \text{ AND } P_2)$, where P_1 is the triple pattern $(\texttt{?x}, \mathsf{likes}, \mathsf{pasta})$ and P_2 is the triple pattern $(\texttt{?x}, \mathsf{livesIn}, \mathsf{Italy})$. We recall that our goal is to return the following result set:

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

The column labeled `?prov` 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 $P_1 = (\texttt{?x}, \mathsf{likes}, \mathsf{pasta})$ 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:

$$P_1' = (\text{SELECT} \quad \texttt{?x}\ (\text{ProvAggSum}(\texttt{?prov}\oplus\otimes 1\oplus\odot)\ \text{AS}\ \texttt{?prov}\oplus\otimes 1) \\ \text{WHERE} \quad \mathsf{Reify}((\texttt{?x}, \mathsf{likes}, \mathsf{pasta}), \texttt{?prov}\oplus\otimes 1\oplus\odot) \\ \text{GROUP BY} \quad \texttt{?x}),$$

▶ Note 23. For readability, we use the symbols $\oplus$ and $\otimes$ in the variables. Actually, NPCS writes the variable ?prov$\oplus\otimes$1 as ?provSumProdOne. These symbols are just a convention we follow to create fresh variables and avoid clashes between variable names.

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

$$((\texttt{?x}, \mathsf{likes}, \mathsf{pasta}), \mathsf{wasDerivedFrom}, \texttt{?prov}\oplus\otimes 1\oplus\odot).$$

The intermediate variable ?prov$\oplus\otimes$1$\oplus\odot$ is introduced to capture the identifiers of the triples that match P_1, whereas variable ?prov$\oplus\otimes$1 groups all the triple identifiers 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 preceded 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:

$$Q' = (\ \text{SELECT} \quad \texttt{?x}\ (\text{ProvAggSum}(\texttt{?prov}\oplus)\ \text{AS}\ \texttt{?prov}) \\ \text{WHERE} \quad ((P_1'\ \text{AND}\ P_2')\ \text{BIND}\ (\text{ProvProd}(\texttt{?prov}\oplus\otimes 1, \texttt{?prov}\oplus\otimes 2)\ \text{AS}\ \texttt{?prov}\oplus)) \\ \text{GROUP BY} \quad \texttt{?x}\ ).$$

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 24.** *Let* ?x, ?x$_1$,..., ?x$_n$ *be variables. Then, we define the following SPARQL operators based on the built-in SPARQL functions* concat *and* aggregate_concat*:*

$$\text{ProvAggSum}(\texttt{?x}) = \texttt{concat}(\ \texttt{"(}\oplus\texttt{"},\ \texttt{aggregate\_concat}(\texttt{?x}),\ \texttt{")"}\ ),$$
$$\text{ProvProd}(\texttt{?x}_1, \ldots, \texttt{?x}_n) = \texttt{concat}(\ \texttt{"(}\otimes\texttt{"},\ \texttt{?x}_1, \ldots, \texttt{?x}_n,\ \texttt{")"}\ ),$$
$$\text{ProvDiff}(\texttt{?x}_1, \texttt{?x}_2) = \texttt{concat}(\ \texttt{"(}\ominus\texttt{"},\ \texttt{?x}_1, \texttt{?x}_2,\ \texttt{")"}\ ).$$

▶ **Example 25.** The expression ProvProd(?prov$\oplus\otimes$1, ?prov$\oplus\otimes$2) in the query Q' of our running example is evaluated against the answers of the query (P_1' AND P_2'), which are described in the following result set:

?x	?prov$\oplus\otimes$1	?prov$\oplus\otimes$2
Alice	(u_1)	(u_3)
Alice	(u_2)	(u_3)

Then, this expression generates a third provenance column for the variable `?prov`$\oplus$:

?x	?prov$\oplus\otimes$1	?prov$\oplus\otimes$2	?prov$\oplus$
Alice	(u_1)	(u_3)	$(\otimes(u_1)(u_3))$
Alice	(u_2)	(u_3)	$(\otimes(u_2)(u_3))$

By aggregating these two results with the expression ProvAggSum(`?prov`$\oplus$) we obtain a final provenance value annotated in variable `?prov`:

?x	?prov
Alice	$(\oplus(\otimes(u_1)(u_3))(\otimes(u_2)(u_3)))$

▶ Note 26. The expression $(\oplus(\otimes(u_1)(u_3))(\otimes(u_2)(u_3)))$ obtained in Example 25 is a how-provenance polynomial written in Polish notation, and is equivalent to the aforementioned how-provenance polynomial $(u_1 \oplus u_2) \otimes u_3$ we want to generate. We use Polish notation because it integrates seamlessly with the built-in SPARQL function `aggregate_concat`.

▶ Note 27. In this section we present a query rewriting from the SPARQL fragment described in Definition 16 to a SPARQL fragment that include operations that are not present in Definition 16. For example, the operation BIND. The rational of this discrepancy between both SPARQL fragments is that we assume we can use whatever we have at hand in an standard SPARQL engine to compute the how-provenance of a SPARQL query in a fragment in Definition 16, which corresponds to the one studied by Geerts et al. [22]. We assume that the reader is familiar with SPARQL, and we include the formal definitions of these additional operations in the proofs of the correctness of the method proposed in this paper.

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 28** (Base SPARQL-star query rewriting). *Let Q be a local SPARQL query,* `?prov` *a variable, and* Reify *a reification scheme. Then, the rewritten query for Q and variable* `?prov` *over scheme* Reify*, denoted $\beta(Q, \texttt{?prov})$, is defined recursively as follows:*

1. *If Q is an empty basic graph pattern, then $\beta(Q, \texttt{?prov})$ is the query* ({} BIND (1 AS `?prov`)).
2. *If Q is a triple pattern (s, p, o), then $\beta(Q, \texttt{?prov})$ is the query*

```
( SELECT    inScope(Q) (ProvAggSum(?prov⊕) AS ?prov)
  WHERE     Reify((s, p, o), ?prov⊕)
  GROUP BY  inScope(Q) ).
```

3. *If Q is $(Q_1$ AND $Q_2)$, then $\beta(Q, \texttt{?prov})$ is the query*

```
( SELECT    inScope(Q) (ProvAggSum(?prov⊕) AS ?prov)
  WHERE     (β(Q1, ?prov⊕⊗1) AND β(Q2, ?prov⊕⊗2))
            BIND (ProvProd(?prov⊕⊗1, ?prov⊕⊗2) AS ?prov⊕)
  GROUP BY  inScope(Q) ).
```

4. *If Q is $(P_1$ UNION $P_2)$, then $\beta(Q, \texttt{?prov})$ is the query*

```
( SELECT    inScope(Q) (ProvAggSum(?prov⊕) AS ?prov)
  WHERE     (β(Q1, ?prov⊕) UNION β(Q2, ?prov⊕))
  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* $\mathrm{dom}(Q_1) \cap \mathrm{dom}(Q_2)$ *that are not strongly bound in query Q_1. Then, $\beta(Q, \texttt{?prov})$ is the query*

$$\begin{array}{ll}(\ \text{SELECT} & \mathrm{inScope}(Q)\ (\mathrm{ProvDiff}(\texttt{?prov}\ominus 1, \mathrm{ProvAggSum}(\texttt{?prov}\ominus 2\oplus))\ \text{AS}\ \texttt{?prov}) \\ \text{WHERE} & (\beta(Q_1, \texttt{?prov}\ominus 1)\ \text{OPTIONAL}_{C_\nu}\ \beta(\nu(Q_2), \texttt{?prov}\ominus 2\oplus)) \\ \text{GROUP BY} & \mathrm{inScope}(Q) \cup \{\texttt{?prov}\ominus 1\}\ ).\end{array}$$

6. *If Q is* (SELECT W WHERE Q')*, then $\beta(Q, \texttt{?prov})$ is the query*

$$\begin{array}{ll}(\ \text{SELECT} & W\ (\mathrm{ProvAggSum}(\texttt{?prov}\oplus)\ \text{AS}\ \texttt{?prov}) \\ \text{WHERE} & \beta(Q', \texttt{?prov}\oplus) \\ \text{GROUP BY} & W\ ).\end{array}$$

7. *If Q is $(Q'$ FILTER $\varphi)$, then $\beta(Q, \texttt{?prov})$ is the query $(\beta(Q', \texttt{?prov})$ FILTER $\varphi)$.*

▶ Note 29. The functions inScope, ProvAggSum, ProvDiff, ProvProd, and Reify are not SPARQL algebra operators (like AND and UNION), but are used to define actual SPARQL expressions. For example, inScope(Q) must be replaced by the set of variables that are in scope of query Q, and ProvAggSum(?prov$\oplus$) must be replaced by the actual SPARQL expression according to Definition 24.

▶ Note 30. 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, the query P_1 OPTIONAL P_2 is equivalent to the query $(P_1$ DIFF $P_2)$ UNION $(P_1$ AND $P_2)$.

▶ Note 31. 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 renaming discards undesired bindings produced in the subtrahend while tracking the provenance. For example, consider the patterns $P_1(\texttt{?x}, \texttt{?y})$ and $P_2(\texttt{?x}, \texttt{?y}, \texttt{?z})$, whose in-scope variables are indicated in the parenthesis, and let $\mu_1 = \{\texttt{?x} \mapsto a\}$ and $\mu_2 = \{\texttt{?x} \mapsto a, \texttt{?y} \mapsto b, \texttt{?z} \mapsto c\}$ be two solutions for the patterns P_1 and P_2. Our goal is to design an operation that returns a mapping with all variable bindings for variables ?x and ?y from pattern P_1 but excluding variable bindings from pattern P_2. The challenge is that it does not suffices simply discarding the variable ?z, which occurs only in pattern P_2, but also considering the variable ?y, which is unbound in pattern P_1. If instead of OPTIONAL_{C_v}, the rule for DIFF (rule 5) used OPTIONAL, the query would return the provenance for the mapping $\{\texttt{?x} \mapsto a, \texttt{?y} \mapsto b\}$ instead of the mapping μ_1. That we would returned a value for variable ?y that is bound in pattern P_2 but not in pattern P_1. Instead, if ?x is strongly bound for P_1 (i.e., always bound in its answers) and ?y is not, then the operation P_1 OPTIONAL_{C_v} P_2 is:

$$\begin{array}{l}((P(\texttt{?x}, \texttt{?y})\ \text{OPTIONAL}\ P(\texttt{?x}, \texttt{?u}, \texttt{?z})) \\ \ \ \text{FILTER}\ (\neg\,\mathrm{bound}(\texttt{?y}) \vee \neg\,\mathrm{bound}(\texttt{?u}) \vee \texttt{?y} = \texttt{?u})).\end{array}$$

where $\texttt{?u} = \nu(\texttt{?y})$ is a fresh variable introduced by the variable renaming function $\nu : \mathbf{V} \to \mathbf{V}$. The result of this operation for the aforementioned mappings is the mapping $\{\texttt{?x} \mapsto a, \texttt{?u} \mapsto b, \texttt{?z} \mapsto c\}$. Since variables ?u and ?z are not in-scope variables of pattern P_1, can be discarded to obtain the actual answer $\mu = \{\texttt{?x} \mapsto a\}$ of the query $(Q_1$ DIFF $Q_2)$.

Formally, the operation $(Q_1$ OPTIONAL_{C_ν} $Q_2)$ has the form $((Q_1$ OPTIONAL $\nu(Q_2))$ FILTER $C_\nu)$ where ν is a function mapping in-scope variables in Q_1 that are not strongly-bound to fresh variables, $\nu(Q_2)$ is the query resulting from renaming in Q_2 the variables $\texttt{?y} \in \mathrm{dom}(\nu)$ with $\nu(\texttt{?y})$, and C_ν is a conjunctions of selection conditions that check the compatibility between the values of such pairs of variables ?y and ν(?y) (i.e., formulas of the form $\neg\,\mathrm{bound}(\texttt{?y}) \vee \neg\,\mathrm{bound}(\nu(\texttt{?y})) \vee \texttt{?y} = \nu(\texttt{?y})$).

4.3 Correctness criteria

To define correctness of the query rewriting described in Definition 28 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 32** (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* $\texttt{?prov} \notin \text{inScope}(Q)$*, and returns a SPARQL query* $\gamma(Q, \texttt{?prov})$ *with* $\text{inScope}(\gamma(Q, \texttt{?prov})) = \text{inScope}(Q) \cup \{\texttt{?prov}\}$*. Let* Γ *be an* X*-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. Then:*

- γ *is called* sound *for* Reify *if, for every answer of the rewritten query* $\mu \cup \{\texttt{?prov} \mapsto e\}$ *in* $[\![\gamma(Q, \texttt{?prov})]\!]_G$*,* e *is an expression for the polynomial* $(\!|Q|\!)_\Gamma(\mu)$*.*
- 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* $\mu \cup \{\texttt{?prov} \mapsto e\} \in [\![\gamma(Q, \texttt{?prov})]\!]_G$*.*

▶ **Theorem 33.** *Let* Reify *be a reification scheme and* β *be the function described in Definition 28. Then, function* β *is sound and complete for the reification scheme* Reify*.*

Proof. It can be shown by induction on the query structure. There are two base cases. First, when the query Q is an empty basic graph pattern, then we know that for every annotated graph Γ, $(\!|Q|\!)_\Gamma = \{\!|\mu_\emptyset \mapsto 1|\!\}$. Similarly, for every graph G, $[\![(\{\} \text{ BIND } (1 \text{ AS } \texttt{?prov}))]\!]_G = \{\{\texttt{?prov} \mapsto 1\}\}$, which corresponds to the $\mathcal{K}$-relation $\{\!|\mu_\emptyset \mapsto 1|\!\}$. Second, when the query is a triple pattern T, then $(\!|Q|\!)_\Gamma = \{\!|\mu_t \mapsto \Gamma(t)|\!\}$, where $\text{dom}(\mu_t)$ are the variables occurring in T and $\mu_t(T) = t$. If G is a corresponding graph for Γ, then $[\![\text{Reify}(T, \texttt{?prov}\oplus)]\!]_G$ has the form $\{\mu_t \cup \{\texttt{?prov}\oplus \mapsto k_1\}, \ldots, \mu_t \cup \{\texttt{?prov}\oplus \mapsto k_n\}\}$, where $\sum_1^n k_i = \Gamma(t)$. This sum is required because regular graphs G representing an annotated graph Γ can include the same triple t multiple times (e.g., when they are the result of aggregating multiple data sources). The function ProvAggSum($\texttt{?prov}\oplus$) in query $\beta(Q, \texttt{?prov})$ will then return the expression $(\oplus\, k_1 \ldots k_n)$, which is the expression for the polynomial $\Gamma(t)$. Hence, $[\![\beta(Q, \texttt{?prov})]\!]_G = \{\mu_t \cup \{\texttt{?prov} \mapsto (\oplus\, k_1 \ldots k_n)\}\}$, which corresponds to the expected $\mathcal{K}$-relation.

We next consider the non-base cases:

Case $Q = (Q_1 \text{ AND } Q_2)$. For every mapping μ and annotated graph Γ,

$$(\!|(Q_1 \text{ AND } Q_2)|\!)_\Gamma(\mu) = \sum_{\mu=\mu_1\cup\mu_2} ((\!|Q_1|\!)_\Gamma(\mu_1) \otimes (\!|Q_2|\!)_\Gamma(\mu_2)).$$

Similarly for a corresponding regular graph G for the annotated graph Γ, the set $[\![\beta(Q, \texttt{?prov})]\!]_G$ has the form $\{\mu'_1 \cup \{\texttt{?prov} \mapsto k_1\}, \ldots, \mu'_m \cup \{\texttt{?prov} \mapsto k_m\}\}$, where $\mu'_1, \ldots, \mu'_m$ are distinct mappings. By construction, for $1 \le j \le m$, k_j has the form $(\oplus\, (\otimes\, k_1^1\ k_2^1) \cdots (\otimes\, k_1^p\ k_2^p))$, and for $1 \le i \le p$, there exists two mappings μ_1^i and μ_2^i such that

- $\mu_1^i \cup \{\texttt{?prov}\oplus\otimes 1 \mapsto k_1^i\} \in [\![\beta(Q_1, \texttt{?prov}\oplus\otimes 1)]\!]_G$,
- $\mu_2^i \cup \{\texttt{?prov}\oplus\otimes 2 \mapsto k_2^i\} \in [\![\beta(Q_2, \texttt{?prov}\oplus\otimes 2)]\!]_G$.

By the induction hypothesis, the query rewriting is sound for the queries Q_1 and Q_2. Thus, k_1^i and k_2^i represent $(\!|Q_1|\!)_\Gamma(\mu_1^i)$ and $(\!|Q_1|\!)_\Gamma(\mu_2^i)$, respectively. The rewriting is sound if the expression $(\oplus\, (\otimes\, k_1^1\ k_2^1) \cdots (\otimes\, k_1^p\ k_2^p))$ represents $(\!|Q|\!)_\Gamma(\mu'_j)$ if it contains all then non-zero expressions for the mappings μ_1 and μ_2 such that $\mu'_j = \mu_1 \cup \mu_2$. This requirement holds by the induction hypothesis: the query rewriting is complete for the queries Q_1 and Q_2. Hence, the rewriting is sound.

To prove the completeness, we need to show that every mapping μ in the support of $(\!|Q|\!)_\Gamma$ is one of the aforementioned mappings $\mu'_1, \ldots, \mu'_m$. Since $\mu \in \text{supp}((\!|Q|\!)_\Gamma)$, there exists at least two mappings $\mu_1 \in \text{supp}((\!|Q_1|\!)_\Gamma)$ and $\mu_2 \in \text{supp}((\!|Q_2|\!)_\Gamma)$ such that $\mu = \mu_1 \cup \mu_2$. By the induction hypothesis, these mappings appear in the solutions of queries $\beta(Q_1, \texttt{?prov}\oplus\otimes 1)$ and $\beta(Q_2, \texttt{?prov}\oplus\otimes 2)$, respectively. Then, μ appears in the solutions of query $(\beta(Q_1, \texttt{?prov}\oplus\otimes 1)$ AND $\beta(Q_2, \texttt{?prov}\oplus\otimes 2))$. Then, μ, appears in the solutions of query $\beta(Q, \texttt{?prov})$. Hence, the rewriting is complete.

Case $Q = (Q_1$ UNION $Q_2)$. For every mapping μ and annotated graph Γ,

$$(\!|(Q_1 \text{ UNION } Q_2)|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) \oplus (\!|Q_2|\!)_\Gamma(\mu).$$

Similarly for a corresponding regular graph G for the annotated graph Γ, the set $[\![\beta(Q, \texttt{?prov})]\!]_G$ has the form $\{\mu'_1 \cup \{\texttt{?prov} \mapsto r_1\}, \ldots, \mu'_m \cup \{\texttt{?prov} \mapsto k_m\}\}$, where $\mu'_1, \ldots, \mu'_m$ are distinct mappings. By construction, for $1 \leq j \leq m$, k_j has either the form $(\oplus\ r_1\ r_2)$, $(\oplus\ r_1)$, or $(\oplus\ r_2)$, and

- $\mu'_j \cup \{\texttt{?prov}\oplus \mapsto r_1\} \in [\![\beta(Q_1, \texttt{?prov}\oplus)]\!]_G$ if and only if $k_j = (\oplus\ r_1\ r_2)$ or $k_j = (\oplus\ r_1)$,
- $\mu'_j \cup \{\texttt{?prov}\oplus \mapsto r_2\} \in [\![\beta(Q_2, \texttt{?prov}\oplus)]\!]_G$ if and only if $k_j = (\oplus\ r_1\ r_2)$ or $k_j = (\oplus\ r_2)$.

By the induction hypothesis, the query rewriting is sound and complete for the queries Q_1 and Q_2. Thus, if defined, r_1 and r_2 represent $(\!|Q_1|\!)_\Gamma(\mu'_j)$ and ~~$(\!|Q_1|\!)_\Gamma(\mu'_j)$~~$(\!|Q_2|\!)_\Gamma(\mu'_j)$, respectively. If r_1 or r_2 is undefined, the respective value of $(\!|Q_1|\!)_\Gamma(\mu'_j)$ or ~~$(\!|Q_1|\!)_\Gamma(\mu'_j)$~~$(\!|Q_2|\!)_\Gamma(\mu'_j)$ is 0. Hence, the rewriting is sound.

To prove the completeness, we need to show that every mapping μ in the support of $(\!|Q|\!)_\Gamma$ is one of the aforementioned mappings $\mu'_1, \ldots, \mu'_m$. Since $\mu \in \text{supp}((\!|Q|\!)_\Gamma)$, it must happen that $\mu \in \text{supp}((\!|Q_1|\!)_\Gamma)$ or ~~$\mu \in \text{supp}((\!|Q_1|\!)_\Gamma)$~~$\mu \in \text{supp}((\!|Q_2|\!)_\Gamma)$. By the induction hypothesis, μ appears in the solutions of queries $\beta(Q_1, \texttt{?prov}\oplus)$ or $\beta(Q_2, \texttt{?prov}\oplus)$. Then, μ appears in the solutions of query $(\beta(Q_1, \texttt{?prov}\oplus)$ UNION $\beta(Q_2, \texttt{?prov}\oplus))$. Then, μ, appears in the solutions of query $\beta(Q, \texttt{?prov})$. Hence, the rewriting is complete.

Case $Q = (Q_1$ DIFF $Q_2)$. For every mapping μ and annotated graph Γ,

$$(\!|(Q_1 \text{ DIFF } Q_2)|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) \ominus \sum_{\mu \sim \mu_2} ((\!|Q_2|\!)_\Gamma(\mu_2)).$$

Similarly for a corresponding regular graph G for the annotated graph Γ, the set $[\![\beta(Q, \texttt{?prov})]\!]_G$ has the form $\{\mu'_1 \cup \{\texttt{?prov} \mapsto k_1\}, \ldots, \mu'_m \cup \{\texttt{?prov} \mapsto k_m\}\}$, where $\mu'_1, \ldots, \mu'_m$ are distinct mappings. By construction, for $1 \leq j \leq m$, k_j has the form $(\ominus\ k\ (\oplus\ k_1 \cdots k_p))$, and

- $\mu'_j \cup \{\texttt{?prov}\ominus 1 \mapsto k\} \in [\![\beta(Q_1, \texttt{?prov}\ominus 1)]\!]_G$,
- for $1 \leq i \leq p$, there is a mapping μ_i, such that $\mu_i \cup \{\texttt{?prov}\ominus 2\oplus \mapsto k_i\} \in [\![\beta(Q_2, \texttt{?prov}\ominus 2\oplus)]\!]_G$ and $\mu_i \sim \mu'_j$.

By the induction hypothesis, the query rewriting is sound and complete for the queries Q_1 and Q_2. ~~Thus, k and $(\oplus\ k_1 \ldots k_p)$ represent $(\!|Q_1|\!)_\Gamma(\mu'_j)$ and $\sum_{\mu_i \sim \mu'_j} (\!|Q_1|\!)_\Gamma(\mu^i_2)$.~~ Thus, k and $(\oplus\ k_1 \ldots k_p)$ represent $(\!|Q_1|\!)_\Gamma(\mu'_j)$ and $\sum_{\mu_i \sim \mu'_j} (\!|Q_2|\!)_\Gamma(\mu_i)$. Hence, the rewriting is sound.

To prove the completeness, we need to show that every mapping μ in the support of $(\!|Q|\!)_\Gamma$ is on of the aforementioned mappings $\mu'_1, \ldots, \mu'_m$. Since $\mu \in \text{supp}((\!|Q|\!)_\Gamma)$, also $\mu \in \text{supp}((\!|Q_1|\!)_\Gamma)$. By the induction hypothesis, μ appears in the solutions of queries $\beta(Q_1, \texttt{?prov}\ominus 1)$. Then, μ, appears in the solutions of query $\beta(Q, \texttt{?prov})$. Hence, the rewriting is complete.

Case $Q = (\text{SELECT}\, W \text{ WHERE } Q_1)$. For every mapping μ and annotated graph Γ,

$$(\!|(\text{SELECT}\, W \text{ WHERE } Q_1)|\!)_\Gamma(\mu) = \sum_{\mu' : \mu'|_W = \mu} (\!|Q_1|\!)_\Gamma(\mu').$$

Similarly for a corresponding regular graph G for the annotated graph Γ, the set $[\![\beta(Q, \texttt{?prov})]\!]_G$ has the form $\{\mu'_1 \cup \{\texttt{?prov} \mapsto k_1\}, \ldots, \mu'_m \cup \{\texttt{?prov} \mapsto k_m\}\}$, where $\mu'_1, \ldots, \mu'_m$ are distinct mappings. By construction, for $1 \leq j \leq m$, k_j has the form $(\oplus\, k_1 \ldots k_p)$, and for $1 \leq i \leq p$, there exists a mapping μ_i such that $\mu_i|_W = \mu'_j$ and $\mu_i \cup \{\texttt{?prov}\oplus \mapsto k_i\} \in [\![\beta(Q_1, \texttt{?prov}\oplus]\!]_G$. By the induction hypothesis, the query rewriting is sound and complete for query Q_1. Thus, each k_i represents $(\!|Q_1|\!)_\Gamma(\mu_i)$. Hence, the rewriting is sound.

To prove the completeness, we need to show that every mapping μ in the support of $(\!|Q|\!)_\Gamma$ is on of the aforementioned mappings $\mu'_1, \ldots, \mu'_m$. Since $\mu \in \text{supp}((\!|Q|\!)_\Gamma)$, there exists at least a mappings $\mu_1 \in \text{supp}((\!|Q_1|\!)_\Gamma)$ such that $\mu = \mu_1|_W$. By the induction hypothesis, mapping μ_1 appears in the solutions of query $\beta(Q_1, \texttt{?prov}\oplus)$. Then, μ, appears in the solutions of query $\beta(Q, \texttt{?prov})$. Hence, the rewriting is complete.

Case $Q = (Q_1 \text{ FILTER } \varphi)$. For every mapping μ and annotated graph Γ,

$$(\!|(Q_1 \text{ FILTER } \varphi)|\!)_\Gamma(\mu) = (\!|Q_1|\!)_\Gamma(\mu) \otimes 1_{\mu \models \varphi}.$$

Similarly for a corresponding regular graph G for the annotated graph Γ, the set $[\![\beta(Q, \texttt{?prov})]\!]_G$ has the form $\{\mu'_1 \cup \{\texttt{?prov} \mapsto k_1\}, \ldots, \mu'_m \cup \{\texttt{?prov} \mapsto k_m\}\}$, where $\mu'_1, \ldots, \mu'_m$ are distinct mappings. By construction, for $1 \leq j \leq m$, k_j, $\mu'_j \cup \{\texttt{?prov} \mapsto k_i\} \in [\![\beta(Q_1, \texttt{?prov}]\!]_G$ and $\mu_j \models \varphi$. By the induction hypothesis, the query rewriting is sound for query Q_1. Thus, k_j represents the $(\!|Q_1|\!)_\Gamma(\mu'_j)$. Hence, the rewriting is sound.

To prove the completeness, we need to show that every mapping μ in the support of $(\!|Q|\!)_\Gamma$ is on of the aforementioned mappings $\mu'_1, \ldots, \mu'_m$. Since $\mu \in \text{supp}((\!|Q|\!)_\Gamma)$, $\mu \in \text{supp}((\!|Q_1|\!)_\Gamma)$ and $\mu \models \varphi$. By the induction hypothesis, mapping μ appears in the solutions of query $\beta(Q_1, \texttt{?prov})$. Since $\mu \models \varphi$, μ appears in the solutions of query $(\beta(Q_1, \texttt{?prov}) \text{ FILTER } \varphi)$, which are the solutions of query $\beta(Q, \texttt{?prov})$. Hence, the rewriting is complete.

We have proved that the rewriting is sound and complete for the two base cases and all the inductive cases.

◀

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.

4.4 Query Rewriting Optimizations

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 35 we describe an alternative query rewriting that produces equivalent polynomial expressions, but reduces the number of aggregate operations.

▶ **Definition 34.** *A* sum-query Q *is a query such that the query rewriting* β *described in Definition 28 returns a query of the form:*

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

We call query T the pattern *of* $\beta(Q, \texttt{?prov})$.

Note that, according to Definition 28, sum-queries are all queries that match the rules for the triple patterns and the operators AND, UNION, and SELECT (rewriting rules 2–4, and 6).

▶ **Definition 35.** *Let Q be a SPARQL query,* `?prov` *be a variable, and* Reify *a reification scheme. Then, the rewritten query for Q and variable* `?prov` *over scheme* Reify*, denoted $\beta(Q, \texttt{?prov})$, is defined recursively as is specified in Definition 28, 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 $\beta(Q_i, \texttt{?prov}{\oplus}{\otimes}i)$ is T_i, then $\beta(Q, \texttt{?prov})$ is the query*

$$
\begin{array}{ll}
(\text{SELECT} & \text{inScope}(Q)\ (\text{ProvAggSum}(\texttt{?prov}{\oplus})\ \text{AS}\ \texttt{?prov}) \\
\text{WHERE} & ((T_1\ \text{AND} \cdots \text{AND}\ T_n) \\
 & \ \text{BIND}\ (\text{ProvProd}(\texttt{?prov}{\oplus}{\otimes}1, \ldots, \texttt{?prov}{\oplus}{\otimes}n)\ \text{AS}\ \texttt{?prov}{\oplus}) \\
\text{GROUP BY} & \text{inScope}(Q)\ ).
\end{array}
$$

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 $\beta(Q_i, \texttt{?prov}{\oplus})$ is T_i, then $\beta(Q, \texttt{?prov})$ is the query*

$$
\begin{array}{ll}
(\text{SELECT} & \text{inScope}(Q)\ (\text{ProvAggSum}(\texttt{?prov}{\oplus})\ \text{AS}\ \texttt{?prov}) \\
\text{WHERE} & (T_i\ \text{UNION} \cdots \text{UNION}\ T_n) \\
\text{GROUP BY} & \text{inScope}(Q)\ ).
\end{array}
$$

3. *If Q is $(Q_1 \text{ DIFF } Q_2)$, 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, then $\beta(Q, \texttt{?prov})$ is the query*

$$
\begin{array}{ll}
(\text{SELECT} & \text{inScope}(Q)\ (\text{ProvDiff}(\texttt{?prov}{\ominus}1, \text{ProvAggSum}(\texttt{?prov}{\ominus}2{\oplus}))\ \text{AS}\ \texttt{?prov}) \\
\text{WHERE} & (\beta(Q_1, \texttt{?prov}{\ominus}1)\ \text{OPTIONAL}_{C_\nu}\ \beta(\nu(Q_2), \texttt{?prov}{\ominus}2{\oplus})) \\
\text{GROUP BY} & \text{inScope}(Q) \cup \{\texttt{?prov}{\ominus}1\}\ ).
\end{array}
$$

4. *If Q is $(\text{SELECT } W \text{ WHERE } Q')$, where Q' is a sum-query, then $\beta(Q, \texttt{?prov})$ is*

$$
\begin{array}{ll}
(\text{SELECT} & W\ (\text{ProvAggSum}(\texttt{?prov}{\oplus})\ \text{AS}\ \texttt{?prov}) \\
\text{WHERE} & \beta(Q', \texttt{?prov}{\oplus}) \\
\text{GROUP BY} & W\ ).
\end{array}
$$

▶ Note 36. In the first two rules of Definition 35, 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 37.** Consider the query Q from Example 15. Then, according to the query rewriting described in Definition 35 the rewritten query of Q is:

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

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 28) has three.

▶ **Theorem 38.** *Let* Reify *be a reification scheme, and β be the function described in Definition 35. Then, function β is sound and complete for the reification scheme* Reify*.*

Proof. We prove this theorem by induction on the query structure. Since we already proved that the rewriting in Definition 28 is sound and complete, it suffices proving that each of the new rewriting rules in Definition 35 produce the same results as the rewriting in Definition 28.

Case $Q = (Q_1 \text{ AND} \cdots \text{AND } Q_n)$. It suffices to prove this case by induction on n. For the base case $n = 2$, both query rewritings produce the same query. For the inductive case, we abbreviate $(Q_1 \text{ AND} \cdots \text{AND } Q_{n-1})$ as $Q_{1,n-1}$. Let Q'_1 be $\beta(((Q_{1,n-1} \text{ AND } Q_n) \text{ AND } Q_{n+1}), \texttt{?prov})$ according to Definition 35, and Q'_2 be $\beta((Q_{1,n-1} \text{ AND } Q_n \text{ AND } Q_{n+1}), \texttt{?prov})$ according to Definition 35. It is not difficult to see that every mapping μ that appears in the answers to query Q'_1 appears in the answers to query Q'_2, and vice versa. Assume that $\mu \cup \{\texttt{?prov} \mapsto k_1\} \in [\![Q'_1]\!]_G$ and $\mu \cup \{\texttt{?prov} \mapsto k_2\} \in [\![Q'_2]\!]_G$. Then, we need to prove that the expressions k_1 and k_2 are equivalent. Let Ω_a, Ω_b, and Ω_c be the respective set of mappings that appear in the support of $(\!|Q_{1,n-1}|\!)_G$, $(\!|Q_{n-1}|\!)_G$, and $(\!|Q_n|\!)_G$. By construction,

$$k_1 = \sum_{\substack{\mu_{ab} \in \Omega_a \bowtie \Omega_b \\ \mu_c \in \Omega_c \\ \mu_{ab} \sim \mu \\ \mu_c \sim \mu}} \Big(\otimes \sum_{\substack{\mu_a \in \Omega_a \\ \mu_b \in \Omega_b \\ \mu_a \sim \mu_{ab} \\ \mu_b \sim \mu_{ab}}} (\otimes\ k_a\ k_b)\ k_c \Big), \qquad k_2 = \sum_{\substack{\mu_a \in \Omega_a \\ \mu_b \in \Omega_b \\ \mu_c \in \Omega_c \\ \mu_a \sim \mu \\ \mu_b \sim \mu \\ \mu_c \sim \mu}} (\otimes\ k_a\ k_b\ k_c),$$

where the $\sum$ denote the expressions $(\oplus\ \ldots)$, $k_a = \Omega_a(\mu_a)$, $k_b = \Omega_b(\mu_b)$, and $k_c = \Omega_c(\mu_c)$. By construction, $\mu_{ab} = \mu_a \cup \mu_b$. Hence, k_1 and k_2 are equivalent.

Case $Q = (Q_1 \text{ UNION} \cdots \text{UNION } Q_n)$. It suffices to prove this case by induction on n. For the base case $n = 2$, both query rewritings produce the same query. For the inductive case, we abbreviate $(Q_1 \text{ UNION} \cdots \text{UNION } Q_{n-1})$ as $Q_{1,n-1}$. Let Q'_1 be $\beta(((Q_{1,n-1} \text{ UNION } Q_n) \text{ UNION } Q_{n+1}), \texttt{?prov})$ according to Definition 35, and Q'_2 be $\beta((Q_{1,n-1} \text{ UNION } Q_n \text{ UNION } Q_{n+1}), \texttt{?prov})$ according to Definition 35. It is not difficult to see that every mapping μ that appears in the answers to query Q'_1 appears in the answers to query Q'_2, and vice versa. Assume that $\mu \cup \{\texttt{?prov} \mapsto k_1\} \in [\![Q'_1]\!]_G$ and $\mu \cup \{\texttt{?prov} \mapsto k_2\} \in [\![Q'_2]\!]_G$. Then, we need to prove that the expressions k_1 and k_2 are equivalent. Let Ω_a, Ω_b, and Ω_c be the respective set of mappings that appear in the support of $(\!|Q_{1,n-1}|\!)_G$, $(\!|Q_{n-1}|\!)_G$, and $(\!|Q_n|\!)_G$. By construction,

$$k_1 = (\oplus\ (\oplus\ k_a\ k_b)\ k_c)\,, \qquad k_2 = (\oplus\ k_a\ k_b\ k_c),$$

where $k_a = \Omega_a(\mu)$, $k_b = \Omega_b(\mu)$, $k_c = \Omega_c(\mu)$, and if any of the terms k_a, k_b, and k_c is 0, then it does not necessarily appears in the expression. Hence, k_1 and k_2 are equivalent.

Case $Q = (Q_1 \text{ DIFF } Q_2)$. Let Q'_1 be $\beta(Q, \texttt{?prov})$ according to Definition 35, and Q'_2 be $\beta(Q, \texttt{?prov})$ according to Definition 35. It is not difficult to see that every mapping μ that appears in the answers to query Q'_1 appears in the answers to query Q'_2, and vice versa. Assume that $\mu \cup \{\texttt{?prov} \mapsto k_1\} \in [\![Q'_1]\!]_G$ and $\mu \cup \{\texttt{?prov} \mapsto k_2\} \in [\![Q'_2]\!]_G$. Then, we need to prove that the expressions k_1 and k_2 are equivalent. By construction,

$$k_1 = (\ominus\ k\ \ (\oplus\ (\oplus\ a_1)\ \ldots\ (\oplus\ a_n)))\,, \qquad k_2 = (\ominus\ k\ (\oplus\ a_1\ \ \ldots\ a_n))\,,$$

where, k is the how-provenance of mapping μ for query Q_1, and for $1 \leq j \leq n$, a_j is a sequence of the form $k_1 \ldots k_m$ that represents the how-provenance of a solution of the pattern of the sum-query Q_2. Hence, k_1 and k_2 are equivalent.

Case $Q = (\text{SELECT}\, W \text{ WHERE } Q_1)$. Let Q'_1 be $\beta(Q, \texttt{?prov})$ according to Definition 35, and Q'_2 be $\beta(Q, \texttt{?prov})$ according to Definition 35. It is not difficult to see that every mapping μ that appears in the answers to query Q'_1 appears in the answers to query Q'_2, and vice versa. Assume that $\mu \cup \{\texttt{?prov} \mapsto k_1\} \in [\![Q'_1]\!]_G$ and $\mu \cup \{\texttt{?prov} \mapsto k_2\} \in [\![Q'_2]\!]_G$. Then, we need to prove that the expressions k_1 and k_2 are equivalent. By construction,

$$k_1 = (\oplus\, (\oplus\, a_1)\ \ldots\ (\oplus\, a_n))\,, \qquad\qquad k_2 = (\oplus\, a_1\ \ldots\ a_n),$$

where, for $1 \leq j \leq n$, a_j is a sequence of the form $k_1 \ldots k_m$ that represents the how-provenance of a solution of the pattern of the sum-query Q_2. Hence, k_1 and k_2 are equivalent.

◀

5 How-provenance of federated SPARQL query computation

In the previous section we have described NPCS, an engine-agnostic method to compute provenance polynomials that explain the answers of SPARQL queries run against a single SPARQL endpoint. They are therefore suitable for a centralized setting. In a federated setting, a query is executed on multiple SPARQL endpoints. Keeping track of the endpoints that contribute to each answer has multiple potential applications: for example, we can use that information to optimize query execution, to control the access to information on materialized views, and to assign trust levels to the retrieved answers. With this in mind, we present an extension to the NPCS system for the federated context, called Fed-NPCS.

5.1 An algebra for Federated Query Computation

We start by introducing an algebraic structure and a set of expressions that extend our how-provenance polynomials to provide explanations for the answers of SPARQL queries run on a federation. The how-provenance polynomials, discussed in the previous section and proposed by Geerts et al. [22] provided the semantics for the expressions in the set ProvExp(X). These expressions combine statement identifiers from a finite set $X \subset \mathbf{I}$ with the operations $\oplus$, $\otimes$, and $\ominus$. We wrote $(\text{ProvExp}_{\cong}(X), \oplus, \otimes, \ominus, [0], [1])$ for the spm-semiring these expressions define. We now show that this algebra can also be used to record the endpoints where the statements are found. Indeed, instead of using how-provenance expressions in ProvExp(X) we can consider expressions in ProvExp$(Y \times X)$ where each pair $(y, x) \in Y \times X$ consists of an element $y \in Y$ that identifies a SPARQL endpoint and an element $x \in X$ that identifies a statement. The following example illustrates the use of these pairs in provenance polynomials.

▶ **Example 39.** Let G_1 and G_2 be two RDF-star graphs defined as follows:

$$\begin{aligned} G_1 = \{&((\mathsf{Ulm}, \mathsf{a}, \mathsf{City}), \mathsf{wasDerivedFrom}, x_1),\\ &((\mathsf{Ulm}, \mathsf{country}, \mathsf{Germany}), \mathsf{wasDerivedFrom}, x_2),\\ &((\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}), \mathsf{wasDerivedFrom}, x_3),\\ &((\mathsf{Budapest}, \mathsf{a}, \mathsf{City}), \mathsf{wasDerivedFrom}, x_4),\\ &((\mathsf{Budapest}, \mathsf{country}, \mathsf{Hungary}), \mathsf{wasDerivedFrom}, x_5)\}.\\ G_2 = \{&((\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}), \mathsf{wasDerivedFrom}, x_3),\\ &((\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}), \mathsf{wasDerivedFrom}, x_6)\}. \end{aligned}$$

Graphs G_1 and G_2 can share statements and identifiers. For example, "The Danube crosses Ulm" is stated in both graphs, and annotated with the IRI x_3. To use the non-federated NPCS

framework we should first merge these two graphs into one. By naming the graph G_1 as y_1 and the graph G_2 as y_2, we can define the following RDF-star graph as the merge of both graphs:

$$\begin{aligned} G = \{&((\mathsf{Ulm}, \mathsf{a}, \mathsf{City}), \mathsf{wasDerivedFrom}, (y_1, x_1)),\\ &((\mathsf{Ulm}, \mathsf{country}, \mathsf{Germany}), \mathsf{wasDerivedFrom}, (y_1, x_2)),\\ &((\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}), \mathsf{wasDerivedFrom}, (y_1, x_3)),\\ &((\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}), \mathsf{wasDerivedFrom}, (y_2, x_3)),\\ &((\mathsf{Budapest}, \mathsf{a}, \mathsf{City}), \mathsf{wasDerivedFrom}, (y_1, x_4)),\\ &((\mathsf{Budapest}, \mathsf{country}, \mathsf{Hungary}), \mathsf{wasDerivedFrom}, (y_1, x_5)),\\ &((\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}), \mathsf{wasDerivedFrom}, (y_2, x_6))\}. \end{aligned}$$

In an abuse of notation, each pair (y_i, x_j) in the merged graph G denotes an IRI that can be used to identify the statement, as we already did with NPCS. The corresponding $(Y \times X)$-graph for G is the graph Γ defined as follows:

$$\begin{aligned} \Gamma = \{\!\{&(\mathsf{Ulm}, \mathsf{a}, \mathsf{City}) && \mapsto (y_1, x_1),\\ &(\mathsf{Ulm}, \mathsf{country}, \mathsf{Germany}) && \mapsto (y_1, x_2),\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) && \mapsto (y_1, x_3),\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) && \mapsto (y_2, x_3),\\ &(\mathsf{Budapest}, \mathsf{a}, \mathsf{City}) && \mapsto (y_1, x_4),\\ &(\mathsf{Budapest}, \mathsf{country}, \mathsf{Hungary}) && \mapsto (y_1, x_5),\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}) && \mapsto (y_2, x_6)\}\!\}. \end{aligned}$$

Let Q_O be the following query, asking for countries with a city crossed by the Danube river:

$$\begin{aligned} &(\ \text{SELECT} \quad \{\texttt{?country}\}\\ &\ \ \text{WHERE} \quad ((\texttt{?city}, \mathsf{a}, \mathsf{City}) \ \text{AND}\ (\texttt{?city}, \mathsf{country}, \texttt{?country}) \ \text{AND}\ (\mathsf{Danube}, \mathsf{crosses}, \texttt{?city}))\ ). \end{aligned}$$

By executing the query Q_O on graph Γ, the answers Germany and Hungary are explained by the following how-provenance polynomials:

$$(\!|Q_O|\!)_\Gamma(\{\texttt{?country} \mapsto \mathsf{Germany}\}) = (y_1, x_1) \otimes (y_1, x_2) \otimes ((y_1, x_3) \oplus (y_2, x_3)),$$
$$(\!|Q_O|\!)_\Gamma(\{\texttt{?country} \mapsto \mathsf{Hungary}\}) = (y_1, x_4) \otimes (y_1, x_5) \otimes (y_2, x_6).$$

As described in Section 4, the how provenance of these solution mappings can be computed with NPCS by rewriting the query Q_O into a query Q_R, and then executing query Q_R on the graph Γ.

▶ Note 40. Example 39 shows that, by merging all graphs, we can reuse NPCS to compute how-provenance over a federation of SPARQL endpoints. However, this merging of graphs precludes the direct use of the federation and motivates the introduction of Fed-NPCS, an extension of NPCS to compute how-provenance on federated SPARQL services. To define Fed-NPCS we extend the spm-semiring structure so as to include the pairs (y_i, x_j) that appear in the example. Concretely, these pairs will be encoded with an operation $y_i \circledast x_j$ that tell us that the computation of an answer uses a statement identified as x_j and retrieved from a service y_i.

▶ **Definition 41** (Federated spm-semiring). *Let $\mathcal{M} = (M, \oplus, \circledast, 0_\mathcal{M}, 1_\mathcal{M})$ be a structure where $(M, \oplus, 0_\mathcal{M})$ is a commutative zero-sum-free monoid, $(M, \circledast, 1_\mathcal{M})$ is a zero-sum-free monoid, and $\mathcal{K} = (K, +_\mathcal{K}, \times_\mathcal{K}, -_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ be an spm-semiring. An spm-semiring $(R, \oplus, \otimes, \ominus, 0, 1)$ is said to be a* federated spm-semiring *over $\mathcal{M}$ and $\mathcal{K}$ if there exists a function $f : M \times K \to R$ such that:*

- $f(0_{\mathcal{M}}, k) = 0$ *and* $(m, 0_{\mathcal{K}}) = 0$ *for every* $k \in K$ *and* $m \in M$.
- $f(1_{\mathcal{M}}, 1_{\mathcal{K}}) = 1$.
- *For every* $m \in M$ *and* $k_1, k_2 \in K$,
  - $f(m, k_1 +_{\mathcal{K}} k_2) = f(m, k_1) \oplus f(m, k_2)$.
  - $f(m, k_1 \times_{\mathcal{K}} k_2) = f(m, k_1) \otimes f(m, k_2)$.
  - $f(m, k_1 -_{\mathcal{K}} k_2) = f(m, k_1) \ominus f(m, k_2)$.
- *For every* $m_1, m_2 \in M$ *and* $k \in K$, $f(m_1 \oplus m_2, k) = f(m_1, k) \oplus f(m_2, k)$
- *For every element* $r \in R$ *there is a finite set* $\{(m_1, k_1), \ldots, (m_n, k_n)\} \subseteq M \times K$ *such that* $r = f(m_1, k_1) \oplus \cdots \oplus f(m_n, k_n)$.

$\mathcal{M}$ *is called the* fed-structure *and* $\mathcal{K}$ *is called the* how-structure *of the federated spm-semiring* R.

▶ **Definition 42** (Federated how-provenance expressions)**.** *Let* Y *and* X *be two disjoint sets of IRIs. We write* $\mathrm{FedProvExp}(X, Y)$ *for the set of expressions, called* federated how-provenance expressions*, defined recursively as follows:*

- ~~$0 \in \mathrm{FedProvExp}(X, Y)$, $0 \in \mathrm{FedProvExp}(X, Y)$, *and* $X \subseteq \mathrm{FedProvExp}(X, Y)$.~~
- $0 \in \mathrm{FedProvExp}(X, Y)$, $1 \in \mathrm{FedProvExp}(X, Y)$, *and* $X \subseteq \mathrm{FedProvExp}(X, Y)$.
- *We call* service-expressions *to the expressions combining elements in set* $Y \cup \{0, 1\}$ *with the operators* $\oplus$ *and* $\circledast$. *If* $a \in \mathrm{FedProvExp}(X, Y)$ *and* s *is a service-expression, then* $s \circledast a \in \mathrm{FedProvExp}(X, Y)$.
- *If* $a, b \in \mathrm{FedProvExp}(X, Y)$ *then* $a \oplus b$, $a \otimes b$, *and* $a \ominus b$ *are in* $\mathrm{FedProvExp}(X, Y)$.

The semantics of federated how-provenance expressions is defined by mapping them to a federated spm-semiring $(R, \oplus, \otimes, \ominus, 0, 1)$ with fed-structure $\mathcal{M}$ and how-structure $\mathcal{K}$. Such a mapping h is defined by two mappings $h_Y : Y \to \mathcal{M}$ and $h_X : Y \to \mathcal{K}$ as follows:

- $h(0) = 0$ and $h(1) = 1$.
- If $x \in X$ then $h(x) = h_X(x)$.
- If $y \in Y$ then $h(y) = h_Y(y)$.
- If s_1 and s_2 are service-expressions, then $h(s_1 \oplus s_2) = h_Y(s_1) \oplus h_Y(s_2)$ and $h(s_1 \circledast s_2) = h_Y(s_1) \circledast h_Y(s_2)$.
- If a and b are federated how-provenance expressions, then $h(a \oplus b) = h_Y(a) \oplus h_Y(b)$, $h(a \otimes b) = h_Y(a) \otimes h_Y(b)$, and $h(a \ominus b) = h_Y(a) \ominus h_Y(b)$.

▶ **Example 43.** If we replace the pairs (y_i, x_j) in Example 39 with expressions of the form $y_i \circledast x_j$, then we can codify the polynomials on that example with the following federated how-provenance expressions:

$$(\!|Q_O|\!)_\Gamma(\{\texttt{?country} \mapsto \mathsf{Germany}\}) = (y_1 \circledast x_1) \otimes (y_1 \circledast x_2) \otimes ((y_1 \circledast x_3) \oplus (y_2 \circledast x_3)),$$
$$(\!|Q_O|\!)_\Gamma(\{\texttt{?country} \mapsto \mathsf{Hungary}\}) = (y_1 \circledast x_4) \otimes (y_1 \circledast x_5) \otimes (y_2 \circledast x_6).$$

Observe that the first expression in Example 43 includes the sub-expression $(y_1 \circledast x_3) \oplus (y_2 \circledast x_3)$. Intuitively, this sub-expression tells us that the mapping can be alternatively computed using the statement x_3 on service y_1 or on service y_2. According to Definition 41, this expression can be abbreviated as $(y_1 \oplus y_2) \circledast x_3$.

5.2 Annotated Answers for Federated SPARQL queries

So far, we presented an algebraic structure for queries in the federated setting. Now, we will show how to extend the annotated SPARQL algebra by Geerts et al. [22] for the SERVICE operator.

Recall that given an spm-semiring $\mathcal{K}$ over a set of elements K, a K-graph Γ is a function that maps every RDF triple t to a value $\Gamma(t) \in K$. Similarly, we define a federated dataset as a collection of multiple K-graphs.

▶ **Definition 44** (Annotated Federated Dataset). *Let $\mathcal{R} = (R, \oplus, \otimes, \ominus, 0, 1)$ be a federated spm-semiring, $\mathcal{K}$ be the how-structure of $\mathcal{R}$, and Y be a finite set of elements of the fed-structure of $\mathcal{R}$ such that Y is disjoint with $\{0, 1\}$. An* annotated federated dataset *is a partial function Δ that maps each element $y \in Y$ to a $\mathcal{K}$-graph. The set Y is called the set of* service names *of the annotated federated dataset.*

▶ **Example 45.** The graphs G_1 and G_2 of Example 39 encode the following annotated X-graphs:

$$\begin{aligned}\Gamma_1 = \{\!\!\{ &(\mathsf{Ulm}, \mathsf{a}, \mathsf{City}) \mapsto x_1,\\ &(\mathsf{Ulm}, \mathsf{country}, \mathsf{Germany}) \mapsto x_2,\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) \mapsto x_3,\\ &(\mathsf{Budapest}, \mathsf{a}, \mathsf{City}) \mapsto x_4,\\ &(\mathsf{Budapest}, \mathsf{country}, \mathsf{Hungary}) \mapsto x_5 \}\!\!\},\\ \Gamma_2 = \{\!\!\{ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) \mapsto x_3,\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}) \mapsto x_6 \}\!\!\}.\end{aligned}$$

The function $\Delta = \{y_1 \mapsto \Gamma_1, y_2 \mapsto \Gamma_2\}$ is an annotated federated dataset.

▶ Remark 46. Recall that there are two ways to query a federated dataset. The firs way, described in Section 3.3, consists of evaluating a local query (i.e., a query without SERVICE clauses) over the union of all graphs in the federated dataset. The second way consists of using SERVICE clauses to indicate that certain parts of the query should be evaluated remotely. Hence, in the second way, queries are not local, but remote or fully remote.

Geerts et al. [22] defined an annotated SPARQL algebra (see Definition. 16), which only considers local queries. This algebra can be extended to the evaluation of queries over annotated federated datasets by indicating that one of the services in the federation is used for the local evaluation via the parameter G of the semantics (see Definition 8). Definition 47 does so by extending Definition 16 for federated datasets and SERVICE clauses and it is inspired by the formalization of the semantics of federated query processing by Buil-Aranda [11]. Hence, it defines an annotated SPARQL algebra which gives a semantics to local, remote and fully remote SPARQL queries over federated datasets.

▶ **Definition 47** (Semantics of the Federated Annotated SPARQL Algebra). *Given an annotated federated dataset Δ, an element $y \in Y$, and a mapping μ, the semantics of FedSPARQL queries is defined recursively as follows:*

- $(\!|(s,p,o)|\!)_{\Delta,y}(\mu) = \Delta(y)(\mu(s,p,o))$,
- $(\!|\text{SELECT } W \text{ WHERE } Q|\!)_{\Delta,y}(\mu) = \bigoplus_{\mu':\mu'|_W=\mu} (\!|Q|\!)_{\Delta,y}(\mu')$,
- $(\!|Q \text{ FILTER } \varphi|\!)_{\Delta,y}(\mu) = (\!|Q|\!)_{\Delta,y}(\mu) \otimes 1_{\mu\models\varphi}$,
- $(\!|Q_1 \text{ UNION } Q_2|\!)_{\Delta,y}(\mu) = (\!|Q_1|\!)_{\Delta,y}(\mu) \oplus (\!|Q_2|\!)_{\Delta,y}(\mu)$,
- $(\!|Q_1 \text{ AND } Q_2|\!)_{\Delta,y}(\mu) = \bigoplus_{\mu=\mu_1\cup\mu_2} ((\!|Q_1|\!)_{\Delta,y}(\mu_1) \otimes (\!|Q_2|\!)_{\Delta,y}(\mu_2))$,
- $(\!|Q_1 \text{ DIFF } Q_2|\!)_{\Delta,y}(\mu) = (\!|Q_1|\!)_{\Delta,y}(\mu) \ominus (\bigoplus_{\mu'\sim\mu} (\!|Q_2|\!)_{\Delta,y}(\mu'))$,

- $(\!|Q_1 \text{ OPTIONAL } Q_2|\!)_{\Delta,y}(\mu) = (\!|Q_1 \text{ AND } Q_2|\!)_{\Delta,y}(\mu) \oplus (\!|Q_1 \text{ DIFF } Q_2|\!)_{\Delta,y}(\mu)$,
- $(\!|(\text{SERVICE } y' \ Q)|\!)_{\Delta,y}(\mu) = y' \circledast (\!|Q|\!)_{\Delta,y'}(\mu)$,

where $\bigoplus$ *denotes sums using the operation* $\oplus$, $\sim$ *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* $?x \in \text{dom}(\mu) \cap \text{dom}(\mu')$.

Definition 47 differs from the definition of the annotated SPARQL algebra in three cases. The first case is the evaluation of a triple pattern over a federated graph $\Gamma = \Delta(y)$. The second case defines the SERVICE operation. The current graph identified by y is replaced by another graph y', and this replacement is annotated in the how-provenance expression with the operation $\circledast$.

▶ **Example 48.** Consider the annotated federated dataset Δ from Example 45, and let Q be the query

$$((?\text{city}, \mathsf{a}, \mathsf{City}) \text{ AND } (\text{SERVICE } y_2 \ (\mathsf{Danube}, \mathsf{crosses}, ?\text{city}))),$$

and μ be the mapping $\{?\text{city} \mapsto \mathsf{Ulm}\}$. Then, the first triple pattern is locally evaluated on the annotated graph $\Delta(y_1) = \Gamma_1$:

$$(\!|(?\text{city}, \mathsf{a}, \mathsf{City})|\!)_{\Delta,y_1}(\mu) = \Delta(y_1)(\mu(?\text{city}, \mathsf{a}, \mathsf{City})) = x_1.$$

The service clause in the second part of the query is remotely evaluated on the annotated graph $\Delta(y_2) = \Gamma_2$:

$$\begin{aligned}(\!|(\text{SERVICE } y_2 \ (\mathsf{Danube}, \mathsf{crosses}, ?\text{city}))|\!)_{\Delta,y_1}(\mu) &= y_2 \circledast (\!|(\mathsf{Danube}, \mathsf{crosses}, ?\text{city})|\!)_{\Delta,y_2}(\mu) \\ &= y_2 \circledast \Delta(y_2)(\mu(\mathsf{Danube}, \mathsf{crosses}, ?\text{city})) \\ &= y_2 \circledast x_3.\end{aligned}$$

Then, $(\!|Q|\!)_{\Delta,y_1}(\mu) = x_1 \otimes y_2 \circledast x_3$.

5.3 How-Provenance for Federated Query Evaluation

In the preliminaries (Section 3.3), we described the evaluation of a local SPARQL query Q over multiple RDF graphs $G_1, G_2, \ldots, G_n$, as the evaluation of the query over the union of these graphs. That is, the result is the set of mappings

$$[\![Q]\!]_{\bigcup_{i=1}^{n} G_i} .$$

We called this process the *federated query evaluation* of query Q. However, this definition does not consider the how-provenance of the computed solutions. In Section 5.2, we introduced the federated annotated SPARQL algebra to provide a theoretical framework for how-provenance on federated SPARQL endpoints using the SERVICE clause. In this section, we will connect the notion of federated query evaluation with the federated annotated SPARQL algebra by presenting a simple query rewriting from local queries to remote queries. Doing so, we will formalize the notion of federated how-provenance for the federated query evaluation.

▶ **Definition 49** (Federated Graph). *Let* Δ *be an annotated federated dataset with set of service names* Y. *The* federated graph *of* Δ *is the annotated graph, denoted* $\text{FedGraph}(\Delta)$, *such that for every triple* t, $\text{FedGraph}(\Delta)(t) = \bigoplus_{y \in Y} y \circledast \Gamma(t)$.

▶ **Example 50.** Intuitively, the annotated graph of an annotated federated dataset combines the annotations of the triples using the operation $\oplus$ to indicate the alternative explanations of each triple. The annotated graph for the annotated federated dataset Δ in Example 45 is the following:

$$\begin{aligned}\text{FedGraph}(\Delta) = \{&(\mathsf{Ulm}, \mathsf{a}, \mathsf{City}) \mapsto y_1 \circledast x_1,\\ &(\mathsf{Ulm}, \mathsf{country}, \mathsf{Germany}) \mapsto y_1 \circledast x_2,\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Ulm}) \mapsto (y_1 \circledast x_3) \oplus (y_2 \circledast x_3),\\ &(\mathsf{Budapest}, \mathsf{a}, \mathsf{City}) \mapsto y_1 \circledast x_4,\\ &(\mathsf{Budapest}, \mathsf{country}, \mathsf{Hungary}) \mapsto y_1 \circledast x_5,\\ &(\mathsf{Danube}, \mathsf{crosses}, \mathsf{Budapest}) \mapsto y_2 \circledast x_6\}.\end{aligned}$$

▶ **Definition 51** (Federated Query)**.** *Let Δ be an annotated federated dataset with set of service names $Y = \{y_1, \ldots, y_n\}$, and Q be a local query. The* federated query *of Q over Δ, denoted* $\text{FedQuery}_\Delta(Q)$*, is the query that results from replacing in Q every triple pattern t with the query*

$$((\text{SERVICE } y_1\ t) \text{ UNION } \cdots \text{ UNION } (\text{SERVICE } y_n\ t)).$$

Intuitively, the federated query combines the evaluation of each triple pattern in the different SPARQL endpoints of an annotated federated dataset. The following lemma connects the evaluation of queries on a federated graph with the evaluation of queries in an annotated federated dataset.

▶ **Lemma 52.** *Let Δ be an annotated federated dataset with set of service names Y, and Q be a local SPARQL query. Then, for every mapping μ, and service name $y \in Y$.*

$$(\!|Q|\!)_{\text{FedGraph}(\Delta)}(\mu) = (\!|\text{FedQuery}_\Delta(Q)|\!)_{\Delta,y}(\mu).$$

Proof. It follows directly from definitions 47 and 51. ◀

▶ **Example 53.** Consider the annotated federated dataset $\Delta = \{y_1 \mapsto \Gamma_1, y_2 \mapsto \Gamma_2\}$, the local query

$$Q = ((\texttt{?city}, \mathsf{a}, \mathsf{City}) \text{ AND } (\mathsf{Danube}, \mathsf{crosses}, \texttt{?city})),$$

and the mapping $\mu\{\texttt{?city} \mapsto \mathsf{Ulm}\}$. Then, the polynomial annotation of Q in the federated graph of Δ (see Example 50) is

$$(\!|Q|\!)_{\text{FedGraph}(\Delta)}(\mu) = (y_1 \circledast x_1) \otimes ((y_1 \circledast x_3) \oplus (y_2 \circledast x_3)).$$

Alternatively, the federated query for Q is:

$$\begin{aligned}\text{FedQuery}_Q(\Delta) = (((&\text{SERVICE } y_1\ (\texttt{?city}, \mathsf{a}, \mathsf{City})) \text{ UNION}\\ &(\text{SERVICE } y_2\ (\texttt{?city}, \mathsf{a}, \mathsf{City}))) \text{ AND}\\ &((\text{SERVICE } y_1\ (\mathsf{Danube}, \mathsf{crosses}, \texttt{?city})) \text{ UNION}\\ &(\text{SERVICE } y_2\ (\mathsf{Danube}, \mathsf{crosses}, \texttt{?city})))).\end{aligned}$$

Then, evaluating query $\text{FedQuery}_Q(\Delta)$ on dataset Δ we obtain:

$$(\!|\text{FedQuery}_Q(\Delta)|\!)_\Delta(\mu) = (y_1 \circledast x_1) \otimes ((y_1 \circledast x_3) \oplus (y_2 \circledast x_3)).$$

▶ Note 54. The main implication of Lemma 52 is that we can evaluate the provenance of the evaluation of a local query over an annotated federated dataset without the need of transforming the annotated federated dataset into an annotated graph, but by rewriting the local query. Also, the equivalence of the evaluation of the rewritten query with the evaluation of the local query tells us that the federated evaluation follows the framework of spm-semirings in the non-federated setting.

5.4 Federated Provenance Computation

We have all the fundamentals to extend NPCS for federated provenance computation. By using a query rewriting, NPCS employs a middleware that takes a SPARQL query and generates a new query designed to retrieve provenance information. The resulting query seamlessly executes on a singular instance of a SPARQL endpoint, yielding the desired results along with their how-provenance algebraic expressions. Expanding upon this foundation, our extended approach, *Fed-NPCS*, can be integrated into federation engines. Federation engines, such as FedX [43] and FedUP [3], rewrite an *input query* into a set of subqueries that are executed in different endpoints of the federation. To this end, these federated engines execute queries in two phases. First, in a planning phase, they execute queries or use summaries to elaborate a plan to retrieve data from the different endpoints. This plan includes the definition of subqueries, the endpoints where these subqueries will be evaluated, and the operations to combine the retrieved answers. Second, the engines execute the plans. Our approach, Fed-NPCS, translates the plan generated by these federated engines into a single SPARQL query, called the *federated plan query*. The federated plan query includes SERVICE clauses that encode the execution of the subqueries in the corresponding endpoints, and SPARQL operations to encode the combination of the answers from the subqueries.

The process described so far does not include the federated how-provenance computation but the generation of the federated plan query that can compute the answers to the original query on the federation. To compute the how-provenance data, Fed-NPCS rewrites the federated plan query into a query called the *federated provenance query*. Like in NPCS, Fed-NPCS's federated provenance query is designed to retrieve provenance information.

▶ Remark 55. Unlike the NPCS system, which considers two queries, the Fed-NPCS considers three queries: (1) the input query (Q_O) is the original query sent to NPCS; (2) the federated plan query (Q_F) is the plan generated by the federation engine (e.g., FedX or FedUP) to compute the answers of the input query; and (3) the federated provenance query (Q_R) is the query designed to retrieve provenance information on top of the federated query plan.

▶ Remark 56. Lemma 52 showed that the local query Q_O can be evaluated in a federation of annotated graphs Δ by rewriting to the query $\text{FedQuery}_\Delta(Q_O)$. This query $\text{FedQuery}_\Delta(Q_O)$ is equivalent to the query Q_F generated by a federation engine. The difference is that $\text{FedQuery}_\Delta(Q_O)$ is a simple query that satisfies Lemma 52, and query Q_F is optimized for a fast execution. Heling and Acosta [29] showed that naive decompositions as the one presented in this work are sound and complete for exclusive groups. For arbitrary decompositions, such guarantees are not proven.

5.4.1 Federated Datasets

The aforementioned annotated federated dataset Δ provides a theoretical framework for the notion of how-provenance in the federated setting. In practice, federated engines do not operate over an annotated federated dataset Δ consisting of annotated graphs Γ, but over regular federated datasets D consisting of regular graphs G. Thus, to compute the how-provenance, the input is a regular dataset D which encodes Δ. Like NPCS, Fed-NPCS requires reification to encode each annotated graph Γ with a regular graph G. In this subsection we describe such an encoding.

▶ **Definition 57** (Encoding of the annotated federated dataset)**.** *Let X and Y be two disjoint sets of IRIs, Δ be an annotated federated dataset with set of service names Y, where for each $y \in Y$, $\Delta(y)$ is an X-graph. Given a reification scheme* Reify*, the* encoding of Δ *with the reification scheme* Reify *is the federated dataset D that maps each IRI $y \in Y$ to the RDF-star graph G defined as follows:*

$$G = \{\mathsf{Reify}(t, \Gamma(t)) \mid t \in \text{supp}(\Gamma)\},$$

where Γ *is the* X*-graph* $\Delta(y)$.

▶ **Example 58.** Consider the reification scheme $\mathsf{Reify}(t, x) = (t, \mathsf{wasDerivedFrom}, x)$, and the federated dataset Δ described in Example 45. Then, the encoding of Δ with the reification scheme Reify is the RDF-star dataset $D = \{y_1 \mapsto G_1, y_2 \mapsto G_2\}$, where G_1 and G_2 are the RDF-star graphs described in Example 39.

5.4.2 Query rewriting design

Recall (Section 4) that NPCS computes how-provenance of a query Q_O over an RDF graph G that uses reification and that results in an X-graph Γ. The provenance polynomial expression is bound to an additional SPARQL variable `?prov` for each mapping μ in the support of $(\!|Q_O|\!)_\Gamma$. To this end, NPCS translates the local SPARQL query Q_O into another local SPARQL query Q_R whose answers are extended with provenance annotations. Formally, the answers in the set $[\![Q_R]\!]_G$ have the form $\mu \cup \{\texttt{?prov} \mapsto k\}$, where k is the how-provenance polynomial of μ (i.e., $(\!|Q_O|\!)_\Gamma(\mu) = k$). Similarly, our goal is to define a method, called Fed-NPCS, that rewrites every local SPARQL query Q_O into a fully remote SPARQL query Q_R such that the answers of Q_R over a federated dateset D have the form $\mu \cup \{\texttt{?prov} \mapsto k\}$ and $(\!|Q_O|\!)_\Delta(\mu) = k$, where Δ is the annotated federated dataset encoded by the federated dataset D.

Fed-NPCS uses two steps to translate the original query Q_O into the query Q_R that computes the provenance:

1. In the first step, Fed-NPCS translates the original query Q_O into an intermediate query Q_F which returns the expected answers of query Q_O over a federated dataset. For example, this intermediate query can be $Q_F = \mathrm{FedQuery}_\Delta(Q_O)$ (see Definition 51). Indeed, by Lemma 52, query Q_F returns the same answer as the federated evaluation of query Q_O. The intermediary query Q_F used in this work uses a naive decomposition that is proven to be sound and complete for exclusive groups [29].
2. In the second step, Fed-NPCS translates Q_F into the desired query Q_R that computes the provenance. This step is equivalent to what NPCS does on non-federated queries.

In the remainder of this section we will describe these two steps.

Step 1: The federated engine plan

$\mathrm{FedQuery}_\Delta(Q_O)$ is a simple alternative to the definition of the intermediate query Q_F. However, this query does not take advantage of the distribution of the data across the endpoints in the federation. Federated query engines, such as FedX [42] and FedUP [3], define more efficient query plans for the federated query evaluation, which can be encoded into intermediate queries Q_F to improve the efficiency of Fed-NPCS.

These query plans reflect strategic decisions informed by the structure and content of the underlying federation. In practice, federation engines must determine which endpoints are likely to return useful results for each triple pattern or basic graph pattern. To do so, they rely on techniques such as metadata inspection, runtime source probing, or precomputed summaries to avoid contacting irrelevant federation members. This process, often referred to as *source selection*, plays a critical role in scaling to federations with a large number of endpoints.

After identifying relevant sources, federation engines apply *query decomposition* to assign subqueries to endpoints and construct an execution plan that minimizes intermediate result sizes and total query latency. Fed-NPCS translates these assignments subqueries-endpoint to SERVICE clauses.

▶ **Example 59.** Let $Q_O = (T_1 \text{ AND } T_2 \text{ AND } T_3)$ be a SPARQL query composed of the three triple patterns T_1, T_2, and T_3. Then, the simplest way to define a intermediate query Q_F is using the federated query $\text{FedQuery}_\Delta(Q_O)$. If Δ includes three service names, y_1, y_2, and y_3, then Q_F is:

$$\begin{aligned} Q_F = (&((\text{SERVICE } y_1\ T_1) \text{ UNION } (\text{SERVICE } y_2\ T_1) \text{ UNION } (\text{SERVICE } y_3\ T_1)) \text{ AND} \\ &((\text{SERVICE } y_1\ T_2) \text{ UNION } (\text{SERVICE } y_2\ T_2) \text{ UNION } (\text{SERVICE } y_3\ T_2)) \text{ AND} \\ &((\text{SERVICE } y_1\ T_3) \text{ UNION } (\text{SERVICE } y_2\ T_3) \text{ UNION } (\text{SERVICE } y_3\ T_3))). \end{aligned}$$

If we are informed that service y_3 has no answers for triple patterns T_1 and T_2, and that services y_1 and y_2 have no answers for triple pattern T_3, then we can rewrite Q_F as follows:

$$\begin{aligned} Q'_F = (&((\text{SERVICE } y_1\ T_1) \text{ UNION } (\text{SERVICE } y_2\ T_1)) \text{ AND} \\ &((\text{SERVICE } y_1\ T_2) \text{ UNION } (\text{SERVICE } y_2\ T_2)) \text{ AND} \\ &(\text{SERVICE } y_3\ T_3)). \end{aligned}$$

If we additionally know that each answer μ_1 of triple pattern T_1 in y_1 is incompatible with all the answers of triple pattern T_2 in service y_2, and similarly, each answer μ_2 to triple pattern T_1 in y_2 is incompatible with all the answers to triple pattern T_2 in service y_1, then we can rewrite Q_F as follows:

$$\begin{aligned} Q''_F = (&((((\text{SERVICE } y_1\ (T_1 \text{ AND } T_2))) \text{ UNION} \\ &((\text{SERVICE } y_2\ (T_1 \text{ AND } T_2)))) \text{ AND} \\ &(\text{SERVICE } y_3\ T_3)). \end{aligned}$$

Queries Q'_F and Q''_F require less time to be executed than query Q_F because they avoid unnecessary pattern evaluation, and reduce network data transfer because of evaluating the joins in the service for which the basic graph patterns are exclusive.

▶ Remark 60. The queries Q_F, Q'_F, and Q''_F in Example 59 are not equivalent in general, but under the information described in the example. Federated engines use availability and pattern compatibility information across endpoints to generate efficient query plans. These efficient plans discard services that are known to do not return answers to a triple pattern or pairs of services that are known to do not produce joinable mappings for a given AND operation.

Step 2: Query Rewriting

After generating an intermediate query Q_F, which includes SERVICE operators, the next step is to rewrite Q_F into a query Q_R that computes the provenance. The difference between a local and a federated query provenance computing lies in the use of the SERVICE operator, which directs portions of the query to be evaluated on external SPARQL endpoints. Our rewriting must ensure that each part of the federated query can be rewritten to preserve both the provenance of intermediate results and the aggregation of these results across sources. To define this query rewriting we first define an auxiliary function that annotates the data from remote services with the service name.

▶ **Definition 61.** *Given a SPARQL variable* `?prov` *and an IRI* y*, the expression* $\text{ServiceOp}(y, \texttt{?prov})$ *is defined as follows:*

$$\text{ServiceOp}(y, \texttt{?prov}) = \texttt{concat("(} \circledast \texttt{ ", } y\texttt{, ?prov, ")")}.$$

Like the functions in Definition 24, the expression in Definition 61 defines a function to combine polynomial expressions. In particular, the function ServiceOp combines a polynomial with the service name using the service operator.

▶ **Definition 62** (Base Fed-NPCS query rewriting). *Let Q_F be a SPARQL query,* `?prov` *a variable, and* Reify *a reification scheme. Then, the rewritten query for Q_F and variable* `?prov` *over scheme* Reify*, denoted $\beta(Q_F, \texttt{?prov})$, is defined recursively as follows:*

- *If Q_F matches one of the rewriting rules in Definition 28 (i.e., the NPCS query rewriting), then $\beta(Q_F, \texttt{?prov})$ is the result of applying that rule and then recursively applying rewriting function β.*
- *If Q_F has the form* (SERVICE y Q) *then $\beta(Q_F, \texttt{?prov})$ is the query*

$$\begin{array}{ll}(\ \text{SELECT} & \text{inScope}(Q) \cup \{\texttt{?prov}\} \\ \ \text{WHERE} & ((\text{SERVICE } y\ \beta(Q, \texttt{?prov}\circledast))\ \text{BIND}\ (\text{ServiceOp}(y, \texttt{?prov}\circledast)\ \text{AS}\ \texttt{?prov}))).\end{array}$$

▶ **Example 63.** Consider the federated dataset $D = \{y_1 \mapsto G_1, y_2 \mapsto G_2\}$ from Example 45, and the query

$$Q_F = ((\text{SERVICE } y_1\ (\mathsf{Danube}, \mathsf{crosses}, \texttt{?city}))\ \text{UNION}\ (\text{SERVICE } y_2\ (\mathsf{Danube}, \mathsf{crosses}, \texttt{?city}))).$$

By the rule for the UNION operator (in Definition 28), the rewritten query for Q_F is

$$\begin{array}{lll}\beta(Q_F, \texttt{?prov}) = (\ \text{SELECT} & \texttt{?city}\ (\text{ProvAggSum}(\texttt{?prov}\oplus)\ \text{AS}\ \texttt{?prov}) \\ \text{WHERE} & (P_1\ \text{UNION}\ P_2) \\ \text{GROUP BY} & \texttt{?city}\ ),\end{array}$$

where, for $i \in \{1, 2\}$, P_i is the query $\beta((\text{SERVICE } y_i\ T), \texttt{?prov}\oplus)$ and T is the triple pattern $(\mathsf{Danube}, \mathsf{crosses}, \texttt{?city})$. These queries P_i are computed using the rule for the SERVICE operator:

$$\begin{array}{ll}P_i = (\ \text{SELECT} & \texttt{?city}\ \texttt{?prov}\oplus \\ \ \text{WHERE} & ((\text{SERVICE } y_i\ \beta(T, \texttt{?prov}\oplus\circledast))\ \text{BIND}\ (\text{ServiceOp}(y_i, \texttt{?prov}\oplus\circledast)\ \text{AS}\ \texttt{?prov}\oplus))).\end{array}$$

The query $\beta(T, \texttt{?prov}\oplus\circledast)$ is defined by the rule for triple patterns

$$\begin{array}{lll}\beta(T, \texttt{?prov}\oplus\circledast) = (\ \text{SELECT} & \texttt{?city}\ (\text{ProvAggSum}(\texttt{?prov}\oplus\circledast\oplus)\ \text{AS}\ \texttt{?prov}\oplus\circledast) \\ \text{WHERE} & \mathsf{Reify}((\mathsf{Danube}, \mathsf{crosses}, \texttt{?city}), \texttt{?prov}\oplus\circledast\oplus) \\ \text{GROUP BY} & \texttt{?city}\ ).\end{array}$$

Then, the query $\beta(T, \texttt{?prov}\oplus\circledast)$ returns the following answers in each service:

$$[\![\beta(T, \texttt{?prov}\oplus\circledast)]\!]_{D,y_1} = \left[\begin{array}{c|c} \texttt{?city} & \texttt{?prov}\oplus\circledast \\ \hline \mathsf{Ulm} & (\oplus\ (x_3)) \end{array}\right],$$

$$[\![\beta(T, \texttt{?prov}\oplus\circledast)]\!]_{D,y_2} = \left[\begin{array}{c|c} \texttt{?city} & \texttt{?prov}\oplus\circledast \\ \hline \mathsf{Ulm} & (\oplus\ (x_3)) \\ \mathsf{Budapest} & (\oplus\ (x_6)) \end{array}\right].$$

Then, the results of evaluating queries P_1 and P_2 are:

$$[\![\beta(P_1, \texttt{?prov}\oplus)]\!]_D = \left[\begin{array}{c|c} \texttt{?city} & \texttt{?prov}\oplus \\ \hline \mathsf{Ulm} & (\circledast\ y_1\ (\oplus\ (x_3))) \end{array}\right],$$

$$[\![\beta(P_2, \texttt{?prov}\oplus)]\!]_D = \left[\begin{array}{c|c} \texttt{?city} & \texttt{?prov}\oplus \\ \hline \mathsf{Ulm} & (\circledast\ y_2\ (\oplus\ (x_3))) \\ \mathsf{Budapest} & (\circledast\ y_2\ (\oplus\ (x_6))) \end{array}\right].$$

Combining these results we obtain:

$$[\![\beta(Q_F, \texttt{?prov})]\!]_D = \left[\begin{array}{c|l} \texttt{?city} & \texttt{?prov} \\ \hline \mathsf{Ulm} & (\oplus\ (\circledast\ y_1\ (\oplus\ (x_3)))\ (\circledast\ y_2\ (\oplus\ (x_3)))) \\ \mathsf{Budapest} & (\oplus\ (\circledast\ y_2\ (\oplus\ (x_6)))) \end{array}\right].$$

Thus, the values for variable ?prov for solutions where variable ?city takes the values Ulm and Budapest are the Polish notation expressions for the polynomials $(y_1 \circledast x_3) \oplus (y_2 \circledast x_3)$ and $y_2 \circledast x_6$.

Like in NPCS (see Theorem 33), the correctness of the Fed-NPCS's query rewriting is based on the underlying algebra of queries evaluated over annotated datasets.

▶ **Theorem 64.** *Let* Reify *be a reification scheme, and β be the function described in Definition 62. Then, function β is sound and complete for the reification scheme* Reify.

Proof. It can be shown by induction on the structure of the query. Excluding the rule with the SERVICE operator, the proof is the same as for Theorem 33. Let us review the new case and assume a federated dataset D for the corresponding annotated dataset Δ. If a query Q_F has the form (SERVICE y Q) then $\beta(Q_F, \texttt{?prov})$ is the query:

$$\begin{array}{rl} \beta(Q_F, \texttt{?prov}) = (\ \text{SELECT} & \text{inScope}(Q) \cup \{\texttt{?prov}\} \\ \text{WHERE} & ((\text{SERVICE}\ y\ \beta(Q, \texttt{?prov}\circledast))\ \text{BIND}\ (\text{ServiceOp}(y, \texttt{?prov}\circledast)\ \text{AS}\ \texttt{?prov}))). \end{array}$$

First, to show that β is sound, assume that mapping $\mu \cup \{\texttt{?prov} \mapsto y \circledast k\}$ is an answer to query $\beta(Q_F, \texttt{?prov})$. By construction, mapping $\mu \cup \{\texttt{?prov} \mapsto k\}$ is then an answer to query $\beta(Q, \texttt{?prov}\circledast)$. By induction, $(\!|Q|\!)_{\Delta,y}(\mu) = k$. By the semantics of the SERVICE operator (see Definition 47), $(\!|Q_F|\!)_{\Delta,y}(\mu) = y \circledast k$. Hence, β is sound.

Second, to show that β is complete, we use the inverse reasoning. By induction the answers to query $\beta(Q, \texttt{?prov}\circledast)$ include all mappings $\mu \cup \{\texttt{?prov}\circledast \mapsto k\}$ such that $(\!|Q|\!)_{\Delta,y}(\mu) = k$ and $k \neq 0$. Since $y \circledast 0 = 0$, the answers to query $\beta(Q_F, \texttt{?prov})$ include all solutions $\mu \cup \{\texttt{?prov} \mapsto y \circledast k\}$ such that $y \circledast k \neq 0$. ◀

So far, we have presented a sound and complete base query rewriting to compute provenance for queries evaluated on federated SPARQL endpoints. However, this base query rewriting generates some redundant operations that can be further simplified. Fed-NPCS applies the same techniques used by NPCS and described in Section 4.4 for optimization.

▶ **Example 65.** Consider the query Q_F from Example 63. Wrapping up, the rewritten query $\beta(Q_F, \texttt{?prov})$ is

```
( SELECT    ?city (ProvAggSum(?prov⊕) AS ?prov)
  WHERE     (( SELECT  ?city ?prov⊕
                WHERE   ((SERVICE y1
                           ( SELECT     ?city (ProvAggSum(?prov⊕⊛⊕) AS ?prov⊕⊛)
                             WHERE      Reify((Danube, crosses, ?city), ?prov⊕⊛⊕)
                             GROUP BY   ?city ))
                          BIND (ServiceOp(y1, ?prov⊕⊛) AS ?prov⊕)))
              UNION
              ( SELECT  ?city ?prov⊕
                WHERE   ((SERVICE y2
                           ( SELECT     ?city (ProvAggSum(?prov⊕⊛⊕) AS ?prov⊕⊛)
                             WHERE      Reify((Danube, crosses, ?city), ?prov⊕⊛⊕)
                             GROUP BY   ?city ))
                          BIND (ServiceOp(y2, ?prov⊕⊛) AS ?prov⊕))))
  GROUP BY  ?city ).
```

Following the optimization rules, we can rewrite this query as follows:

```
( SELECT    ?city (ProvAggSum(?prov⊕) AS ?prov)
  WHERE     (((SERVICE y1 Reify((Danube, crosses, ?city), ?prov⊕⊛))
               BIND (ServiceOp(y1, ?prov⊕⊛) AS ?prov⊕))
             UNION
             ((SERVICE y2 Reify((Danube, crosses, ?city), ?prov⊕⊛))
               BIND (ServiceOp(y2, ?prov⊕⊛) AS ?prov⊕))
  GROUP BY  ?city ).
```

In the spirit of the simplification rule no. 2 in Definition 35 (for chains of UNION operators), we can remove two SELECT operations and obtained a simpler provenance query.

To summarize, our query rewriting framework, Fed-NPCS, extends NPCS to federated queries by introducing the ⊛ operator to track the provenance of intermediate results obtained from different SPARQL endpoints. The extension preserves the soundness and completeness of NPCS, ensuring correct results even when querying across multiple sources. By leveraging existing base rewriting rules and adapting them to handle federations, our approach provides a robust mechanism for executing and combining federated SPARQL queries with annotated provenance information.

6 Evaluation

Our evaluation is structured in two parts. In the first part, i.e., Section 6.1, we evaluate NPCS, our how-provenance solution for SPARQL queries on centralized knowledge graphs using two state-of-the-art benchmark datasets and two SPARQL engines. Then, Section 6.2 provides an evaluation of Fed-NPCS on the FedShop federation benchmark [16] that provides federations of different sizes.

6.1 Evaluation on Centralized KGs (NPCS)

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

6.1.1 Experimental Setup

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[3] (version 10.2.0) and Stardog[4] (version 9.1.0). For all our experiments, we set a timeout of 350 seconds 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.

Competitor. We compare NPCS with SPARQLprov [30], a state-of-the-art solution for how-provenance in SPARQL, which is also based on query rewriting. We used the implementation provided with the paper [30], and extended it to support the RDF-star reification scheme.

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

SPARQLprov and NPCS compute the same provenance polynomials since they both rely on spm-semirings.

Synthetic Workload. We employed the Watdiv [4] 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 [30]. These non-monotonic queries were created by enclosing one of the triple patterns in the linear queries with an OPTIONAL clause. The triple pattern to be enclosed was chosen randomly to ensure a diverse set of non-monotonic query templates.

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 [30]. 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.

Real Workload. We tested NPCS and SPARQLprov on the WDBench benchmark [5], which provides real-world data. The benchmark uses 15.2 billion triples encoded using the Wikidata reification scheme from 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 one triple pattern. The queries were randomly chosen.

6.1.2 Results

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 regarding the execution of the original query. 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 highlights that runtime across query templates exhibits higher variability in Stardog compared to GraphDB. Regardless of the data size and the query engine, templates C3 and F5 are by far the most challenging, and make 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

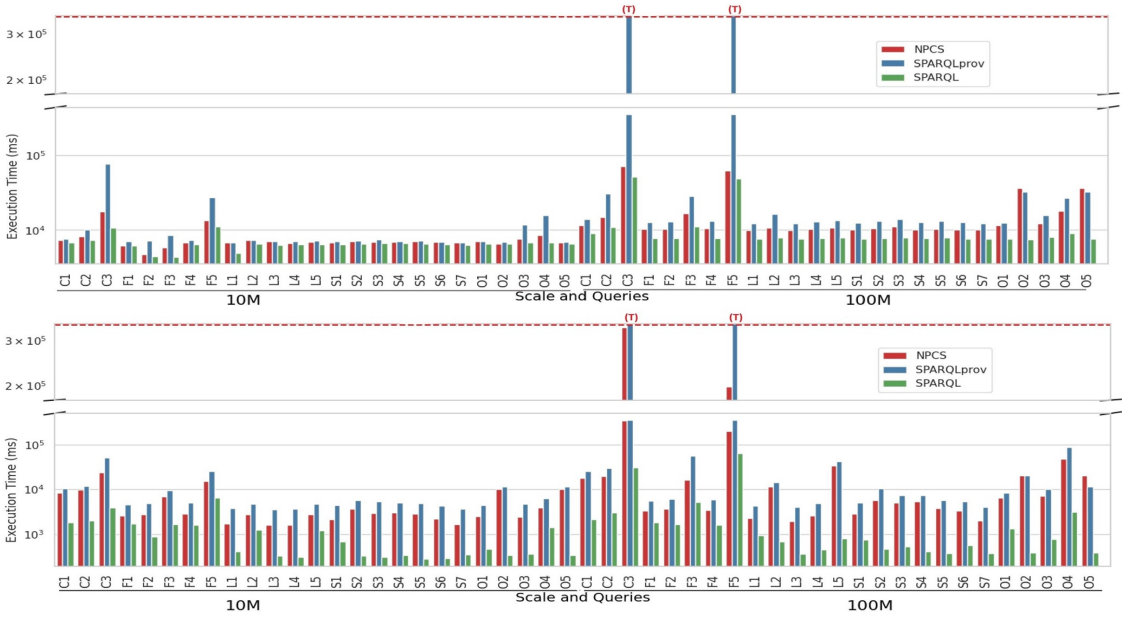

Figure 2 Query execution times on Watdiv 10M and 100M reified with RDF-star on GraphDB (top) and Stardog (bottom). The "SPARQL" label indicates the performance of the query engine without any provenance support.

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 SPARQL 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.

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.

Real Workload. We evaluate NPCS on 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

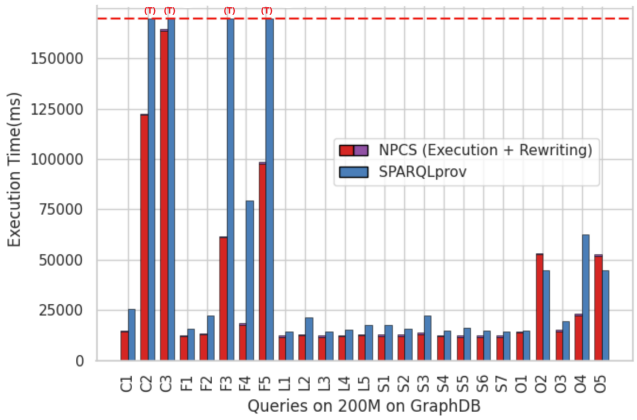

Figure 3 Query execution times on Watdiv 200M reified with RDF-star on GraphDB

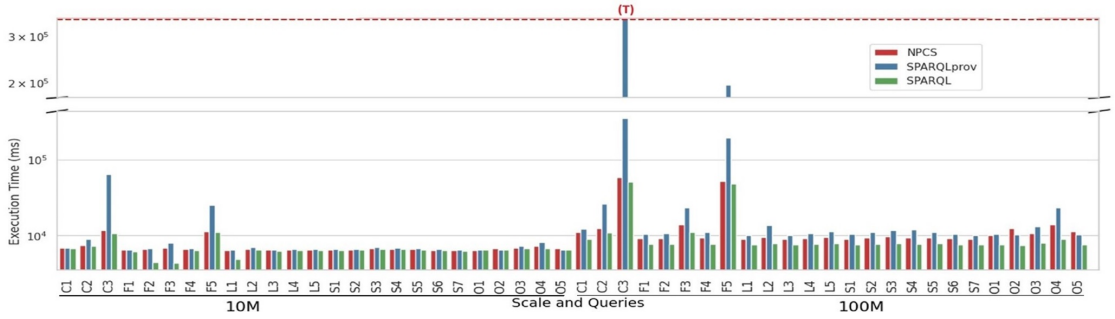

Figure 4 Query execution times on Watdiv 10M and 100M reified as named graphs on GraphDB

of results is above 700K on Stardog (on GraphDB timeouts start after 1M solutions). We also observe that for queries with a few thousand results executed on Stardog, NPCS's overhead can be minimal.

6.2 Evaluation on a Federated Setting (Fed-NPCS)

6.2.1 Experimental Setup

Environment. Fed-NPCS was implemented in Java, using the Java Development Kit (JDK) version 11. All the experiments were conducted on the same hardware environment used in the centralized evaluation: 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. In this evaluation we tested Fed-NPCS on GraphDB[5] (version 10.2.0) as it exhibits better performance than Stardog in Figure 5. To simulate a federated environment, we deployed multiple endpoints hosting different RDF datasets in a distributed manner. Each dataset was hosted on separate instances of the engine, configured to act as individual SPARQL services accessible via the `SERVICE` keyword. The evaluation considered varying the number of distributed endpoints, ranging from 20 to 200, to assess the scalability and efficiency of the rewritten federated queries. As with the centralized evaluation, a timeout of 350 seconds was enforced to individual query executions. For consistency, we report the average response time over five executions in a cold setting (with the disk cache cleared).

[5] `https://graphdb.ontotext.com/`

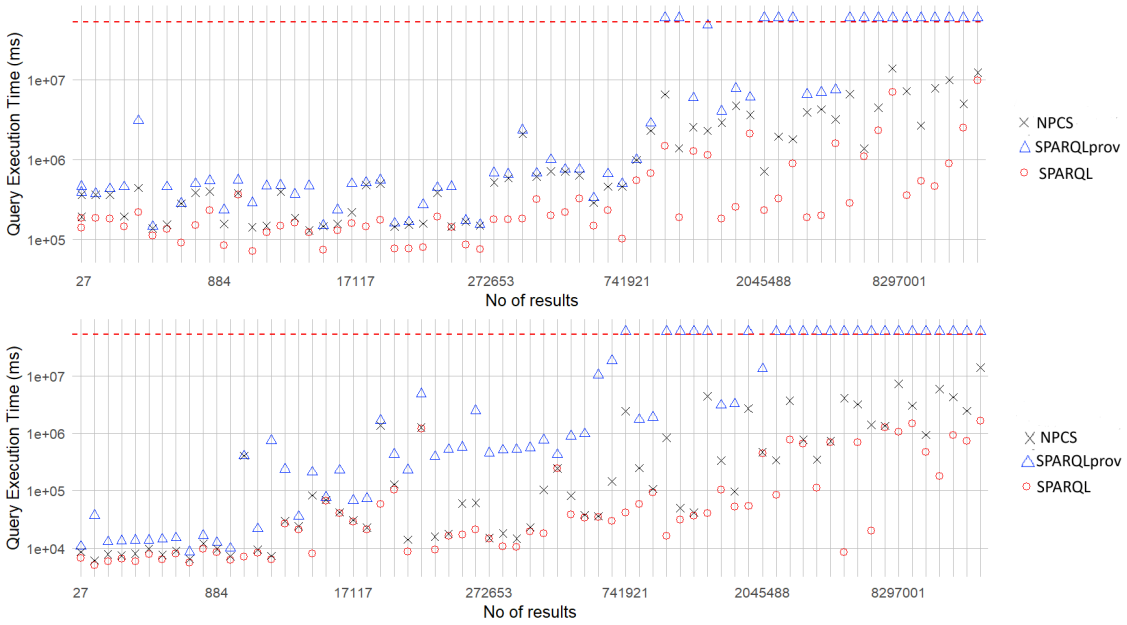

Figure 5 Number of results vs. query execution time for the WDBench queries run on Wikidata, stored in GraphDB (top) and Stardog (bottom), using the Wikidata reification scheme

Query Workload. We used the FedShop benchmark [16], which is specifically designed to evaluate the scalability of federated RDF/SPARQL query engines. The FedShop schema replicates a virtual catalog across autonomous vendors and rating sites, linking local and global entities using `owl:sameAs` statements, and generates federations of varying sizes using schema-based data generators while maintaining local independence and global interoperability. FedShop provides pre-configured federations and a comprehensive query workload, which we employed for our evaluation. We generated 10 federations $F(N)$ with sizes $N = \{20, 40, \ldots, 200\}$. Each federation has as many vendors as reviewing sites. For instance, $F(200)$ consists of 100 vendors and 100 review sites. All necessary instructions to install, configure, and run the benchmark are available on the FedShop GitHub repository[6].

The FedShop's query generator was used to create a workload of 120 queries, each derived from 12 template queries and executed across our 10 federations – from $F(20)$ to $F(200)$ – providing a robust testing environment. ~~FedShop handles query generation, and relies on FedX~~ [43] ~~for sub-query decomposition (using a unions-over-joins rewriting strategy) and the necessary source selection over federated datasets.~~ FedShop handles query generation, query decomposition, and source selection over federated datasets, providing pre-computed Reference Source Assignments (RSA) that we used as the basis for our evaluation.

The FedShop benchmark is designed to assess the performance and scalability of query federation engines across multiple sources. It categorizes queries based on their data location patterns, namely into, single-domain, multi-domain, and cross-domain queries. These are based on the queries' join variables and on how they combine data from the different sources, and can be explained by the structure of the provenance explanations of their solutions. Single-domain queries are queries without global join variables and with a bounded subject in at least one triple pattern. Since that bounded subject appears only in one of the endpoints, this allows the query

[6] `https://github.com/GDD-Nantes/FedShop.git`

to be executed on a single endpoint using one SERVICE clause, as seen for query templates Q9, Q11, and Q12. The provenance of their solutions consists of polynomials with a single $\circledast$ operator, e.g., $e_1 \circledast (s_1 \otimes s_2)$ with endpoint e_1. Multi-domain queries also lack global join variables but do not include bounded subjects, requiring all triple patterns to be grouped into one SERVICE clause, with results potentially retrieved from multiple endpoints. Unlike single-domain queries the resulting provenance explanations include summations of polynomials where each term in the sum is labeled with a different source using the $\circledast$ operator, e.g., $e_1 \circledast (s_1 \otimes s_2) \oplus e_2 \circledast (s_3 \otimes s_4)$. This is the case for query templates Q1, Q2, Q3, Q4, Q6, Q8, and Q10. Cross-domain decomposition involves queries with global join variables linking data across multiple datasets, which requires multiple SERVICE clauses that combine results from several endpoints, as in query templates Q5 and Q7. This translates into how-provenance explanations that contain products labeled with different sources, e.g., $e_1 \circledast (s_1 \otimes s_2) \otimes e_2 \circledast (s_3 \otimes s_4)$.

For our evaluation, we relied entirely on FedShop's query generation, decomposition, and execution framework, enabling a straightforward comparison across varying federation sizes. The benchmark efficiently manages the minimal source selection over federated datasets and provides valuable insights into the performance of our approach. This corresponds to Step 1 in Section 5.4.2; we therefore evaluate the overhead of rewriting the federated plan to augment its results with how-provenance (Step 2).

6.2.2 Results

We evaluated the performance of Fed-NPCS using the FedShop benchmark on a subset of 12 queries. Out of these, four were optional queries (Q2, Q3, Q7, and Q8). Each query consists of nine subqueries, and each subquery was executed 10 times, corresponding to the number of endpoints (from 20 to 200) involved in the execution. Since there is no direct competitor to Fed-NPCS, the only meaningful comparison is with a baseline federated SPARQL plan. This comparison allows us to assess the performance overhead introduced by provenance in a federated setting and to evaluate the scalability of Fed-NPCS with an increasing number of endpoints.

Fed-NPCS demonstrated higher execution times compared to SPARQL in most cases, which can be attributed to the additional overhead of query rewriting.~~As shown in Figure~~ 7~~, this overhead consists of two components: (i) query rewriting time, which is minimal (typically $<$ 1 second), and (ii) query execution time on the reified knowledge graph, which dominates the total runtime.~~ As shown in Figure 8, this overhead consists of two components: (i) query rewriting time, which is minimal across all evaluated configurations (typically <1% of total runtime at larger scales), and (ii) query execution time on the reified knowledge graph, which dominates the total runtime. To avoid visual distortion that arises from stacked bar charts on a logarithmic scale, Figure 8 presents these two components as percentage contributions on a linear scale, where rewriting time represents at most 15% of total runtime (WatDiv–GraphDB named graph reification at 10M triples) and decreases substantially as data scale increases. However, as the number of endpoints increased, Fed-NPCS showed a more stable performance pattern, particularly at handling a larger number of results. Below, we provide the performance outcomes for selected queries.

Impact of the number of endpoints. Figure 6 illustrates the execution time in milliseconds for both the SPARQL and Fed-NPCS as the number of endpoints varies from 20 to 200. Each data point represents the average execution time for 12 queries across five runs[7].

The results show a clear trend: as the number of endpoints increases, both SPARQL and

[7] In reality we run each query five times and dropped the first measurement as it reports the response time in a cold setting.

Fed-NPCS experience a gradual increase in execution time. Not surprisingly Fed-NPCS takes longer to execute than SPARQL across all federation sizes due to the additional overhead associated with provenance computation, which requires extra resources for tracking and processing query provenance data.

Despite this overhead, Fed-NPCS exhibits a relatively stable and predictable growth pattern in execution time. Both systems demonstrate near-linear scalability as the endpoint count rises. Notably, the gap between SPARQL and Fed-NPCS widens slightly at higher endpoint counts as query processing on larger federations incurs the exchange of more query results – both intermediate and final. This overhead reaches 10.3% for the 200-source federation.

In short, while Fed-NPCS incurs higher execution costs than SPARQL due to provenance computation, its consistent scalability suggests it can still be viable for large federations where provenance tracking is required. This comparison highlights the trade-off between execution speed and the added value of provenance tracking in distributed query processing scenarios.

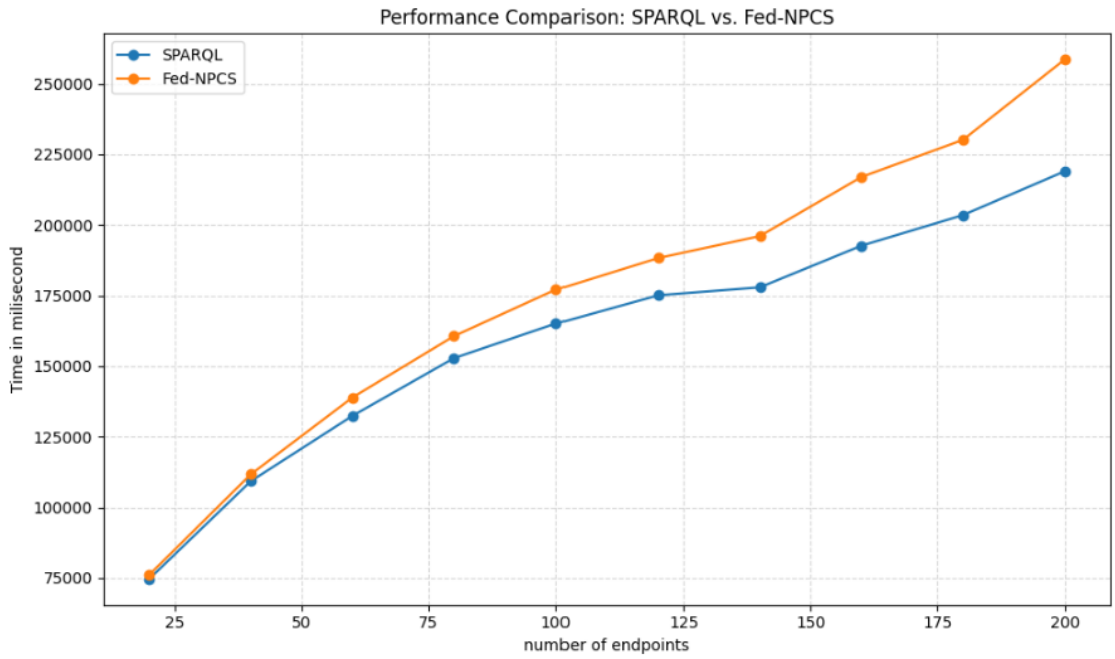

Figure 6 Average execution time of the different FedShop query templates (y-axis) vs. the size of the federation (x-axis).

Individual query analysis. Figure 7 presents the execution time comparison between the baseline SPARQL engine and Fed-NPCS across 12 queries on 200 endpoints. The y-axis is in logarithmic scale and is also split to accommodate for the wide range of execution times.

Notably, for complex queries (such as Q5 and Q6), Fed-NPCS exhibits substantially higher execution times compared to plain SPARQL. However, for less intensive queries (like Q8-Q12 that are not cross-domain), the difference in execution times is less pronounced.

The performance of individual queries reflects varying levels of complexity and data distribution. Notably, Q6 shows the highest execution time among all queries, as it is a multi-domain query involving the largest number of results. The high volume of intermediate results significantly contributes to the execution time for both SPARQL and Fed-NPCS, though the latter incurs additional overhead due to provenance computation.

The cross-domain queries, such as Q5 and Q7, also exhibit longer execution times as they

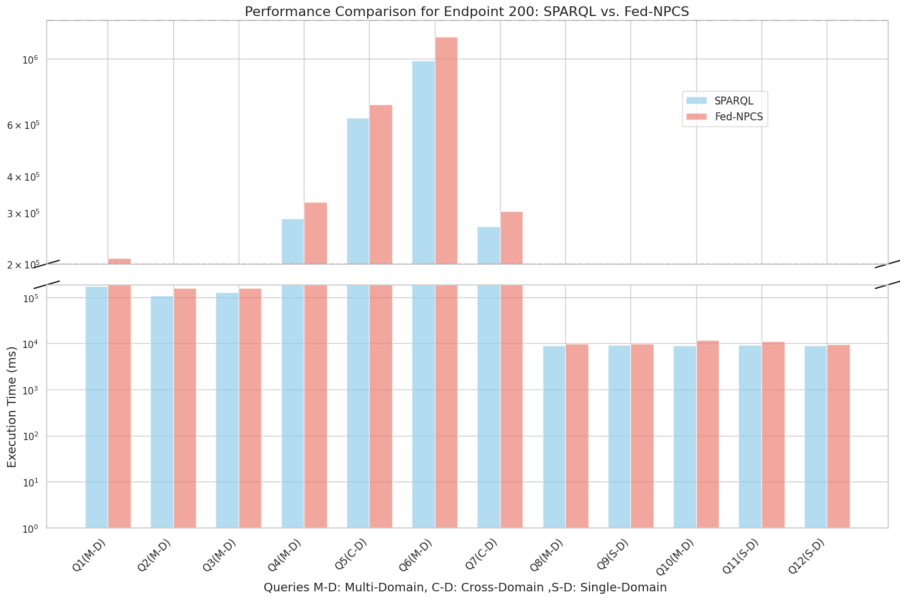

Figure 7 Average query template runtime for the original SPARQL queries vs. the Fed-NPCS queries on a FedShop federation with 200 sources.

involve extensive data joining across federated endpoints. The performance of the multi-domain queries Q1-Q4 and Q8, is mainly explained by their respective numbers of intermediate and final results. For example, queries with more intermediate results tend to require more computation, leading to higher execution times.

Conversely, the single-domain queries, such as Q10, Q11, and Q12, are simpler and thus exhibit the shortest execution times. Both SPARQL and Fed-NPCS perform comparably on these queries, with minimal overhead observed in Fed-NPCS. This indicates that, while the Fed-NPCS introduces additional computation, its impact is less pronounced for simpler queries that do not involve complex joins or provenance tracking across domains.

Overall, while the Fed-NPCS consistently incurs additional overhead due to provenance computation, the resulting system scales to complex multi-domain and cross-domain queries. This makes it a suitable choice for large-scale federated environments where provenance tracking is a priority.

7 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. NPCS can be easily applied to standard and already deployed RDF/SPARQL engines, without the need for customized extensions or post-processing steps. Moreover, we introduce NPCS's federated counterpart, Fed-NPCS, which ports this idea to federations of SPARQL endpoints; this is also a novel extension, as, to our knowledge, there is currently no other engine allowing the computation of provenance information over federations.

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

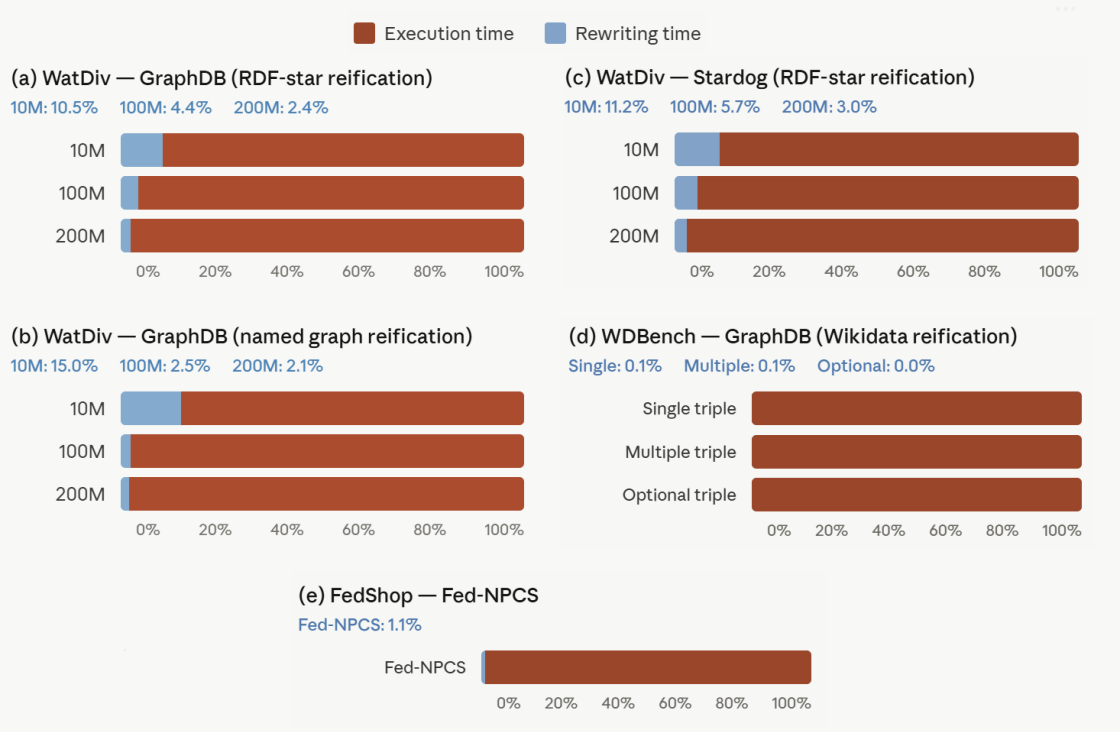

Figure 8 Breakdown of total query runtime into rewriting time (blue) and execution time (red), expressed as percentages. Rewriting time is consistently negligible, particularly at larger data scales.

on large volumes of data—a common scenario for multi-source KG construction and OLAP for KGs [18, 31–33, 35]. But NPCS's performance also makes it feasible to compute how-provenance explanations in federated settings. As shown by our evaluation of Fed-NPCS on the FedShop benchmark, our approach scales to federations with up to 200 sources. We expect this work to open new avenues in the fields of federated query optimization, access control, and data quality.

In this paper the result of a SPARQL query has a column with character strings which represent expressions in ProvExp(X) in Polish notation. For example, a mapping μ can be annotated with an expression $(\oplus\ u_1\ (\otimes\ u_2\ u_3))$ where u_1, u_2, and u_3 are URLs identifying statements. To apply an homomorphism to a spm-semiring $\mathcal{K}$ we can replace these URLs with the respective elements in $\mathcal{K}$ and then apply the corresponding operations on $\mathcal{K}$. This procedure is less efficient than using buit-in operations during the query evaluation. For example, for the natural numbers spm-semiring, we can redefine the operations ProvAggSum, ProvProd, and ProvDiff as follows:

$$\begin{aligned}&\text{ProvAggSum}(\texttt{?x}) = \text{sum}(\texttt{?x}),\\ &\text{ProvProd}(\texttt{?x}_1, \ldots, \texttt{?x}_n) = (\texttt{?x}_1 * \cdots * \texttt{?x}_n),\\ &\text{ProvDiff}(\texttt{?x}_1, \texttt{?x}_2) = \text{if}(\texttt{?x}_2 = 0, \texttt{?x}_1, 0).\end{aligned}$$

If a structure different from $\mathbb{N}$ were used, we could still have redefined these operations based on custom functions supported by the SPARQL engine. Indeed, several SPARQL engines allow the definition of custom functions. Therefore, our query rewriting can be used as the base for an efficient application of homomorhisms. We plan the study of such an extension as future work.

As another 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 execution of expensive queries for results that are not of interest of the user. ~~Finally,~~

~~we argue that the field of query processing and provenance for SPARQL would benefit from the development of new benchmarks that enable the evaluation of existing systems on centralized and federated versions of the same dataset.~~ Finally, we argue that the field of query processing and provenance for SPARQL would benefit from the development of new benchmarks specifically designed for provenance evaluation. While existing benchmarks such as FedShop [16] provide query workloads executable under different federation configurations, they do not directly support the evaluation of provenance computation—e.g., they lack ground-truth provenance annotations or queries designed to stress-test provenance systems. New benchmarks addressing these gaps would enable a more systematic comparison of provenance-aware systems across both centralized and federated settings.

Supplementary Material Statement

The source code of NPCS and Fed-NPCS [8], along with scripts to recreate the experimental setup, all required libraries, queries, and results can be found at `https://github.com/ZubariaForthAcc/NPCS_Fed-NPCS`.

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, German Research Foundation) under Germany's748 Excellence Strategy – EXC 2120/1 – 390831618; and the European Research Council (ERC) under the European Union's Horizon Europe Research and Innovation Programme, grant agreement No. 101200433, project META-LEARN..

References

1 Maribel Acosta, Olaf Hartig, and Juan Sequeda. *Federated RDF Query Processing*, pages 1–8. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-319-63962-8_228-2.

2 Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *ISWC (1)*, Lecture Notes in Computer Science, pages 18–34. Springer, 2011.

3 Julien Aimonier-Davat, Brice Nédelec, Minh Hoang Dang, Pascal Molli, and Hala Skaf-Molli. FedUP: Querying large-scale federations of SPARQL endpoints. In *WWW*, pages 2315–2324. ACM, 2024.

4 Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of rdf data management systems. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble, editors, *The Semantic Web – ISWC 2014*, pages 197–212, Cham, 2014. Springer International Publishing.

5 Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoc. Wdbench: A wikidata graph query benchmark. In *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings*, volume 13489 of *Lecture Notes in Computer Science*, pages 714–731, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-19433-7_41.

6 Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *ISWC*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2008.

7 Renzo Angles and Claudio Gutierrez. 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*, volume 9981 of *Lecture Notes in Computer Science*, pages 20–36, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-46523-4_2.

8 Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. Gprom - A swiss army knife for your provenance needs. *IEEE Data Engineering Bulletin*, 41(1):51–62,

[8] URL: `https://github.com/ZubariaForthAcc/NPCS_Fed-NPCS`

2018. URL: http://sites.computer.org/debull/A18mar/p51.pdf.

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

10 Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irini Fundulaki, and Katja Hose. NPCS: Native provenance computation for SPARQL. In *WWW-24*, 2024.

11 Carlos Buil-Aranda. *Federated Query Processing for the Semantic Web*, volume 15. IOS Press, 2014.

12 Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory — ICDT 2001*, pages 316–330, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

13 Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. SemaGrow: optimizing federated SPARQL queries. In *SEMANTiCS*, pages 121–128. ACM, 2015.

14 Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, jun 2000. doi:10.1145/357775.357777.

15 Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. 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*, volume 7649 of *Lecture Notes in Computer Science*, pages 625–640, Cham, 2012. Springer International Publishing. doi:10.1007/978-3-642-35176-1_39.

16 Minh-Hoang Dang, Julien Aimonier-Davat, Pascal Molli, Olaf Hartig, Hala Skaf-Molli, and Yotlan Le Crom. Fedshop: A benchmark for testing the scalability of sparql federation engines. In Terry R. Payne, Valentina Presutti, Guilin Qi, María Poveda-Villalón, Giorgos Stoilos, Laura Hollink, Zoi Kaoudi, Gong Cheng, and Juanzi Li, editors, *The Semantic Web – ISWC 2023: 22nd International Semantic Web Conference*, volume 14266 of *Lecture Notes in Computer Science*, pages 285–301, Cham, 2023. Springer. URL: https://github.com/GDD-Nantes/FedShop, doi:10.1007/978-3-031-47243-5_16.

17 Renata Queiroz Dividino, Sergej Sizov, Steffen Staab, and Bernhard Schueler. Querying for provenance, trust, uncertainty and other meta knowledge in RDF. *Journal of Web Semantics*, 7(3):204–219, 2009.

18 Luis Galárraga, Kim Ahlstrøm Jakobsen, Katja Hose, and Torben Bach Pedersen. Answering provenance-aware queries on RDF data cubes under memory budgets. In *ISWC (1)*, volume 11136 of *Lecture Notes in Computer Science*, pages 547–565. Springer, 2018.

19 Garima Gaur, Arnab Bhattacharya, and Srikanta Bedathur. How and why is an answer (still) correct? maintaining provenance in dynamic knowledge graphs. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management*, pages 405–414. ACM, 2020.

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

21 Floris Geerts and Antonella Poggi. On database query languages for k-relations. *Journal of Applied Logic*, 8(2):173–185, 2010.

22 Floris Geerts, Thomas Unger, Grigoris Karvounarakis, Irini Fundulaki, and Vassilis Christophides. Algebraic structures for capturing the provenance of SPARQL queries. *Journal of the ACM*, 63(1):7:1–7:63, 2016.

23 Boris Glavic and Gustavo Alonso. 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*, pages 174–185, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/ICDE.2009.15.

24 Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *COLD*, CEUR Workshop Proceedings. CEUR-WS.org, 2011.

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

26 Zhenzhen Gu, Francesco Corcoglioniti, Davide Lanti, Alessandro Mosca, Guohui Xiao, Jing Xiong, and Diego Calvanese. A systematic overview of data federation systems. *Semantic Web*, (Preprint):1–59, 2024.

27 Olaf Hartig. 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*, volume 5554 of *Lecture Notes in Computer Science*, pages 5–20, Cham, 2009. Springer International Publishing. doi:10.1007/978-3-642-02121-3_5.

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

29 Lars Heling and Maribel Acosta. Federated sparql query processing over heterogeneous linked data fragments. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1047–1057, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3485447.3511947.

30 Daniel Hernández, Luis Galárraga, and Katja Hose. Computing how-provenance for SPARQL queries via query rewriting. *Proc. VLDB Endow.*, 14(13):3389–3401, 2021. URL: http://www.vldb.org/pvldb/vol14/p3389-galarraga.pdf, doi:10.14778/3484224.3484235.

31 Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Processing aggregate queries in a federation of SPARQL endpoints. In

ESWC, Lecture Notes in Computer Science, pages 269–285. Springer, 2015.

32 Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Optimizing aggregate SPARQL queries using materialized RDF views. In *ISWC (1)*, volume 9981 of *Lecture Notes in Computer Science*, pages 341–359, 2016.

33 Kim Ahlstrøm Jakobsen, Katja Hose, and Torben Bach Pedersen. Towards answering provenance-enabled SPARQL queries over RDF data cubes. In *JIST*, Lecture Notes in Computer Science, pages 186–203. Springer, 2016.

34 Roman Kontchakov and Egor V. Kostylev. On expressibility of non-monotone operators in SPARQL. In *KR*, pages 369–379. AAAI Press, 2016.

35 Matteo Lissandrini, Katja Hose, and Torben Bach Pedersen. Example-driven exploratory analytics over knowledge graphs. In *EDBT*, pages 105–117. OpenProceedings.org, 2023.

36 Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *ISWC (1)*, Lecture Notes in Computer Science, pages 471–489. Springer, 2017.

37 Bofeng Pan, Natalia Stakhanova, and Suprio Ray. Data provenance in security and privacy. *ACM Comput. Surv.*, 55(14s), July 2023. doi:10.1145/3593294.

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

39 Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, Lecture Notes in Computer Science, pages 524–538. Springer, 2008.

40 Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proc. VLDB Endow.*, 12(9):975–988, May 2019. doi:10.14778/3329772.3329775.

41 Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation. In *ESWC*, Lecture Notes in Computer Science, pages 176–191. Springer, 2014.

42 Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2011*, pages 601–616, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

43 Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 601–616. Springer, 2011.

44 Aisling Third and John Domingue. Ethics and executability: Tracing decency in decentralised knowledge graph applications. In *ESWC 2023 Workshops and Tutorials. Semantic Methods for Events and Stories (SEMMES)*, volume 3443. CEUR Workshop Proceedings (CEUR-WS.org), July 2023. URL: https://ceur-ws.org/Vol-3443/ESWC%5f2023%5fTrusDeKW%5fpaper%5f3033.pdf.

45 Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. 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*, pages 455–466, New York, NY, USA, 2014. ACM. doi:10.1145/2566486.2568014.