

# 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-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.4.1.4

**Supplementary Material** Source code, queries, datasets, and experimental results for NPCS and Fed-NPCS.

*Software (Source Code, Dataset):* <https://doi.org/10.18419/DARUS-5944> [10]

**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; the German Research Foundation (DFG) under Germany’s Excellence



© Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irini Fundulaki, and Katja Hose; licensed under Creative Commons License CC-BY 4.0

Transactions on Graph Data and Knowledge, Vol. 4, Issue 1, Article No. 4, pp. 4:1–4:44



Transactions on Graph Data and Knowledge

TGDK Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

Received To be completed by Dagstuhl editorial office Accepted To be completed by Dagstuhl editorial office Published To be completed by Dagstuhl editorial office

## 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, 14, 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 [43] 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 [20]. 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,

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

and the SPARQL query

```
SELECT ?x WHERE { { UK, capital, ?x } UNION { ?x in UK ; a 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 [16, 22, 25], but this paper considers spm-semirings [22] because they are designed for the semantics of SPARQL, including

54 its non-monotonic fragment. We highlight that query provenance assumes the availability of  
55 identifiers for triples, in other words, it assumes that the KG has been reified using some scheme.  
56 Examples of reification schemes are RDF-star and named graphs.

57 In the centralized setting, there are essentially two main strategies to compute how-provenance  
58 for SPARQL queries. Methods such as TripleProv [44] opt for customized engines that compute  
59 provenance alongside query evaluation. Since provenance support is embedded in the engine,  
60 such solutions allow for advanced optimizations. On the downside, customized engines are not  
61 applicable to already deployed SPARQL endpoints. The other alternative is query rewriting [9, 30].  
62 In this approach, SPARQL queries are rewritten so that the new query retrieves both the query  
63 solutions and the polynomials describing their provenance, potentially with some post-processing.  
64 This design provides the flexibility to be applied to any SPARQL endpoint on the Web at the  
65 price of a runtime overhead. Both of these approaches have been applied for the centralized  
66 setting; however, to the best of our knowledge, no system exists that allows the computation of  
67 how-provenance for query results in the federated setting.

68 In this paper, we first describe the *NPCS*<sup>1</sup> approach [9], a method to compute provenance  
69 for SPARQL queries via query rewriting. NPCS is the first fully native SPARQL solution for  
70 how-provenance computation, as previous approaches [30] required some sort of post-processing.  
71 Moreover, NPCS supports different data reification schemes, including RDF-star. NPCS is designed  
72 for centralized KGs; in this paper we introduce an extension of the method, called *Fed-NPCS*, which  
73 is applicable for queries run against federated systems. This new work includes an extension of  
74 the how-provenance formalism for federated settings, as well as a runtime evaluation of Fed-NPCS  
75 on FedShop [17], a challenging benchmark for SPARQL federated systems. To the best of our  
76 knowledge, this is the first work that addresses the computation of how-provenance explanations  
77 for SPARQL queries on federations.

78 Our experimental evaluation demonstrates that NPCS is consistently faster than SPARQL-  
79 prov [30], the state of the art in how-provenance in SPARQL for centralized KGs. Moreover,  
80 NPCS and Fed-NPCS provide provenance explanations with reasonable runtime overhead (around  
81 10%) while scaling efficiently to large datasets with billions of triples and federations of up to 200  
82 endpoints.

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

## 89 **2 Related work**

90 We survey the literature on provenance for query results along two axes: provenance models  
91 (Section 2.1) and provenance support for RDF/SPARQL engines (Section 2.2). For a survey on  
92 federated SPARQL systems and related benchmarks we refer the reader to [17, 26]. We highlight  
93 that, so far, no other work has addressed the problem of computing how-provenance for SPARQL  
94 federated queries.

---

<sup>1</sup> Native Provenance Computation for SPARQL

## 95 2.1 Semirings and Provenance Models

96 Semirings were first used to model query provenance in the groundbreaking work by Green et  
 97 al. [25]. This work proposed *commutative semirings* to annotate query results for selection,  
 98 projection, join, and union queries for Datalog and the positive fragment of relational algebra.  
 99 Commutative semirings cannot model provenance for non-monotonic operators such as the left-  
 100 outer join and the difference [25], hence the algebraic structures were expanded to include a  
 101 monus operator<sup>2</sup> that accounts for the relational difference [21]. Commutative semirings and their  
 102 extensions model provenance as polynomial expressions. These expressions, called *how-provenance*,  
 103 encode both the sources and the data transformations required to obtain (and sometimes exclude)  
 104 a particular query answer. How-provenance is more expressive than other provenance models such  
 105 as lineage [15] or why-provenance [13].

106 Damasio et al. [16] showed that by rewriting SPARQL queries into relational algebra, we can  
 107 provide provenance annotations for SPARQL queries using *m-semirings*. However, Geerts et al. [22]  
 108 showed that these can yield very long and complex provenance expressions, and thus developed  
 109 the *spm-semirings* formalism (spm stands for SPARQL Minus) to overcome these limitations.  
 110 Spm-semirings guarantee more compact explanations and offer native support for non-monotonic  
 111 SPARQL operators such as OPTIONAL and MINUS. Since how-provenance polynomials are abstract  
 112 annotations, they are useful to a handful of metadata management applications [18, 27] via the  
 113 notion of commutation with homomorphisms.

## 114 2.2 Provenance-supported SPARQL engines

115 Wylot et al. [44] introduced TripleProv, a system to compute provenance annotations in the  
 116 commutative semiring framework for queries with basic graph patterns, union, and the OPTIONAL  
 117 operator. Due to its reliance on commutative semirings, TripleProv cannot guarantee commutation  
 118 with homomorphisms for queries involving the non-monotonic OPTIONAL operator. Additionally,  
 119 TripleProv uses a customized engine that organizes data into molecules—sort of indexes for star  
 120 patterns—, and thus it cannot be used on already deployed SPARQL engines. This is why our  
 121 approach resorts to query rewriting for how-provenance computation.

122 But approaches based on query rewriting are not rare at all. Perm [23] and GProM [8] are two  
 123 examples. Such approaches are, however, tailored for relational databases. Hence, they are not  
 124 applicable to SPARQL queries out of the box. Similarly to TripleProv, none of these methods can  
 125 properly support non-monotonic SPARQL queries because Perm is based on the lineage model,  
 126 and GProM relies on commutative semirings.

127 While the work of Geerts et al. [22] was the first to study provenance for the non-monotonic  
 128 fragment of SPARQL, the first concrete method to compute how-provenance under the spm-  
 129 semiring formalism was proposed by Hernández et al. [30]. They introduced SPARQLprov, a  
 130 method based on query rewriting that can annotate query results with how-provenance polynomials  
 131 for both monotonic and non-monotonic queries, establishing query rewriting as a practical  
 132 approach for computing SPARQL how-provenance. Contrary to NPCCS, SPARQLprov is not a  
 133 100% SPARQL solution because it relies on a subsequent decoding phase to compute the final  
 134 provenance annotations from the results of the rewritten query. As our experimental evaluation  
 135 shows, this decoding phase can incur prohibitive runtime overheads for non-selective queries.

---

<sup>2</sup> 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]).

### 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  $D : Y \rightarrow \mathbf{G}$ .*

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 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  $Q = (\text{SERVICE } y \ Q')$  is a fully remote query.*
- *Given two queries  $Q_1$  and  $Q_2$ , the expressions  $(Q_1 \text{ AND } Q_2)$ ,  $(Q_1 \text{ UNION } Q_2)$ ,  $(Q_1 \text{ DIFF } Q_2)$ , and  $(Q_1 \text{ OPTIONAL } Q_2)$  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.*

176 ■ A selection formula consists of a combination of atoms of the form  $A = B$  and  $\text{bound}(\text{?x})$ ,  
 177 where  $A, B \in \mathbf{I} \cup \mathbf{L}$  and  $\text{?x} \in \mathbf{V}$ , with the logical connectives  $\wedge$ ,  $\vee$ , and  $\neg$ . Given a query  $Q'$   
 178 and a selection formula  $\varphi$ , the expression  $Q = (Q' \text{ FILTER } \varphi)$  is a query, which is local if  $Q'$   
 179 is local, fully remote if  $Q'$  is fully remote, and remote if  $Q'$  is remote.

180 ■ Given a query  $Q'$  and a finite set of variables  $W$ , the expression  $Q = (\text{SELECT } W \text{ WHERE } Q')$   
 181 is a query, which is local if  $Q'$  is local, fully remote if  $Q'$  is fully remote, and remote if  $Q'$  is  
 182 remote.

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

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

187 ■  $Q_1 = (T_1 \text{ AND } T_2)$

188 ■  $Q_2 = (T_1 \text{ AND } (\text{SERVICE } y_2 \ T_2))$

189 ■  $Q_3 = ((\text{SERVICE } y_1 \ T_1) \text{ AND } (\text{SERVICE } y_2 \ T_2))$

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

193 ► **Note 6.** The syntax described in Definition 3 follows the syntax introduced by Perez et al. [38],  
 194 and extended in Geerts et al. [22] with the operator SELECT, and by Buil-Aranda [12] with the  
 195 operator SERVICE. Unlike them [22, 38], we call the difference operator DIFF because it differs  
 196 from the standard MINUS operator [6, 34]. The operator DIFF is part of the standard SPARQL  
 197 1.1 algebra, and the operator MINUS is part of the standard SPARQL 1.1 query language syntax.  
 198 Also, the operator MINUS is expressible with the other operators we include in Definition 3 [34].  
 199 An additional difference compared to the above works is that we syntactically classify queries  
 200 in *local*, *remote* and *fully-remote* depending on where the triple patterns are evaluated. In this  
 201 paper we build upon the how-provenance theoretical framework by Geerts et al. [22] who used the  
 202 operator DIFF in their annotated SPARQL algebra (but called it MINUS).

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

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

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

220 ■ If  $\{\}$  is an empty basic graph pattern, then  $\llbracket \{\} \rrbracket_{D,G}$  is the set with a single mapping  $\mu_\emptyset$  such  
 221 that  $\text{dom}(\mu) = \emptyset$ .

- 222 ■ If  $T$  is a triple pattern, then  $\llbracket T \rrbracket_{D,G}$  is the set of mappings  $\mu$  such that  $\text{dom}(\mu)$  is the set of  
 223 variables occurring in  $T$  and  $\mu(T) \in G$ .
- 224 ■  $\llbracket (\text{SERVICE } y \ Q) \rrbracket_{D,G} = \llbracket Q \rrbracket_{D,D(y)}$ .
- 225 ■  $\llbracket Q_1 \text{ AND } Q_2 \rrbracket_{D,G}$  is the set of all mappings  $\mu$  such that  $\mu_1 \in \llbracket Q_1 \rrbracket_{D,G}$  and  $\mu_2 \in \llbracket Q_2 \rrbracket_{D,G}$ ,  
 226  $\mu_1 \sim \mu_2$ , and  $\mu = \mu_1 \cup \mu_2$ .
- 227 ■  $\llbracket Q_1 \text{ UNION } Q_2 \rrbracket_{D,G}$  is the set of all mappings  $\mu$  such that  $\mu \in \llbracket Q_1 \rrbracket_{D,G}$  or  $\mu \in \llbracket Q_2 \rrbracket_{D,G}$ .
- 228 ■  $\llbracket Q_1 \text{ DIFF } Q_2 \rrbracket_{D,G}$  is the set of all mappings  $\mu_1$  such that  $\mu_1 \in \llbracket Q_1 \rrbracket_{D,G}$  and every mapping  
 229  $\mu_2 \in \llbracket Q_2 \rrbracket_{D,G}$  is incompatible with  $\mu_1$ .
- 230 ■  $\llbracket Q_1 \text{ OPTIONAL } Q_2 \rrbracket_{D,G}$  is the set of all mappings  $\mu$  such that  $\mu \in \llbracket Q_1 \text{ AND } Q_2 \rrbracket_{D,G}$  or  $\mu \in$   
 231  $\llbracket Q_1 \text{ DIFF } Q_2 \rrbracket_{D,G}$ .
- 232 ■  $\llbracket Q \text{ FILTER } \varphi \rrbracket_{D,G}$  is the set of all mappings  $\mu$  such that  $\mu \in \llbracket Q \rrbracket_{D,G}$  and  $\varphi$  is a true formula  
 233 according to mapping  $\mu$ , denoted  $\mu \models \varphi$ .
- 234 We write  $\mu \models \varphi$  if the value of the formula  $\varphi$  according to  $\mu$ , denoted  $\varphi[\mu]$  is 1. The value  $\varphi[\mu]$   
 235 is one the values in  $\{0, \frac{1}{2}, 1\}$  defined recursively as follows:
- 236 ■ Let  $\varphi$  be the a formula of the form  $A = B$ . If there is a variable in  $\varphi$  that is not in  $\text{dom}(\mu)$ ,  
 237 then  $\varphi[\mu] = \frac{1}{2}$ . Otherwise, the formula  $\mu(\varphi)$  resulting from replacing the variables  $?x$  in  $\varphi$   
 238 with  $\mu(?x)$  can have the forms  $a = a$  or  $a = b$ , where  $a$  and  $b$  are two different values in the  
 239 set  $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$  (e.g., if  $\varphi$  is the formula  $?x = a$  and  $\mu(?x) = a$  then the  $\mu(\varphi)$  is  $a = a$ ). If  
 240  $\mu(\varphi)$  is  $a = a$  then  $\varphi[\mu] = 1$ . Otherwise, if  $\mu(\varphi)$  is  $a = b$  then  $\varphi[\mu] = 0$ .
- 241 ■ If  $\varphi$  has the form  $\text{bound}(?x)$ , then  $\varphi[\mu] = 1$  if  $?x \in \text{dom}(\mu)$ . Otherwise,  $\varphi[\mu] = 0$ .
- 242 ■  $(\neg\varphi)[\mu] = 1 - \varphi[\mu]$ ,  $(\varphi \wedge \psi)[\mu] = \min(\varphi[\mu], \psi[\mu])$ , and  $(\varphi \vee \psi)[\mu] = \max(\varphi[\mu], \psi[\mu])$ .
- 243 ■  $\llbracket \text{SELECT } W \text{ WHERE } Q \rrbracket_{D,G}$  is the set of all mappings  $\mu$  such that there exists a mapping  
 244  $\mu' \in \llbracket Q \rrbracket_{D,G}$  with  $\mu = \mu'|_W$ .

245 ► **Note 9.** It is not difficult to see that if a query  $Q$  is fully-remote, then the set  $\llbracket Q \rrbracket_{D,G}$  does not  
 246 depend on the graph  $G$ . Similarly, if query  $Q$  is local, then the set  $\llbracket Q \rrbracket_{D,G}$  does not depend on  
 247 the federated dataset  $D$ . Following this observation, we will abbreviate the aforementioned set as  
 248  $\llbracket Q \rrbracket_D$  if  $Q$  is fully-remote, and as  $\llbracket Q \rrbracket_G$  if  $Q$  is local.

249 Not all variables occurring in a query necessarily appear on the solutions. The problem of deter-  
 250 mining which variables can appear and which variables cannot appear is in general undecidable [12],  
 251 but an approximation can be defined syntactically. A variable is said *in-scope* when it can appear  
 252 in the solutions and *strongly bound* when it necessarily appears in all the solutions.

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

- 256 ■  $\text{inScope}(\{\}) = \emptyset$  and  $\text{stronglyBound}(\{\}) = \emptyset$ .
- 257 ■ *Given a triple pattern  $T$ . The sets  $\text{inScope}(T)$  and  $\text{stronglyBound}(T)$  are equal to the set of*  
 258 *variables occurring in  $T$ .*
- 259 ■ *Given the queries  $Q_1$  and  $Q_2$ , the following identities define the in-scope variables in compound*  
 260 *queries:*
- 261 ■  $\text{inScope}(\text{SERVICE } y \ Q_1) = \text{inScope}(Q_1)$ ,
- 262 ■  $\text{inScope}(Q_1 \text{ AND } Q_2) = \text{inScope}(Q_1) \cup \text{inScope}(Q_2)$ ,
- 263 ■  $\text{inScope}(Q_1 \text{ UNION } Q_2) = \text{inScope}(Q_1) \cup \text{inScope}(Q_2)$ ,
- 264 ■  $\text{inScope}(Q_1 \text{ OPTIONAL } Q_2) = \text{inScope}(Q_1) \cup \text{inScope}(Q_2)$ ,
- 265 ■  $\text{inScope}(Q_1 \text{ DIFF } Q_2) = \text{inScope}(Q_1)$ ,
- 266 ■  $\text{inScope}(Q_1 \text{ FILTER } \varphi) = \text{inScope}(Q_1)$ ,
- 267 ■  $\text{inScope}(\text{SELECT } W \text{ WHERE } Q_1) = W \cap \text{inScope}(Q_1)$ .

268 ■ Given the queries  $Q_1$  and  $Q_2$ , the following identities define the strongly bound variables in  
269 compound queries:

- 270 ■  $\text{stronglyBound}(\text{SERVICE } y \ Q_1) = \text{stronglyBound}(Q_1)$
- 271 ■  $\text{stronglyBound}(Q_1 \ \text{AND} \ Q_2) = \text{stronglyBound}(Q_1) \cap \text{stronglyBound}(Q_2),$
- 272 ■  $\text{stronglyBound}(Q_1 \ \text{UNION} \ Q_2) = \text{stronglyBound}(Q_1) \cap \text{stronglyBound}(Q_2),$
- 273 ■  $\text{stronglyBound}(Q_1 \ \text{OPTIONAL} \ Q_2) = \text{stronglyBound}(Q_1) \cap \text{stronglyBound}(Q_2),$
- 274 ■  $\text{stronglyBound}(Q_1 \ \text{DIFF} \ Q_2) = \text{stronglyBound}(Q_1),$
- 275 ■  $\text{stronglyBound}(Q_1 \ \text{FILTER} \ \varphi) = \text{stronglyBound}(Q_1),$
- 276 ■  $\text{stronglyBound}(\text{SELECT } W \ \text{WHERE} \ Q_1) = W \cap \text{stronglyBound}(Q_1).$

277 ► **Note 11.** One may think that  $?x \in \text{inScope}(Q)$  implies that there exists one graph where there  
278 is a mapping  $\mu \in \llbracket Q \rrbracket_G$  such that  $?x \in \text{dom}(\mu)$ . However, this is not true. Indeed, the query

$$279 \quad Q = (\{\} \ \text{OPTIONAL} \ ((a, p, ?x) \ \text{DIFF} \ (a, p, ?x)))$$

280 includes the variable  $?x$  in  $\text{inScope}(Q)$  and for every graph  $G$ , each  $\mu \in \llbracket Q \rrbracket_G$  holds that  $?x \notin$   
281  $\text{dom}(\mu)$ . One may think something similar regarding the variables in  $\text{stronglyBound}(Q)$ . That is,  
282 thinking that every variable  $?x$  that occurs in the domain of all solution mappings of a query  $Q$  in  
283 every graph  $G$  must be a strongly bound. This happens to variable  $?x$  in the query

$$284 \quad Q' = ((a, p, ?x) \ \text{UNION} \ (\{\} \ \text{FILTER} \ \neg(a = a))).$$

285 However, variable  $?x \notin \text{stronglyBound}(Q')$ . The problem of determining if a variable can occur in  
286 the domain of all or some of the mappings is in general undecidable [12]. Hence, the notions of  
287  $\text{inScope}$  and  $\text{stronglyBound}$  are approximations of these more complex notions. The variables in  
288  $\text{inScope}(Q)$  are a superset of the variables that occur in some mappings  $\mu \in \llbracket Q \rrbracket_G$  for every graph  
289  $G$ . The variables in  $\text{stronglyBound}(Q)$  are a subset of the variables that occur in all mappings  
290  $\mu \in \llbracket Q \rrbracket_G$  for every graph  $G$ .

### 291 3.3 Federated Query Processing in RDF/SPARQL

292 Federated querying consists of executing queries across multiple, potentially heterogeneous, RDF  
293 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  $\llbracket \cdot \rrbracket_F$  is defined in terms of the evaluation function over graphs  $\llbracket \cdot \rrbracket_G$ . For a local SPARQL  
query  $Q$ ,  $\llbracket Q \rrbracket_F = \llbracket Q \rrbracket_G$  where the graph  $G$ , called the *union graph*, is defined as follows:

$$G = \llbracket Q \rrbracket_{\bigcup_{e \in E} d(e)}.$$

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

296 Notice that query  $Q$  is local. That is,  $Q$  does not contain `SERVICE` clauses. However, the  
297 federated query processing of  $Q$  consists of execution  $Q$  over a set of graphs from remote services.  
298 Since the `SERVICE` clause allows for remote query execution, one may think that a federated query  
299 processing can be implemented by rewriting the local query  $Q$  as a fully remote query. Indeed, the  
300 following subsection describes how federated SPARQL query engines decompose the local query  $Q$   
301 into multiple queries that are remotely executed by rewriting  $Q$  as a single fully remote query  
302 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:

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

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:

$$T_1 = (?city, a, City),$$

$$T_2 = (?city, country, Germany),$$

$$T_3 = (Rhine, crosses, ?city).$$

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,

$$\llbracket T_1 \rrbracket_{G_1 \cup G_2} = \llbracket T_1 \rrbracket_{G_1} \cup \llbracket T_1 \rrbracket_{G_2}.$$

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

$$\llbracket T_1 \rrbracket_{G_1 \cup G_2} = \llbracket ((\text{SERVICE } y_2 \ T_1)) \text{ UNION } ((\text{SERVICE } y_1 \ T_1)) \rrbracket_D.$$

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.,  $\llbracket Q \rrbracket_{G_1 \cup G_2} = \llbracket Q' \rrbracket_D$ ). Such a query  $Q'$  is the following:

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

Schwarte et al. [42] 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  $\llbracket T_1 \rrbracket_{G_2}$ ,  $\llbracket T_2 \rrbracket_{G_2}$ , and  $\llbracket T_3 \rrbracket_{G_1}$  are empty, then the result of evaluating  $Q'$  in the federation will be equal to the result of evaluating the following query  $Q''$  on  $D$ :

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

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

### 346 3.4 How-provenance in SPARQL

#### 347 3.4.1 Semirings

348 A *commutative monoid*  $\mathcal{M}$  is an algebraic structure  $(M, +_{\mathcal{M}}, 0_{\mathcal{M}})$  such that  $M \neq \emptyset$  is a set closed  
 349 under a commutative and associate binary operation  $+_{\mathcal{M}}$ . The element  $0_{\mathcal{M}}$  is the identity operand  
 350 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  
 351 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  
 352 *commutative semiring*. An *spm-semiring*  $(K, +_{\mathcal{K}}, \times_{\mathcal{K}}, -_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$  extends a commutative semiring  
 353 with a minus operation  $-_{\mathcal{K}}$ . This operator follows a set of axioms that allows us to model  
 354 non-monotonic operations such as the relational difference. For more details about spm-semirings,  
 355 we refer the reader to [22]. The algebraic expressions within an spm-semiring are used to annotate  
 356 query solutions. To see how, we need to introduce the concepts of  $\mathcal{K}$ -relations and  $\mathcal{K}$ -graphs.

#### 357 3.4.2 $\mathcal{K}$ -relations and $\mathcal{K}$ -graphs

358 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  $f : A \rightarrow K$  is called  
 359 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  
 360 element  $a \in A$ , we call  $f(a)$  the  *$\mathcal{K}$ -value* of  $a$  in  $f$ . If  $f$  has a finite support  $\{a_1, \dots, a_n\}$ , we write  
 361  $f = \{a_1 \mapsto k_1, \dots, a_n \mapsto k_n\}$  to denote that  $f(a_i) = k_i$ , for  $1 \leq i \leq n$ .

362 Intuitively,  $\mathcal{K}$ -sets annotate the elements of a set  $A$  with values in the structure. By assuming  
 363 that  $\mathcal{K}$  is the structure of provenance annotations, this formalism is used to represent provenance  
 364 of answer to queries or statements in an RDF graph. A  $\mathcal{K}$ -set  $\Omega$  over the set of all possible  
 365 SPARQL mappings is called a  *$\mathcal{K}$ -relation*, and a  $\mathcal{K}$ -set  $\Gamma$  over all possible RDF triples is called a  
 366  *$\mathcal{K}$ -graph*.

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

371 ► **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  $G$   
 372 and  $Q$  be the following RDF-star graph and query:

$$373 \quad G = \{ ((\text{Alice}, \text{likes}, \text{pasta}), \text{wasDerivedFrom}, u_1), \\ ((\text{Alice}, \text{likes}, \text{pasta}), \text{wasDerivedFrom}, u_2), \\ ((\text{Alice}, \text{livesIn}, \text{Italy}), \text{wasDerivedFrom}, u_3) \},$$

$$374 \quad Q = (?x, \text{likes}, \text{pasta}) \text{ AND } (?x, \text{livesIn}, \text{Italy}).$$

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

380 Alice is a query solution for  $Q$  as long as the triple identified by  $u_3$  is present in conjunction with  
 381 either the triples  $u_1$  or  $u_2$ .

382 ► **Definition 16.** Given an *spm-semiring*  $\mathcal{K}$ , a SPARQL query  $Q$  consisting of a combination  
 383 of triple patterns with the operators AND, UNION, DIFF, FILTER, OPTIONAL and SELECT, and a  
 384  $\mathcal{K}$ -graph  $\Gamma$ , we write  $\llbracket Q \rrbracket_\Gamma$  to denote the  $\mathcal{K}$ -relation obtained from evaluating  $Q$  in  $\Gamma$ , which is  
 385 defined recursively for an arbitrary mapping  $\mu$  as follows:

- 386 ■  $\llbracket \{\} \rrbracket_\Gamma(\mu) = 1$  if  $\text{dom}(\mu) = \emptyset$ , and  $\llbracket \{\} \rrbracket_\Gamma(\mu) = 0$  if  $\text{dom}(\mu) \neq \emptyset$ ,
  - 387 ■  $\llbracket (s, p, o) \rrbracket_\Gamma(\mu) = \Gamma(\mu(s, p, o))$ ,
  - 388 ■  $\llbracket \text{SELECT } W \text{ WHERE } Q \rrbracket_\Gamma(\mu) = \sum_{\mu': \mu'|_W = \mu} \llbracket Q \rrbracket_\Gamma(\mu')$ ,
  - 389 ■  $\llbracket Q \text{ FILTER } \varphi \rrbracket_\Gamma(\mu) = \llbracket Q \rrbracket_\Gamma(\mu) \times_{\mathcal{K}} 1_{\mu \models \varphi}$ , where  $1_{\mu \models \varphi} = 1$  if  $\mu \models \varphi$  and  $1_{\mu \models \varphi} = 0$ , otherwise,
  - 390 ■  $\llbracket Q_1 \text{ UNION } Q_2 \rrbracket_\Gamma(\mu) = \llbracket Q_1 \rrbracket_\Gamma(\mu) +_{\mathcal{K}} \llbracket Q_2 \rrbracket_\Gamma(\mu)$ ,
  - 391 ■  $\llbracket Q_1 \text{ AND } Q_2 \rrbracket_\Gamma(\mu) = \sum_{\mu = \mu_1 \cup \mu_2} (\llbracket Q_1 \rrbracket_\Gamma(\mu_1) \times_{\mathcal{K}} \llbracket Q_2 \rrbracket_\Gamma(\mu_2))$ ,
  - 392 ■  $\llbracket Q_1 \text{ DIFF } Q_2 \rrbracket_\Gamma(\mu) = \llbracket Q_1 \rrbracket_\Gamma(\mu) -_{\mathcal{K}} (\sum_{\mu' \sim \mu} \llbracket Q_2 \rrbracket_\Gamma(\mu'))$ ,
  - 393 ■  $\llbracket Q_1 \text{ OPTIONAL } Q_2 \rrbracket_\Gamma(\mu) = \llbracket Q_1 \text{ AND } Q_2 \rrbracket_\Gamma(\mu) +_{\mathcal{K}} \llbracket Q_1 \text{ DIFF } Q_2 \rrbracket_\Gamma(\mu)$ ,
- 394 where  $\sum$  denotes sums using the operation  $+_{\mathcal{K}}$ .

### 395 3.4.3 How-provenance polynomials

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

400 ► **Definition 17 (spm-homomorphism).** An *spm-homomorphism* between two *spm-semirings*  $\mathcal{K}_1 =$   
 401  $(K, +_{\mathcal{K}_1}, \times_{\mathcal{K}_1}, -_{\mathcal{K}_1}, 0_{\mathcal{K}_1}, 1_{\mathcal{K}_1})$  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  $h : K_1 \rightarrow K_2$   
 402 that preserves the identities and the operations. That is:

- 403 ■  $h(0_{\mathcal{K}_1}) = 0_{\mathcal{K}_2}$ ,
- 404 ■  $h(1_{\mathcal{K}_1}) = 1_{\mathcal{K}_2}$ ,
- 405 ■  $h(a +_{\mathcal{K}_1} b) = h(a) +_{\mathcal{K}_2} h(b)$ ,
- 406 ■  $h(a \times_{\mathcal{K}_1} b) = h(a) \times_{\mathcal{K}_2} h(b)$ ,
- 407 ■  $h(a -_{\mathcal{K}_1} b) = h(a) -_{\mathcal{K}_2} h(b)$ .

408 ► **Example 18.** Consider the following *spm-semirings*:

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

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

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

420 We write  $\cong$  to denote the congruence relation between polynomials in  $\mathbb{N}[X_B]$ . That is, if  
 421  $p[X_B] \cong q[X_B]$  and  $p'[X_B] \cong q'[X_B]$ , then  $p[X_B] + p'[X_B] \cong q[X_B] + q'[X_B]$  and  $p[X_B] \cdot p'[X_B] \cong$   
 422  $q[X_B] \cdot q'[X_B]$ . In addition,  $\cong$  is assumed to be the minimum congruence such that for all monomials  
 423  $\mu \in \text{mon}(X)$ ,  $b_\mu \cdot \bar{b}_\mu \cong 0$ ,  $b_\mu + \bar{b}_\mu \cong 1$ ,  $b_\mu + b_\mu \cong b_\mu$  and  $\bar{b}_\mu + \bar{b}_\mu \cong \bar{b}_\mu$ ,  $\bar{b}_m \cdot \mu \cong 0$ , and  $b_\mu \cdot b_\nu \cong b_\mu$

424 whenever  $\mu = \nu \cdot \nu'$  for some monomial  $\nu'$ . Since these last entities characterize the elements of  
 425  $B_X$  and  $\bar{B}_X$  as Booleans, the quotient semiring of  $\mathbb{N}[X_B]$  with respect to  $\cong$  is called the *semiring*  
 426 *of boolynomials*. Abusing of notation, we write  $(\mathbb{N}_{\cong}[X_B], +, \cdot, [0], [1])$  for this quotient semiring,  
 427 where the elements of  $\mathbb{N}_{\cong}[X_B]$  are the equivalence classes  $[p[X_B]] = \{p'[X_B] \mid p[X_B] \cong p'[X_B]\}$ ,  
 428 and the operations are  $[p[X_B]] + [q[X_B]] = [p[X_B] + q[X_B]]$  and  $[p[X_B]] \cdot [q[X_B]] = [p[X_B] \cdot q[X_B]]$ .

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

434 In what follows, we write  $\text{ProvExp}(X)$  to denote the set of all the expressions generated with  
 435 set of variables  $X$ , the operator symbols  $\oplus$ ,  $\otimes$ , and  $\ominus$ , and the constants 0 and 1. The semantics  
 436 of these expressions is given by associating every expression in  $\text{ProvExp}(X)$  to an element in  
 437  $\mathbb{N}_{\cong}[X_B]$  by mapping every variable  $x \in X$  to an equivalence class  $[x]$ , the constants 0 and 1 with  
 438 the equivalence classes  $[0]$  and  $[1]$ , and the operators  $\oplus$ ,  $\otimes$ , and  $\ominus$  with the operators  $+$ ,  $\cdot$ , and  $-$   
 439 of the universal spm-semiring. The expressions computed by the methods proposed by Hernández  
 440 et al. [30] and Asma et al. [9] represent thus elements in the universal spm-semiring.

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

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

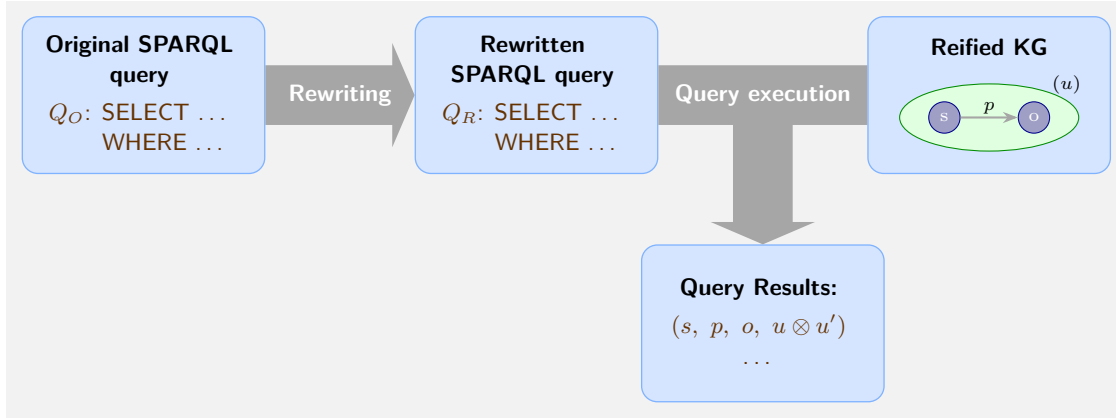
- 449 ■  $h([x]) = g(x)$  if  $x \in X$ .
- 450 ■  $h([0]) = 0_{\mathcal{K}}$  and  $h([1]) = 1_{\mathcal{K}}$ .
- 451 ■  $h([a \oplus b]) = h([a]) +_{\mathcal{K}} h([b])$ ,  $h([a \otimes b]) = h([a]) \times_{\mathcal{K}} h([b])$ , and  $h([a \ominus b]) = h([a]) -_{\mathcal{K}} h([b])$ .

452 The function  $h$  is an spm-homomorphism.

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

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

460 ► **Example 21.** Let  $\Gamma_1$  be the  $\text{ProvExp}_{\cong}(X)$ -graph whose support has two triples annot-  
 461 ated as follows:  $\{[(\text{Danube}, \text{crosses}, \text{Ulm}) \mapsto x_1, (\text{Danube}, \text{crosses}, \text{Budapest}) \mapsto x_2]\}$ , where  $x_1$   
 462 and  $x_2$  are elements in set  $X$ . Let  $g : X \rightarrow \mathbb{N}$  be the function such that  $g(x_1) =$   
 463  $3$  and  $g(x_2) = 5$ , and  $h : \text{ProvExp}_{\cong}(X) \rightarrow \mathbb{N}$  be the spm-homomorphism defined on  
 464 top of function  $g$  as it is described in Example 19. Then,  $\Gamma_2 = \Gamma_1 \circ h$  is the  $\mathbb{N}$ -  
 465 graph  $\{[(\text{Danube}, \text{crosses}, \text{Ulm}) \mapsto 3, (\text{Danube}, \text{crosses}, \text{Budapest}) \mapsto 5]\}$ . Let  $Q$  be the query  
 466  $(\text{SELECT ?river WHERE } ((\text{?river}, \text{crosses}, \text{?city})))$ , asking for rivers crossing cities. Then,  $(Q)_{\Gamma_1}$   
 467 is the  $\mathbb{N}$ -relation  $\{[\text{Danube} \mapsto x_1 \oplus x_2]\}$ . Then  $(Q)_{\Gamma_1} \circ h$  is the  $\mathbb{N}$ -relation  $\{[\text{Danube} \mapsto 8]\}$ , which is  
 468 equal to  $(Q)_{\Gamma_2}$ .



■ **Figure 1** The query rewriting process for NPCS.

469 ▶ **Remark 22.** Example 21 illustrates the generality of how-provenance polynomials. It is well-  
 470 known that every pair of SPARQL algebras commute with homomorphisms. Also, by using the  
 471 spm-homomorphism like the one described in Example 19, we can use how-provenance polynomials  
 472 to symbolically operate elements of other spm-semirings. Thus, the how-provenance polynomials  
 473 generated with the method we propose in the next section, NPCS, can be used in downstream  
 474 tasks over other spm-semirings.

## 475 4 Provenance computation with NPCS

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

479 Given a finite set  $X \subset \mathbf{I}$ , an  $X$ -graph  $\Gamma$ , its corresponding RDF-star graph  $G$ , and a SPARQL  
 480 query  $Q_O$ , NPCS rewrites  $Q_O$  into  $Q_R$  so that  $\llbracket Q_R \rrbracket_G$  is a set of mappings with an additional  
 481 variable that encodes the provenance polynomials. Thus, the output of  $Q_R$  represents the  
 482  $\text{ProvExp}_{\cong}(X)$ -relation  $\llbracket Q_O \rrbracket_{\Gamma}$  that maps each solution to a how-provenance polynomial explanation.  
 483 Those polynomials lie in an spm-semiring  $(\text{ProvExp}_{\cong}(X), \oplus, \otimes, \ominus, 0, 1)$ , where  $X$  is the set of  
 484 triple identifiers in  $G$ . We highlight that computing provenance assumes that the triples in the  
 485 graph are reified, i.e., identified. RDF-star is a natural way to do it, but as shown later, our  
 486 approach supports any reification scheme for RDF data. The query rewriting process of NPCS is  
 487 depicted in Figure 1.

### 488 4.1 Our Query Rewriting in a Nutshell

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

$$493 \left[ \begin{array}{c|c} ?x & ?prov \\ \hline \text{Alice} & (u_1 \oplus u_2) \otimes u_3 \end{array} \right].$$

494 The column labeled  $?prov$  stores the how-provenance of each of the query solutions. To compute  
 495 such an expression, our strategy must rewrite the query such that the rewritten query retrieves

## 4:14 Native Provenance Computation for Federated and Non-Federated SPARQL Queries

496 the identifiers of the triples that match each of the triple patterns. However, this is not enough.  
 497 For instance, the triple pattern  $P_1 = (?x, \text{likes}, \text{pasta})$  has two matches, i.e.,  $u_1$  and  $u_2$ , that must  
 498 be *grouped* into the expression  $(u_1 \oplus u_2)$ . This term tells us that the presence of at least one of  
 499 those sources guarantees the inclusion of Alice in the result set. Finally, the groups extracted from  
 500 each of the triple patterns must be combined with the  $\otimes$  operator that explains the semantics  
 501 of AND. We argue that to obtain the left-hand term of the product we can rewrite  $P_1$  into the  
 502 following sub-query:

```
503   P'_1 = (SELECT   ?x (ProvAggSum(?prov $\oplus$ 1 $\oplus$  $\odot$ ) AS ?prov $\oplus$ 1)
           WHERE     Reify((?x, likes, pasta), ?prov $\oplus$ 1 $\oplus$  $\odot$ )
           GROUP BY  ?x),
```

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

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

```
510   ((?x, likes, pasta), wasDerivedFrom, ?prov $\oplus$ 1 $\oplus$  $\odot$ ).
```

511 The intermediate variable  $?prov\oplus 1\oplus\odot$  is introduced to capture the identifiers of the triples  
 512 that match  $P_1$ , whereas variable  $?prov\oplus 1$  groups all the triple identifiers associated to a query  
 513 solution, which explains the group clause on  $?x$ . The function *ProvAggSum*, later explained,  
 514 combines the different sources into a summation with the operator  $\oplus$ .

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

```
524   Q' = (SELECT   ?x (ProvAggSum(?prov $\oplus$ ) AS ?prov)
           WHERE     ((P'_1 AND P'_2) BIND (ProvProd(?prov $\oplus$ 1, ?prov $\oplus$ 2) AS ?prov $\oplus$ ))
           GROUP BY  ?x).
```

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

529 ► **Definition 24.** Let  $?x, ?x_1, \dots, ?x_n$  be variables. Then, we define the following SPARQL  
 530 operators based on the built-in SPARQL functions *concat* and *aggregate\_concat*:

```
531   ProvAggSum(?x) = concat("( $\oplus$ ", aggregate_concat(?x), ")"),
532   ProvProd(?x_1, \dots, ?x_n) = concat("( $\otimes$ ", ?x_1, \dots, ?x_n, ")"),
533   ProvDiff(?x_1, ?x_2) = concat("( $\ominus$ ", ?x_1, ?x_2, ")").
```

534 ► **Example 25.** The expression  $\text{ProvProd}(\text{?prov}\oplus\otimes 1, \text{?prov}\oplus\otimes 2)$  in the query  $Q'$  of our running  
 535 example is evaluated against the answers of the query  $(P'_1 \text{ AND } P'_2)$ , which are described in the  
 536 following result set:

$$537 \left[ \begin{array}{c|cc} \text{?x} & \text{?prov}\oplus\otimes 1 & \text{?prov}\oplus\otimes 2 \\ \hline \text{Alice} & (u_1) & (u_3) \\ \text{Alice} & (u_2) & (u_3) \end{array} \right].$$

538 Then, this expression generates a third provenance column for the variable  $\text{?prov}\oplus$ :

$$539 \left[ \begin{array}{c|ccc} \text{?x} & \text{?prov}\oplus\otimes 1 & \text{?prov}\oplus\otimes 2 & \text{?prov}\oplus \\ \hline \text{Alice} & (u_1) & (u_3) & (\otimes(u_1)(u_3)) \\ \text{Alice} & (u_2) & (u_3) & (\otimes(u_2)(u_3)) \end{array} \right].$$

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

$$542 \left[ \begin{array}{c|c} \text{?x} & \text{?prov} \\ \hline \text{Alice} & (\oplus(\otimes(u_1)(u_3))(\otimes(u_2)(u_3))) \end{array} \right].$$

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

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

## 555 4.2 Base Rewriting Rules

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

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

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

$$563 \begin{array}{ll} (\text{SELECT} & \text{inScope}(Q) (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov}) \\ & \text{WHERE} \quad \text{Reify}((s, p, o), \text{?prov}\oplus) \\ & \text{GROUP BY} \quad \text{inScope}(Q) ). \end{array}$$

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

$$565 \begin{array}{ll} (\text{SELECT} & \text{inScope}(Q) (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov}) \\ & \text{WHERE} \quad (\beta(Q_1, \text{?prov}\oplus\otimes 1) \text{ AND } \beta(Q_2, \text{?prov}\oplus\otimes 2)) \\ & \quad \text{BIND} (\text{ProvProd}(\text{?prov}\oplus\otimes 1, \text{?prov}\oplus\otimes 2) \text{ AS } \text{?prov}\oplus) \\ & \text{GROUP BY} \quad \text{inScope}(Q) ). \end{array}$$

## 4:16 Native Provenance Computation for Federated and Non-Federated SPARQL Queries

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

```
567      (SELECT      inScope(Q) (ProvAggSum(?prov $\oplus$ ) AS ?prov)
        WHERE      ( $\beta(Q_1, ?\text{prov}\oplus)$  UNION  $\beta(Q_2, ?\text{prov}\oplus)$ )
        GROUP BY   inScope(Q)).
```

568 5. If  $Q$  is  $(Q_1 \text{ DIFF } Q_2)$ , then let  $\nu$  a variable substitution that substitutes with fresh variables the  
569 variables in  $\text{dom}(Q_1) \cap \text{dom}(Q_2)$  that are not strongly bound in query  $Q_1$ . Then,  $\beta(Q, ?\text{prov})$   
570 is the query

```
571      (SELECT      inScope(Q) (ProvDiff(?prov $\ominus$ 1, ProvAggSum(?prov $\ominus$ 2 $\oplus$ )) AS ?prov)
        WHERE      ( $\beta(Q_1, ?\text{prov}\ominus$ 1) OPTIONAL $_{C_v}$   $\beta(\nu(Q_2), ?\text{prov}\ominus$ 2 $\oplus$ ))
        GROUP BY   inScope(Q)  $\cup$  {?prov $\ominus$ 1}).
```

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

```
573      (SELECT      W (ProvAggSum(?prov $\oplus$ ) AS ?prov)
        WHERE       $\beta(Q', ?\text{prov}\oplus)$ 
        GROUP BY   W).
```

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

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

580 **► Note 30.** We omit the rewriting rule for the `OPTIONAL` operator because this operator can be  
581 written in terms of `AND`, `UNION` and `DIFF`. To be precise, the query  $P_1 \text{ OPTIONAL } P_2$  is equivalent  
582 to the query  $(P_1 \text{ DIFF } P_2) \text{ UNION } (P_1 \text{ AND } P_2)$ .

583 **► Note 31.** Since the `DIFF` operation requires tracking the provenance of both operands, `DIFF`  
584 translates to an `OPTIONAL` operation, more precisely, to an `OPTIONAL $_{C_v}$`  operation, which extends  
585 `OPTIONAL` by renaming variables in the optional pattern with fresh ones. This renaming discards  
586 undesired bindings produced in the subtrahend while tracking the provenance. For example,  
587 consider the patterns  $P_1(?x, ?y)$  and  $P_2(?x, ?y, ?z)$ , whose in-scope variables are indicated in the  
588 parenthesis, and let  $\mu_1 = \{?x \mapsto a\}$  and  $\mu_2 = \{?x \mapsto a, ?y \mapsto b, ?z \mapsto c\}$  be two solutions for the  
589 patterns  $P_1$  and  $P_2$ . Our goal is to design an operation that returns a mapping with all variable  
590 bindings for variables `?x` and `?y` from pattern  $P_1$  but excluding variable bindings from pattern  $P_2$ .  
591 The challenge is that it does not suffices simply discarding the variable `?z`, which occurs only in  
592 pattern  $P_2$ , but also considering the variable `?y`, which is unbound in pattern  $P_1$ . If instead of  
593 `OPTIONAL $_{C_v}$` , the rule for `DIFF` (rule 5) used `OPTIONAL`, the query would return the provenance  
594 for the mapping  $\{?x \mapsto a, ?y \mapsto b\}$  instead of the mapping  $\mu_1$ . That we would returned a value  
595 for variable `?y` that is bound in pattern  $P_2$  but not in pattern  $P_1$ . Instead, if `?x` is strongly bound  
596 for  $P_1$  (i.e., always bound in its answers) and `?y` is not, then the operation  $P_1 \text{ OPTIONAL}_{C_v} P_2$  is:

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

598 where  $?u = \nu(?y)$  is a fresh variable introduced by the variable renaming function  $\nu : \mathbf{V} \rightarrow \mathbf{V}$ . The  
599 result of this operation for the aforementioned mappings is the mapping  $\{?x \mapsto a, ?u \mapsto b, ?z \mapsto c\}$ .

600 Since variables  $?u$  and  $?z$  are not in-scope variables of pattern  $P_1$ , can be discarded to obtain the  
 601 actual answer  $\mu = \{?x \mapsto a\}$  of the query  $(Q_1 \text{ DIFF } Q_2)$ .

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

### 607 4.3 Correctness criteria

608 To define correctness of the query rewriting described in Definition 28 we need the notions of  
 609 *soundness* and *completeness*. A query rewriting is sound when the rewritten query returns right  
 610 polynomial expressions, and complete if it returns polynomial expressions for all mappings with  
 611 non-zero polynomials.

612 ► **Definition 32** (Soundness and Completeness). *Let Reify be a function that implements a reification*  
 613 *scheme, and  $\gamma$  be a function that receives a SPARQL query  $Q$  and a variable  $?prov \notin \text{inScope}(Q)$ ,*  
 614 *and returns a SPARQL query  $\gamma(Q, ?prov)$  with  $\text{inScope}(\gamma(Q, ?prov)) = \text{inScope}(Q) \cup \{?prov\}$ .*  
 615 *Let  $\Gamma$  be an X-graph, and  $G = \text{Reify}(\Gamma)$  the RDF-star graph resulting from applying the Reify*  
 616 *function to each triple in  $\Gamma$  in order to encode  $\Gamma$ 's triple annotations. Then:*

- 617 ■  $\gamma$  is called *sound* for Reify if, for every answer of the rewritten query  $\mu \cup \{?prov \mapsto e\}$  in  
 618  $\llbracket \gamma(Q, ?prov) \rrbracket_G$ ,  $e$  is an expression for the polynomial  $\llbracket Q \rrbracket_\Gamma(\mu)$ .
- 619 ■  $\gamma$  is called *complete* for Reify if, for every mapping  $\mu$  such that  $\llbracket Q \rrbracket_\Gamma(\mu)$  is a non-zero polynomial,  
 620 there exists an expression  $e$  such that  $\mu \cup \{?prov \mapsto e\} \in \llbracket \gamma(Q, ?prov) \rrbracket_G$ .

621 ► **Theorem 33.** *Let Reify be a reification scheme and  $\beta$  be the function described in Definition 28.*  
 622 *Then, function  $\beta$  is sound and complete for the reification scheme Reify.*

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

635 We next consider the non-base cases:

636 *Case  $Q = (Q_1 \text{ AND } Q_2)$ .* For every mapping  $\mu$  and annotated graph  $\Gamma$ ,

$$637 \quad \llbracket (Q_1 \text{ AND } Q_2) \rrbracket_\Gamma(\mu) = \sum_{\mu = \mu_1 \cup \mu_2} (\llbracket Q_1 \rrbracket_\Gamma(\mu_1) \otimes \llbracket Q_2 \rrbracket_\Gamma(\mu_2)).$$

638 Similarly for a corresponding regular graph  $G$  for the annotated graph  $\Gamma$ , the set  $\llbracket \beta(Q, ?prov) \rrbracket_G$   
 639 has the form  $\llbracket \mu'_1 \cup \{?prov \mapsto k_1\}, \dots, \mu'_m \cup \{?prov \mapsto k_m\} \rrbracket$ , where  $\mu'_1, \dots, \mu'_m$  are distinct  
 640 mappings. By construction, for  $1 \leq j \leq m$ ,  $k_j$  has the form  $(\oplus (\otimes k_1^1 k_2^1) \dots (\otimes k_1^p k_2^p))$ , and for  
 641  $1 \leq i \leq p$ , there exists two mappings  $\mu_1^i$  and  $\mu_2^i$  such that

$$642 \quad \mu_1^i \cup \{?prov\oplus \mapsto k_1^i\} \in \llbracket \beta(Q_1, ?prov\oplus) \rrbracket_G,$$

#### 4:18 Native Provenance Computation for Federated and Non-Federated SPARQL Queries

643 ■  $\mu_2^i \cup \{\text{?prov} \oplus 2 \mapsto k_2^i\} \in \llbracket \beta(Q_2, \text{?prov} \oplus 2) \rrbracket_G$ .

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

649 To prove the completeness, we need to show that every mapping  $\mu$  in the support of  $\llbracket (Q) \rrbracket_\Gamma$  is one of  
650 the aforementioned mappings  $\mu_1', \dots, \mu_m'$ . Since  $\mu \in \text{supp}(\llbracket (Q) \rrbracket_\Gamma)$ , there exist at least two mappings  
651  $\mu_1 \in \text{supp}(\llbracket (Q_1) \rrbracket_\Gamma)$  and  $\mu_2 \in \text{supp}(\llbracket (Q_2) \rrbracket_\Gamma)$  such that  $\mu = \mu_1 \cup \mu_2$ . By the induction hypothesis, these  
652 mappings appear in the solutions of queries  $\beta(Q_1, \text{?prov} \oplus 1)$  and  $\beta(Q_2, \text{?prov} \oplus 2)$ , respectively.  
653 Then,  $\mu$  appears in the solutions of query  $(\beta(Q_1, \text{?prov} \oplus 1) \text{ AND } \beta(Q_2, \text{?prov} \oplus 2))$ . Then,  $\mu$   
654 appears in the solutions of query  $\beta(Q, \text{?prov})$ . Hence, the rewriting is complete.

655 *Case  $Q = (Q_1 \text{ UNION } Q_2)$ .* For every mapping  $\mu$  and annotated graph  $\Gamma$ ,

$$656 \quad \llbracket (Q_1 \text{ UNION } Q_2) \rrbracket_\Gamma(\mu) = \llbracket (Q_1) \rrbracket_\Gamma(\mu) \oplus \llbracket (Q_2) \rrbracket_\Gamma(\mu).$$

657 Similarly for a corresponding regular graph  $G$  for the annotated graph  $\Gamma$ , the set  $\llbracket \beta(Q, \text{?prov}) \rrbracket_G$   
658 has the form  $\{\mu_1' \cup \{\text{?prov} \mapsto r_1\}, \dots, \mu_m' \cup \{\text{?prov} \mapsto k_m\}\}$ , where  $\mu_1', \dots, \mu_m'$  are distinct  
659 mappings. By construction, for  $1 \leq j \leq m$ ,  $k_j$  has either the form  $(\oplus r_1 r_2)$ ,  $(\oplus r_1)$ , or  $(\oplus r_2)$ ,  
660 and

661 ■  $\mu_j' \cup \{\text{?prov} \oplus \mapsto r_1\} \in \llbracket \beta(Q_1, \text{?prov} \oplus) \rrbracket_G$  if and only if  $k_j = (\oplus r_1 r_2)$  or  $k_j = (\oplus r_1)$ ,

662 ■  $\mu_j' \cup \{\text{?prov} \oplus \mapsto r_2\} \in \llbracket \beta(Q_2, \text{?prov} \oplus) \rrbracket_G$  if and only if  $k_j = (\oplus r_1 r_2)$  or  $k_j = (\oplus r_2)$ .

663 By the induction hypothesis, the query rewriting is sound and complete for the queries  $Q_1$  and  
664  $Q_2$ . Thus, if defined,  $r_1$  and  $r_2$  represent  $\llbracket (Q_1) \rrbracket_\Gamma(\mu_j')$  and  $\llbracket (Q_2) \rrbracket_\Gamma(\mu_j')$ , respectively. If  $r_1$  or  $r_2$  is  
665 undefined, the respective value of  $\llbracket (Q_1) \rrbracket_\Gamma(\mu_j')$  or  $\llbracket (Q_2) \rrbracket_\Gamma(\mu_j')$  is 0. Hence, the rewriting is sound.

666 To prove the completeness, we need to show that every mapping  $\mu$  in the support of  $\llbracket (Q) \rrbracket_\Gamma$   
667 is one of the aforementioned mappings  $\mu_1', \dots, \mu_m'$ . Since  $\mu \in \text{supp}(\llbracket (Q) \rrbracket_\Gamma)$ , it must happen  
668 that  $\mu \in \text{supp}(\llbracket (Q_1) \rrbracket_\Gamma)$  or  $\mu \in \text{supp}(\llbracket (Q_2) \rrbracket_\Gamma)$ . By the induction hypothesis,  $\mu$  appears in the  
669 solutions of queries  $\beta(Q_1, \text{?prov} \oplus)$  or  $\beta(Q_2, \text{?prov} \oplus)$ . Then,  $\mu$  appears in the solutions of query  
670  $(\beta(Q_1, \text{?prov} \oplus) \text{ UNION } \beta(Q_2, \text{?prov} \oplus))$ . Then,  $\mu$  appears in the solutions of query  $\beta(Q, \text{?prov})$ .  
671 Hence, the rewriting is complete.

672 *Case  $Q = (Q_1 \text{ DIFF } Q_2)$ .* For every mapping  $\mu$  and annotated graph  $\Gamma$ ,

$$673 \quad \llbracket (Q_1 \text{ DIFF } Q_2) \rrbracket_\Gamma(\mu) = \llbracket (Q_1) \rrbracket_\Gamma(\mu) \ominus \sum_{\mu \sim \mu_2} (\llbracket (Q_2) \rrbracket_\Gamma(\mu_2)).$$

674 Similarly for a corresponding regular graph  $G$  for the annotated graph  $\Gamma$ , the set  $\llbracket \beta(Q, \text{?prov}) \rrbracket_G$   
675 has the form  $\{\mu_1' \cup \{\text{?prov} \mapsto k_1\}, \dots, \mu_m' \cup \{\text{?prov} \mapsto k_m\}\}$ , where  $\mu_1', \dots, \mu_m'$  are distinct  
676 mappings. By construction, for  $1 \leq j \leq m$ ,  $k_j$  has the form  $(\ominus k (\oplus k_1 \cdots k_p))$ , and

677 ■  $\mu_j' \cup \{\text{?prov} \ominus 1 \mapsto k\} \in \llbracket \beta(Q_1, \text{?prov} \ominus 1) \rrbracket_G$ ,

678 ■ for  $1 \leq i \leq p$ , there is a mapping  $\mu_i$ , such that  $\mu_i \cup \{\text{?prov} \ominus 2 \oplus \mapsto k_i\} \in \llbracket \beta(Q_2, \text{?prov} \ominus 2 \oplus) \rrbracket_G$   
679 and  $\mu_i \sim \mu_j'$ .

680 By the induction hypothesis, the query rewriting is sound and complete for the queries  $Q_1$  and  
681  $Q_2$ . Thus,  $k$  and  $(\oplus k_1 \dots k_p)$  represent  $\llbracket (Q_1) \rrbracket_\Gamma(\mu_j')$  and  $\sum_{\mu_i \sim \mu_j'} \llbracket (Q_2) \rrbracket_\Gamma(\mu_i)$ . Hence, the rewriting is  
682 sound.

683 To prove the completeness, we need to show that every mapping  $\mu$  in the support of  $\llbracket (Q) \rrbracket_\Gamma$  is  
684 one of the aforementioned mappings  $\mu_1', \dots, \mu_m'$ . Since  $\mu \in \text{supp}(\llbracket (Q) \rrbracket_\Gamma)$ , also  $\mu \in \text{supp}(\llbracket (Q_1) \rrbracket_\Gamma)$ . By

685 the induction hypothesis,  $\mu$  appears in the solutions of queries  $\beta(Q_1, ?\text{prov}\ominus 1)$ . Then,  $\mu$  appears  
 686 in the solutions of query  $\beta(Q, ?\text{prov})$ . Hence, the rewriting is complete.

687 *Case*  $Q = (\text{SELECT } W \text{ WHERE } Q_1)$ . For every mapping  $\mu$  and annotated graph  $\Gamma$ ,

$$688 \quad \llbracket (\text{SELECT } W \text{ WHERE } Q_1) \rrbracket_{\Gamma}(\mu) = \sum_{\mu': \mu'|_W = \mu} \llbracket Q_1 \rrbracket_{\Gamma}(\mu').$$

689 Similarly for a corresponding regular graph  $G$  for the annotated graph  $\Gamma$ , the set  $\llbracket \beta(Q, ?\text{prov}) \rrbracket_G$   
 690 has the form  $\{\mu'_1 \cup \{?\text{prov} \mapsto k_1\}, \dots, \mu'_m \cup \{?\text{prov} \mapsto k_m\}\}$ , where  $\mu'_1, \dots, \mu'_m$  are distinct  
 691 mappings. By construction, for  $1 \leq j \leq m$ ,  $k_j$  has the form  $(\oplus k_1 \dots k_p)$ , and for  $1 \leq i \leq p$ , there  
 692 exists a mapping  $\mu_i$  such that  $\mu_i|_W = \mu'_j$  and  $\mu_i \cup \{?\text{prov}\oplus \mapsto k_i\} \in \llbracket \beta(Q_1, ?\text{prov}\oplus) \rrbracket_G$ . By the  
 693 induction hypothesis, the query rewriting is sound and complete for query  $Q_1$ . Thus, each  $k_i$   
 694 represents  $\llbracket Q_1 \rrbracket_{\Gamma}(\mu_i)$ . Hence, the rewriting is sound.

695 To prove the completeness, we need to show that every mapping  $\mu$  in the support of  $\llbracket Q \rrbracket_{\Gamma}$  is  
 696 one of the aforementioned mappings  $\mu'_1, \dots, \mu'_m$ . Since  $\mu \in \text{supp}(\llbracket Q \rrbracket_{\Gamma})$ , there exists at least one  
 697 mapping  $\mu_1 \in \text{supp}(\llbracket Q_1 \rrbracket_{\Gamma})$  such that  $\mu = \mu_1|_W$ . By the induction hypothesis, mapping  $\mu_1$  appears  
 698 in the solutions of query  $\beta(Q_1, ?\text{prov}\oplus)$ . Then,  $\mu$  appears in the solutions of query  $\beta(Q, ?\text{prov})$ .  
 699 Hence, the rewriting is complete.

700 *Case*  $Q = (Q_1 \text{ FILTER } \varphi)$ . For every mapping  $\mu$  and annotated graph  $\Gamma$ ,

$$701 \quad \llbracket (Q_1 \text{ FILTER } \varphi) \rrbracket_{\Gamma}(\mu) = \llbracket Q_1 \rrbracket_{\Gamma}(\mu) \otimes 1_{\mu \models \varphi}.$$

702 Similarly for a corresponding regular graph  $G$  for the annotated graph  $\Gamma$ , the set  $\llbracket \beta(Q, ?\text{prov}) \rrbracket_G$   
 703 has the form  $\{\mu'_1 \cup \{?\text{prov} \mapsto k_1\}, \dots, \mu'_m \cup \{?\text{prov} \mapsto k_m\}\}$ , where  $\mu'_1, \dots, \mu'_m$  are distinct  
 704 mappings. By construction, for  $1 \leq j \leq m$ ,  $k_j$ ,  $\mu'_j \cup \{?\text{prov} \mapsto k_i\} \in \llbracket \beta(Q_1, ?\text{prov}) \rrbracket_G$  and  $\mu_j \models \varphi$ .  
 705 By the induction hypothesis, the query rewriting is sound for query  $Q_1$ . Thus,  $k_j$  represents the  
 706  $\llbracket Q_1 \rrbracket_{\Gamma}(\mu'_j)$ . Hence, the rewriting is sound.

707 To prove the completeness, we need to show that every mapping  $\mu$  in the support of  $\llbracket Q \rrbracket_{\Gamma}$   
 708 is one of the aforementioned mappings  $\mu'_1, \dots, \mu'_m$ . Since  $\mu \in \text{supp}(\llbracket Q \rrbracket_{\Gamma})$ ,  $\mu \in \text{supp}(\llbracket Q_1 \rrbracket_{\Gamma})$  and  
 709  $\mu \models \varphi$ . By the induction hypothesis, mapping  $\mu$  appears in the solutions of query  $\beta(Q_1, ?\text{prov})$ .  
 710 Since  $\mu \models \varphi$ ,  $\mu$  appears in the solutions of query  $(\beta(Q_1, ?\text{prov}) \text{ FILTER } \varphi)$ , which are the solutions  
 711 of query  $\beta(Q, ?\text{prov})$ . Hence, the rewriting is complete.

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

714

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

## 719 4.4 Query Rewriting Optimizations

720 If we look at our example rewritten query  $Q'$  described in Section 4.1, we can notice that this  
 721 query includes a GROUP BY clause, and two subqueries, namely  $P'_1$  and  $P'_2$ , each of which also  
 722 includes a GROUP BY clause. In Definition 35 we describe an alternative query rewriting that  
 723 produces equivalent polynomial expressions, but reduces the number of aggregate operations.

724 ► **Definition 34.** A sum-query  $Q$  is a query such that the query rewriting  $\beta$  described in Defini-  
725 tion 28 returns a query of the form:

726 
$$\beta(Q, ?\text{prov}) = (\text{SELECT} \quad \text{inScope}(Q) (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov})$$

$$\text{WHERE} \quad T$$

$$\text{GROUP BY} \quad \text{inScope}(Q)).$$

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

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

730 ► **Definition 35.** Let  $Q$  be a SPARQL query,  $?\text{prov}$  be a variable, and  $\text{Reify}$  a reification scheme.  
731 Then, the rewritten query for  $Q$  and variable  $?\text{prov}$  over scheme  $\text{Reify}$ , denoted  $\beta(Q, ?\text{prov})$ , is  
732 defined recursively as is specified in Definition 28, but the following rules are applied when possible:

733 1. If  $Q$  is  $(Q_1 \text{ AND } \dots \text{ AND } Q_n)$ , and  $Q_1, \dots, Q_n$  are sum-queries, such that for  $1 \leq i \leq n$ , the  
734 pattern of  $\beta(Q_i, ?\text{prov}\oplus i)$  is  $T_i$ , then  $\beta(Q, ?\text{prov})$  is the query

735 
$$(\text{SELECT} \quad \text{inScope}(Q) (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov})$$

$$\text{WHERE} \quad ((T_1 \text{ AND } \dots \text{ AND } T_n)$$

$$\text{BIND} (\text{ProvProd}(\text{?prov}\oplus 1, \dots, \text{?prov}\oplus n) \text{ AS } \text{?prov}\oplus)$$

$$\text{GROUP BY} \quad \text{inScope}(Q)).$$

736 2. If  $Q$  is  $(Q_1 \text{ UNION } \dots \text{ UNION } Q_n)$ , where  $Q_1, \dots, Q_n$  are sum-queries, and for  $1 \leq i \leq n$ , the  
737 pattern of  $\beta(Q_i, ?\text{prov}\oplus)$  is  $T_i$ , then  $\beta(Q, ?\text{prov})$  is the query

738 
$$(\text{SELECT} \quad \text{inScope}(Q) (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov})$$

$$\text{WHERE} \quad (T_i \text{ UNION } \dots \text{ UNION } T_n)$$

$$\text{GROUP BY} \quad \text{inScope}(Q)).$$

739 3. If  $Q$  is  $(Q_1 \text{ DIFF } Q_2)$ , and  $\nu$  is a variable substitution that substitutes with fresh variables the  
740 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,  
741 then  $\beta(Q, ?\text{prov})$  is the query

742 
$$(\text{SELECT} \quad \text{inScope}(Q) (\text{ProvDiff}(\text{?prov}\ominus 1, \text{ProvAggSum}(\text{?prov}\ominus 2\oplus)) \text{ AS } \text{?prov})$$

$$\text{WHERE} \quad (\beta(Q_1, ?\text{prov}\ominus 1) \text{ OPTIONAL}_{C\nu} \beta(\nu(Q_2), ?\text{prov}\ominus 2\oplus))$$

$$\text{GROUP BY} \quad \text{inScope}(Q) \cup \{\text{?prov}\ominus 1\}).$$

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

744 
$$(\text{SELECT} \quad W (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov})$$

$$\text{WHERE} \quad \beta(Q', ?\text{prov}\oplus)$$

$$\text{GROUP BY} \quad W).$$

745 ► **Note 36.** In the first two rules of Definition 35, we omitted the parenthesis for sequences of  
746 operations AND and UNION, because these operators are associative. Intuitively, the associativity  
747 of these operators allows considering the binary operation as a single variadic operation with a  
748 single GROUP BY clause.

749 ► **Example 37.** Consider the query  $Q$  from Example 15. Then, according to the query rewriting  
750 described in Definition 35 the rewritten query of  $Q$  is:

751 
$$\beta(Q, ?\text{prov}) = (\text{SELECT} \quad ?x (\text{ProvAggSum}(\text{?prov}\oplus) \text{ AS } \text{?prov})$$

$$\text{WHERE} \quad ((\text{Reify}(?x, \text{likes}, \text{pasta}, \text{?prov}\oplus 1) \text{ AND}$$

$$\text{Reify}(?x, \text{livesIn}, \text{Italy}, \text{?prov}\oplus 2))$$

$$\text{BIND} (\text{ProvProd}(\text{?prov}\oplus 1, \text{?prov}\oplus 2) \text{ AS } \text{?prov}\oplus))$$

$$\text{GROUP BY} \quad ?x).$$

752 This query has only one GROUP BY clause whereas the query  $Q'$  at the end of Section 4.1 (generated  
753 using the rewriting of Definition 28) has three.

754 ► **Theorem 38.** *Let Reify be a reification scheme, and  $\beta$  be the function described in Definition 35.*  
755 *Then, function  $\beta$  is sound and complete for the reification scheme Reify.*

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

759 *Case  $Q = (Q_1 \text{ AND } \dots \text{ AND } Q_n)$ .* It suffices to prove this case by induction on  $n$ . For the base  
760 case  $n = 2$ , both query rewritings produce the same query. For the inductive case, we abbreviate  
761  $(Q_1 \text{ AND } \dots \text{ AND } Q_{n-1})$  as  $Q_{1,n-1}$ . Let  $Q'_1$  be  $\beta(((Q_{1,n-1} \text{ AND } Q_n) \text{ AND } Q_{n+1}), ?\text{prov})$  according  
762 to Definition 35, and  $Q'_2$  be  $\beta((Q_{1,n-1} \text{ AND } Q_n \text{ AND } Q_{n+1}), ?\text{prov})$  according to Definition 35. It  
763 is not difficult to see that every mapping  $\mu$  that appears in the answers to query  $Q'_1$  appears  
764 in the answers to query  $Q'_2$ , and vice versa. Assume that  $\mu \cup \{?\text{prov} \mapsto k_1\} \in \llbracket Q'_1 \rrbracket_G$  and  
765  $\mu \cup \{?\text{prov} \mapsto k_2\} \in \llbracket Q'_2 \rrbracket_G$ . Then, we need to prove that the expressions  $k_1$  and  $k_2$  are equivalent.  
766 Let  $\Omega_a, \Omega_b$ , and  $\Omega_c$  be the respective set of mappings that appear in the support of  $\llbracket Q_{1,n-1} \rrbracket_G$ ,  
767  $\llbracket Q_n \rrbracket_G$ , and  $\llbracket Q_{n+1} \rrbracket_G$ . By construction,

$$768 \quad k_1 = \sum_{\substack{\mu_{ab} \in \Omega_a \bowtie \Omega_b \\ \mu_c \in \Omega_c \\ \mu_{ab} \sim \mu \\ \mu_c \sim \mu}} \left( \otimes \sum_{\substack{\mu_a \in \Omega_a \\ \mu_b \in \Omega_b \\ \mu_a \sim \mu_{ab} \\ \mu_b \sim \mu_{ab}}} (\otimes k_a k_b k_c), \quad k_2 = \sum_{\substack{\mu_a \in \Omega_a \\ \mu_b \in \Omega_b \\ \mu_c \in \Omega_c \\ \mu_a \sim \mu \\ \mu_b \sim \mu \\ \mu_c \sim \mu}} (\otimes k_a k_b k_c),$$

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

771 *Case  $Q = (Q_1 \text{ UNION } \dots \text{ UNION } Q_n)$ .* It suffices to prove this case by induction on  $n$ . For the base  
772 case  $n = 2$ , both query rewritings produce the same query. For the inductive case, we abbreviate  
773  $(Q_1 \text{ UNION } \dots \text{ UNION } Q_{n-1})$  as  $Q_{1,n-1}$ . Let  $Q'_1$  be  $\beta(((Q_{1,n-1} \text{ UNION } Q_n) \text{ UNION } Q_{n+1}), ?\text{prov})$   
774 according to Definition 35, and  $Q'_2$  be  $\beta((Q_{1,n-1} \text{ UNION } Q_n \text{ UNION } Q_{n+1}), ?\text{prov})$  according to  
775 Definition 35. It is not difficult to see that every mapping  $\mu$  that appears in the answers to query  
776  $Q'_1$  appears in the answers to query  $Q'_2$ , and vice versa. Assume that  $\mu \cup \{?\text{prov} \mapsto k_1\} \in \llbracket Q'_1 \rrbracket_G$   
777 and  $\mu \cup \{?\text{prov} \mapsto k_2\} \in \llbracket Q'_2 \rrbracket_G$ . Then, we need to prove that the expressions  $k_1$  and  $k_2$  are  
778 equivalent. Let  $\Omega_a, \Omega_b$ , and  $\Omega_c$  be the respective set of mappings that appear in the support of  
779  $\llbracket Q_{1,n-1} \rrbracket_G$ ,  $\llbracket Q_n \rrbracket_G$ , and  $\llbracket Q_{n+1} \rrbracket_G$ . By construction,

$$780 \quad k_1 = (\oplus (\oplus k_a k_b k_c)), \quad k_2 = (\oplus k_a k_b k_c),$$

781 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  
782 does not necessarily appears in the expression. Hence,  $k_1$  and  $k_2$  are equivalent.

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

$$788 \quad k_1 = (\ominus k (\oplus (\oplus a_1) \dots (\oplus a_n))), \quad k_2 = (\ominus k (\oplus a_1 \dots a_n)),$$

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

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

$$797 \quad k_1 = (\oplus (\oplus a_1) \dots (\oplus a_n)), \quad k_2 = (\oplus a_1 \dots a_n),$$

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

800

## 801 **5 How-provenance of federated SPARQL query computation**

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

### 810 **5.1 An algebra for Federated Query Computation**

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

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

$$823 \quad G_1 = \{((\text{Ulm}, \text{a}, \text{City}), \text{wasDerivedFrom}, x_1), \\
 ((\text{Ulm}, \text{country}, \text{Germany}), \text{wasDerivedFrom}, x_2), \\
 ((\text{Danube}, \text{crosses}, \text{Ulm}), \text{wasDerivedFrom}, x_3), \\
 ((\text{Budapest}, \text{a}, \text{City}), \text{wasDerivedFrom}, x_4), \\
 ((\text{Budapest}, \text{country}, \text{Hungary}), \text{wasDerivedFrom}, x_5)\}. \\
 G_2 = \{((\text{Danube}, \text{crosses}, \text{Ulm}), \text{wasDerivedFrom}, x_3), \\
 ((\text{Danube}, \text{crosses}, \text{Budapest}), \text{wasDerivedFrom}, x_6)\}.$$

824 Graphs  $G_1$  and  $G_2$  can share statements and identifiers. For example, “The Danube crosses  
825 Ulm” is stated in both graphs, and annotated with the IRI  $x_3$ . To use the non-federated NPC  
826 framework we should first merge these two graphs into one. By naming the graph  $G_1$  as  $y_1$  and  
827 the graph  $G_2$  as  $y_2$ , we can define the following RDF-star graph as the merge of both graphs:

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

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

832  $\Gamma = \{(\text{Ulm}, \text{a}, \text{City}) \mapsto (y_1, x_1),$   
 $(\text{Ulm}, \text{country}, \text{Germany}) \mapsto (y_1, x_2),$   
 $(\text{Danube}, \text{crosses}, \text{Ulm}) \mapsto (y_1, x_3),$   
 $(\text{Danube}, \text{crosses}, \text{Ulm}) \mapsto (y_2, x_3),$   
 $(\text{Budapest}, \text{a}, \text{City}) \mapsto (y_1, x_4),$   
 $(\text{Budapest}, \text{country}, \text{Hungary}) \mapsto (y_1, x_5),$   
 $(\text{Danube}, \text{crosses}, \text{Budapest}) \mapsto (y_2, x_6)\}.$

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

834  $(\text{SELECT } \{?\text{country}\}$   
 $\text{WHERE } ((?\text{city}, \text{a}, \text{City}) \text{ AND } (?\text{city}, \text{country}, ?\text{country}) \text{ AND } (\text{Danube}, \text{crosses}, ?\text{city})).$

835 By executing the query  $Q_O$  on graph  $\Gamma$ , the answers Germany and Hungary are explained by the  
836 following how-provenance polynomials:

837  $(Q_O)_\Gamma(\{?\text{country} \mapsto \text{Germany}\}) = (y_1, x_1) \otimes (y_1, x_2) \otimes ((y_1, x_3) \oplus (y_2, x_3)),$   
838  $(Q_O)_\Gamma(\{?\text{country} \mapsto \text{Hungary}\}) = (y_1, x_4) \otimes (y_1, x_5) \otimes (y_2, x_6).$

839 As described in Section 4, the how provenance of these solution mappings can be computed with  
840 NPC by rewriting the query  $Q_O$  into a query  $Q_R$ , and then executing query  $Q_R$  on the graph  $\Gamma$ .

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

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

- 852 ■  $f(0_{\mathcal{M}}, k) = 0$  and  $(m, 0_{\mathcal{K}}) = 0$  for every  $k \in K$  and  $m \in M$ .  
853 ■  $f(1_{\mathcal{M}}, 1_{\mathcal{K}}) = 1$ .  
854 ■ For every  $m \in M$  and  $k_1, k_2 \in K$ ,  
855 ■  $f(m, k_1 +_{\mathcal{K}} k_2) = f(m, k_1) \oplus f(m, k_2)$ .  
856 ■  $f(m, k_1 \times_{\mathcal{K}} k_2) = f(m, k_1) \otimes f(m, k_2)$ .  
857 ■  $f(m, k_1 -_{\mathcal{K}} k_2) = f(m, k_1) \ominus f(m, k_2)$ .  
858 ■ For every  $m_1, m_2 \in M$  and  $k \in K$ ,  $f(m_1 \oplus m_2, k) = f(m_1, k) \oplus f(m_2, k)$   
859 ■ For every element  $r \in R$  there is a finite set  $\{(m_1, k_1), \dots, (m_n, k_n)\} \subseteq M \times K$  such that  
860  $r = f(m_1, k_1) \oplus \dots \oplus f(m_n, k_n)$ .  
861  $\mathcal{M}$  is called the fed-structure and  $\mathcal{K}$  is called the how-structure of the federated spm-semiring  $R$ .

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

- 865 ■  $0 \in \text{FedProvExp}(X, Y)$ ,  $1 \in \text{FedProvExp}(X, Y)$ , and  $X \subseteq \text{FedProvExp}(X, Y)$ .  
866 ■ We call service-expressions to the expressions combining elements in set  $Y \cup \{0, 1\}$  with the  
867 operators  $\oplus$  and  $\otimes$ . If  $a \in \text{FedProvExp}(X, Y)$  and  $s$  is a service-expression, then  $s \otimes a \in$   
868  $\text{FedProvExp}(X, Y)$ .  
869 ■ If  $a, b \in \text{FedProvExp}(X, Y)$  then  $a \oplus b$ ,  $a \otimes b$ , and  $a \ominus b$  are in  $\text{FedProvExp}(X, Y)$ .

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

- 873 ■  $h(0) = 0$  and  $h(1) = 1$ .  
874 ■ If  $x \in X$  then  $h(x) = h_X(x)$ .  
875 ■ If  $y \in Y$  then  $h(y) = h_Y(y)$ .  
876 ■ 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  $h(s_1 \otimes s_2) =$   
877  $h_Y(s_1) \otimes h_Y(s_2)$ .  
878 ■ If  $a$  and  $b$  are federated how-provenance expressions, then  $h(a \oplus b) = h_Y(a) \oplus h_Y(b)$ ,  $h(a \otimes b) =$   
879  $h_Y(a) \otimes h_Y(b)$ , and  $h(a \ominus b) = h_Y(a) \ominus h_Y(b)$ .

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

883 
$$\llbracket Q_O \rrbracket_{\Gamma}(\{\text{?country} \mapsto \text{Germany}\}) = (y_1 \otimes x_1) \otimes (y_1 \otimes x_2) \otimes ((y_1 \otimes x_3) \oplus (y_2 \otimes x_3)),$$
  
884 
$$\llbracket Q_O \rrbracket_{\Gamma}(\{\text{?country} \mapsto \text{Hungary}\}) = (y_1 \otimes x_4) \otimes (y_1 \otimes x_5) \otimes (y_2 \otimes x_6).$$

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

## 889 5.2 Annotated Answers for Federated SPARQL queries

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

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

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

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

$$\begin{aligned} 901 \quad \Gamma_1 = \{ & \llbracket (\text{Ulm}, \text{a}, \text{City}) \mapsto x_1, \\ & (\text{Ulm}, \text{country}, \text{Germany}) \mapsto x_2, \\ & (\text{Danube}, \text{crosses}, \text{Ulm}) \mapsto x_3, \\ & (\text{Budapest}, \text{a}, \text{City}) \mapsto x_4, \\ & (\text{Budapest}, \text{country}, \text{Hungary}) \mapsto x_5 \rrbracket, \\ \Gamma_2 = \{ & \llbracket (\text{Danube}, \text{crosses}, \text{Ulm}) \mapsto x_3, \\ & (\text{Danube}, \text{crosses}, \text{Budapest}) \mapsto x_6 \rrbracket. \end{aligned}$$

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

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

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

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

$$\begin{aligned} 919 \quad \blacksquare \quad \llbracket (s, p, o) \rrbracket_{\Delta, y}(\mu) &= \Delta(y)(\mu(s, p, o)), \\ 920 \quad \blacksquare \quad \llbracket (\text{SELECT } W \text{ WHERE } Q) \rrbracket_{\Delta, y}(\mu) &= \bigoplus_{\mu': \mu'|_W = \mu} \llbracket Q \rrbracket_{\Delta, y}(\mu'), \\ 921 \quad \blacksquare \quad \llbracket (Q \text{ FILTER } \varphi) \rrbracket_{\Delta, y}(\mu) &= \llbracket Q \rrbracket_{\Delta, y}(\mu) \otimes 1_{\mu \models \varphi}, \\ 922 \quad \blacksquare \quad \llbracket (Q_1 \text{ UNION } Q_2) \rrbracket_{\Delta, y}(\mu) &= \llbracket Q_1 \rrbracket_{\Delta, y}(\mu) \oplus \llbracket Q_2 \rrbracket_{\Delta, y}(\mu), \\ 923 \quad \blacksquare \quad \llbracket (Q_1 \text{ AND } Q_2) \rrbracket_{\Delta, y}(\mu) &= \bigoplus_{\mu = \mu_1 \cup \mu_2} (\llbracket Q_1 \rrbracket_{\Delta, y}(\mu_1) \otimes \llbracket Q_2 \rrbracket_{\Delta, y}(\mu_2)), \\ 924 \quad \blacksquare \quad \llbracket (Q_1 \text{ DIFF } Q_2) \rrbracket_{\Delta, y}(\mu) &= \llbracket Q_1 \rrbracket_{\Delta, y}(\mu) \ominus (\bigoplus_{\mu' \sim \mu} \llbracket Q_2 \rrbracket_{\Delta, y}(\mu')), \\ 925 \quad \blacksquare \quad \llbracket (Q_1 \text{ OPTIONAL } Q_2) \rrbracket_{\Delta, y}(\mu) &= \llbracket Q_1 \text{ AND } Q_2 \rrbracket_{\Delta, y}(\mu) \oplus \llbracket Q_1 \text{ DIFF } Q_2 \rrbracket_{\Delta, y}(\mu), \\ 926 \quad \blacksquare \quad \llbracket (\text{SERVICE } y' Q) \rrbracket_{\Delta, y}(\mu) &= y' \circledast \llbracket Q \rrbracket_{\Delta, y'}(\mu), \end{aligned}$$

927 where  $\bigoplus$  denotes sums using the operation  $\oplus$ ,  $\sim$  denotes mapping compatibility, and  $\mu'|_W$  is  
 928 the projection of mapping  $\mu'$  on the variables in  $W$ . Two mappings  $\mu$  and  $\mu'$  are compatible if  
 929  $\mu(?x) = \mu'(?x)$  for every variable  $?x \in \text{dom}(\mu) \cap \text{dom}(\mu')$ .

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

934 ► **Example 48.** Consider the annotated federated dataset  $\Delta$  from Example 45, and let  $Q$  be the  
935 query

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

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

939  $((?city, a, City))_{\Delta, y_1}(\mu) = \Delta(y_1)(\mu(?city, a, City)) = x_1.$

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

942 
$$\begin{aligned} ((\text{SERVICE } y_2 (\text{Danube, crosses, ?city}))_{\Delta, y_1}(\mu) &= y_2 \otimes ((\text{Danube, crosses, ?city}))_{\Delta, y_2}(\mu) \\ &= y_2 \otimes \Delta(y_2)(\mu(\text{Danube, crosses, ?city})) \\ &= y_2 \otimes x_3. \end{aligned}$$

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

### 944 5.3 How-Provenance for Federated Query Evaluation

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

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

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

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

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

962 
$$\begin{aligned} \text{FedGraph}(\Delta) = \{ &(\text{Ulm, a, City}) \mapsto y_1 \otimes x_1, \\ &(\text{Ulm, country, Germany}) \mapsto y_1 \otimes x_2, \\ &(\text{Danube, crosses, Ulm}) \mapsto (y_1 \otimes x_3) \oplus (y_2 \otimes x_3), \\ &(\text{Budapest, a, City}) \mapsto y_1 \otimes x_4, \\ &(\text{Budapest, country, Hungary}) \mapsto y_1 \otimes x_5, \\ &(\text{Danube, crosses, Budapest}) \mapsto y_2 \otimes x_6 \}. \end{aligned}$$

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

$$966 \quad ((\text{SERVICE } y_1 \ t) \text{ UNION } \dots \text{ UNION } (\text{SERVICE } y_n \ t)).$$

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

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

$$973 \quad \llbracket Q \rrbracket_{\text{FedGraph}(\Delta)}(\mu) = \llbracket \text{FedQuery}_\Delta(Q) \rrbracket_{\Delta, y}(\mu).$$

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

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

$$977 \quad Q = ((?city, a, City) \text{ AND } (\text{Danube, crosses, ?city})),$$

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

$$980 \quad \llbracket Q \rrbracket_{\text{FedGraph}(\Delta)}(\mu) = (y_1 \otimes x_1) \otimes ((y_1 \otimes x_3) \oplus (y_2 \otimes x_3)).$$

981 Alternatively, the federated query for  $Q$  is:

$$982 \quad \text{FedQuery}_Q(\Delta) = (((\text{SERVICE } y_1 \ (?city, a, City)) \text{ UNION} \\
 (\text{SERVICE } y_2 \ (?city, a, City))) \text{ AND} \\
 ((\text{SERVICE } y_1 \ (\text{Danube, crosses, ?city})) \text{ UNION} \\
 (\text{SERVICE } y_2 \ (\text{Danube, crosses, ?city}))))).$$

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

$$984 \quad \llbracket \text{FedQuery}_Q(\Delta) \rrbracket_{\Delta}(\mu) = (y_1 \otimes x_1) \otimes ((y_1 \otimes x_3) \oplus (y_2 \otimes x_3)).$$

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

## 991 5.4 Federated Provenance Computation

992 We have all the fundamentals to extend NPCS for federated provenance computation. By using  
 993 a query rewriting, NPCS employs a middleware that takes a SPARQL query and generates a  
 994 new query designed to retrieve provenance information. The resulting query seamlessly executes  
 995 on a singular instance of a SPARQL endpoint, yielding the desired results along with their

996 how-provenance algebraic expressions. Expanding upon this foundation, our extended approach,  
 997 *Fed-NPCS*, can be integrated into federation engines. Federation engines, such as FedX [42] and  
 998 FedUP [3], rewrite an *input query* into a set of subqueries that are executed in different endpoints  
 999 of the federation. To this end, these federated engines execute queries in two phases. First, in  
 1000 a planning phase, they execute queries or use summaries to elaborate a plan to retrieve data  
 1001 from the different endpoints. This plan includes the definition of subqueries, the endpoints where  
 1002 these subqueries will be evaluated, and the operations to combine the retrieved answers. Second,  
 1003 the engines execute the plans. Our approach, Fed-NPCS, translates the plan generated by these  
 1004 federated engines into a single SPARQL query, called the *federated plan query*. The federated plan  
 1005 query includes SERVICE clauses that encode the execution of the subqueries in the corresponding  
 1006 endpoints, and SPARQL operations to encode the combination of the answers from the subqueries.

1007 The process described so far does not include the federated how-provenance computation but  
 1008 the generation of the federated plan query that can compute the answers to the original query  
 1009 on the federation. To compute the how-provenance data, Fed-NPCS rewrites the federated plan  
 1010 query into a query called the *federated provenance query*. Like in NPCS, Fed-NPCS's federated  
 1011 provenance query is designed to retrieve provenance information.

1012 ► **Remark 55.** Unlike the NPCS system, which considers two queries, the Fed-NPCS considers  
 1013 three queries: (1) the input query ( $Q_O$ ) is the original query sent to NPCS; (2) the federated plan  
 1014 query ( $Q_F$ ) is the plan generated by the federation engine (e.g., FedX or FedUP) to compute the  
 1015 answers of the input query; and (3) the federated provenance query ( $Q_R$ ) is the query designed to  
 1016 retrieve provenance information on top of the federated query plan.

1017 ► **Remark 56.** Lemma 52 showed that the local query  $Q_O$  can be evaluated in a federation of  
 1018 annotated graphs  $\Delta$  by rewriting to the query  $\text{FedQuery}_\Delta(Q_O)$ . This query  $\text{FedQuery}_\Delta(Q_O)$  is  
 1019 equivalent to the query  $Q_F$  generated by a federation engine. The difference is that  $\text{FedQuery}_\Delta(Q_O)$   
 1020 is a simple query that satisfies Lemma 52, and query  $Q_F$  is optimized for a fast execution. Heling  
 1021 and Acosta [29] showed that naive decompositions as the one presented in this work are sound  
 1022 and complete for exclusive groups. For arbitrary decompositions, such guarantees are not proven.

### 1023 5.4.1 Federated Datasets

1024 The aforementioned annotated federated dataset  $\Delta$  provides a theoretical framework for the notion  
 1025 of how-provenance in the federated setting. In practice, federated engines do not operate over  
 1026 an annotated federated dataset  $\Delta$  consisting of annotated graphs  $\Gamma$ , but over regular federated  
 1027 datasets  $D$  consisting of regular graphs  $G$ . Thus, to compute the how-provenance, the input is a  
 1028 regular dataset  $D$  which encodes  $\Delta$ . Like NPCS, Fed-NPCS requires reification to encode each  
 1029 annotated graph  $\Gamma$  with a regular graph  $G$ . In this subsection we describe such an encoding.

1030 ► **Definition 57** (Encoding of the annotated federated dataset). *Let  $X$  and  $Y$  be two disjoint sets of*  
 1031 *IRIs,  $\Delta$  be an annotated federated dataset with set of service names  $Y$ , where for each  $y \in Y$ ,  $\Delta(y)$*   
 1032 *is an  $X$ -graph. Given a reification scheme  $\text{Reify}$ , the encoding of  $\Delta$  with the reification scheme*  
 1033  *$\text{Reify}$  is the federated dataset  $D$  that maps each IRI  $y \in Y$  to the RDF-star graph  $G$  defined as*  
 1034 *follows:*

$$1035 \quad G = \{\text{Reify}(t, \Gamma(t)) \mid t \in \text{supp}(\Gamma)\},$$

1036 where  $\Gamma$  is the  $X$ -graph  $\Delta(y)$ .

1037 ► **Example 58.** Consider the reification scheme  $\text{Reify}(t, x) = (t, \text{wasDerivedFrom}, x)$ , and the  
 1038 federated dataset  $\Delta$  described in Example 45. Then, the encoding of  $\Delta$  with the reification scheme  
 1039  $\text{Reify}$  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  
 1040 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  $\Gamma$ . The provenance polynomial expression is bound to an additional SPARQL variable `?prov` for each mapping  $\mu$  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  $\llbracket Q_R \rrbracket_G$  have the form  $\mu \cup \{\text{?prov} \mapsto k\}$ , where  $k$  is the how-provenance polynomial of  $\mu$  (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 dataset  $D$  have the form  $\mu \cup \{\text{?prov} \mapsto k\}$  and  $(Q_O)_\Delta(\mu) = k$ , where  $\Delta$  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  $Q_F = \text{FedQuery}_\Delta(Q_O)$  (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

$\text{FedQuery}_\Delta(Q_O)$  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  $\Delta$  includes three service names,  $y_1$ ,  $y_2$ , and  $y_3$ , then  $Q_F$  is:

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

## 4:30 Native Provenance Computation for Federated and Non-Federated SPARQL Queries

1083 If we are informed that service  $y_3$  has no answers for triple patterns  $T_1$  and  $T_2$ , and that services  
1084  $y_1$  and  $y_2$  have no answers for triple pattern  $T_3$ , then we can rewrite  $Q_F$  as follows:

$$1085 \quad Q'_F = (((\text{SERVICE } y_1 \ T_1) \ \text{UNION} \ (\text{SERVICE } y_2 \ T_1)) \ \text{AND} \\ ((\text{SERVICE } y_1 \ T_2) \ \text{UNION} \ (\text{SERVICE } y_2 \ T_2)) \ \text{AND} \\ (\text{SERVICE } y_3 \ T_3)).$$

1086 If we additionally know that each answer  $\mu_1$  of triple pattern  $T_1$  in  $y_1$  is incompatible with all the  
1087 answers of triple pattern  $T_2$  in service  $y_2$ , and similarly, each answer  $\mu_2$  to triple pattern  $T_1$  in  $y_2$   
1088 is incompatible with all the answers to triple pattern  $T_2$  in service  $y_1$ , then we can rewrite  $Q_F$  as  
1089 follows:

$$1090 \quad Q''_F = (((\text{SERVICE } y_1 \ (T_1 \ \text{AND} \ T_2))) \ \text{UNION} \\ ((\text{SERVICE } y_2 \ (T_1 \ \text{AND} \ T_2)))) \ \text{AND} \\ (\text{SERVICE } y_3 \ T_3)).$$

1091 Queries  $Q'_F$  and  $Q''_F$  require less time to be executed than query  $Q_F$  because they avoid unnecessary  
1092 pattern evaluation, and reduce network data transfer because of evaluating the joins in the service  
1093 for which the basic graph patterns are exclusive.

1094 ► **Remark 60.** The queries  $Q_F$ ,  $Q'_F$ , and  $Q''_F$  in Example 59 are not equivalent in general, but  
1095 under the information described in the example. Federated engines use availability and pattern  
1096 compatibility information across endpoints to generate efficient query plans. These efficient plans  
1097 discard services that are known to do not return answers to a triple pattern or pairs of services  
1098 that are known to do not produce joinable mappings for a given AND operation.

### 1099 Step 2: Query Rewriting

1100 After generating an intermediate query  $Q_F$ , which includes SERVICE operators, the next step is  
1101 to rewrite  $Q_F$  into a query  $Q_R$  that computes the provenance. The difference between a local  
1102 and a federated query provenance computing lies in the use of the SERVICE operator, which  
1103 directs portions of the query to be evaluated on external SPARQL endpoints. Our rewriting must  
1104 ensure that each part of the federated query can be rewritten to preserve both the provenance  
1105 of intermediate results and the aggregation of these results across sources. To define this query  
1106 rewriting we first define an auxiliary function that annotates the data from remote services with  
1107 the service name.

1108 ► **Definition 61.** Given a SPARQL variable  $?prov$  and an IRI  $y$ , the expression  
1109  $\text{ServiceOp}(y, ?prov)$  is defined as follows:

$$1110 \quad \text{ServiceOp}(y, ?prov) = \text{concat}("( \otimes ", y, ?prov, ")").$$

1111 Like the functions in Definition 24, the expression in Definition 61 defines a function to combine  
1112 polynomial expressions. In particular, the function  $\text{ServiceOp}$  combines a polynomial with the  
1113 service name using the service operator.

1114 ► **Definition 62** (Base Fed-NPCS query rewriting). Let  $Q_F$  be a SPARQL query,  $?prov$  a variable,  
1115 and Reify a reification scheme. Then, the rewritten query for  $Q_F$  and variable  $?prov$  over scheme  
1116 Reify, denoted  $\beta(Q_F, ?prov)$ , is defined recursively as follows:

1117 ■ If  $Q_F$  matches one of the rewriting rules in Definition 28 (i.e., the NPCCS query rewriting),  
1118 then  $\beta(Q_F, ?prov)$  is the result of applying that rule and then recursively applying rewriting  
1119 function  $\beta$ .

1120 ■ If  $Q_F$  has the form  $(\text{SERVICE } y \ Q)$  then  $\beta(Q_F, ?\text{prov})$  is the query

1121 
$$\begin{aligned} & (\text{SELECT } \text{inScope}(Q) \cup \{?\text{prov}\} \\ & \text{WHERE } ((\text{SERVICE } y \ \beta(Q, ?\text{prov} \otimes)) \text{ BIND } (\text{ServiceOp}(y, ?\text{prov} \otimes) \text{ AS } ?\text{prov}))). \end{aligned}$$

1122 ► **Example 63.** Consider the federated dataset  $D = \{y_1 \mapsto G_1, y_2 \mapsto G_2\}$  from Example 45, and  
1123 the query

1124 
$$Q_F = ((\text{SERVICE } y_1 \ (\text{Danube, crosses, ?city})) \text{ UNION } (\text{SERVICE } y_2 \ (\text{Danube, crosses, ?city}))).$$

1125 By the rule for the UNION operator (in Definition 28), the rewritten query for  $Q_F$  is

1126 
$$\begin{aligned} \beta(Q_F, ?\text{prov}) = & (\text{SELECT } \quad ?\text{city} \ (\text{ProvAggSum}(?\text{prov} \oplus) \text{ AS } ?\text{prov}) \\ & \text{WHERE } \quad (P_1 \text{ UNION } P_2) \\ & \text{GROUP BY } \quad ?\text{city}), \end{aligned}$$

1127 where, for  $i \in \{1, 2\}$ ,  $P_i$  is the query  $\beta((\text{SERVICE } y_i \ T), ?\text{prov} \oplus)$  and  $T$  is the triple pattern  
1128  $(\text{Danube, crosses, ?city})$ . These queries  $P_i$  are computed using the rule for the SERVICE operator:

1129 
$$\begin{aligned} P_i = & (\text{SELECT } \quad ?\text{city} \ ?\text{prov} \oplus \\ & \text{WHERE } \quad ((\text{SERVICE } y_i \ \beta(T, ?\text{prov} \oplus \otimes)) \text{ BIND } (\text{ServiceOp}(y_i, ?\text{prov} \oplus \otimes) \text{ AS } ?\text{prov} \oplus))). \end{aligned}$$

1130 The query  $\beta(T, ?\text{prov} \oplus \otimes)$  is defined by the rule for triple patterns

1131 
$$\begin{aligned} \beta(T, ?\text{prov} \oplus \otimes) = & (\text{SELECT } \quad ?\text{city} \ (\text{ProvAggSum}(?\text{prov} \oplus \otimes \oplus) \text{ AS } ?\text{prov} \oplus \otimes) \\ & \text{WHERE } \quad \text{Reify}((\text{Danube, crosses, ?city}), ?\text{prov} \oplus \otimes \oplus) \\ & \text{GROUP BY } \quad ?\text{city}). \end{aligned}$$

1132 Then, the query  $\beta(T, ?\text{prov} \oplus \otimes)$  returns the following answers in each service:

1133 
$$\begin{aligned} \llbracket \beta(T, ?\text{prov} \oplus \otimes) \rrbracket_{D, y_1} &= \left[ \frac{?\text{city} \mid ?\text{prov} \oplus \otimes}{\text{Ulm} \mid (\oplus (x_3))} \right], \\ \llbracket \beta(T, ?\text{prov} \oplus \otimes) \rrbracket_{D, y_2} &= \left[ \frac{?\text{city} \mid ?\text{prov} \oplus \otimes}{\begin{array}{l} \text{Ulm} \mid (\oplus (x_3)) \\ \text{Budapest} \mid (\oplus (x_6)) \end{array}} \right]. \end{aligned}$$

1134 Then, the results of evaluating queries  $P_1$  and  $P_2$  are:

1135 
$$\begin{aligned} \llbracket \beta(P_1, ?\text{prov} \oplus) \rrbracket_D &= \left[ \frac{?\text{city} \mid ?\text{prov} \oplus}{\text{Ulm} \mid (\otimes y_1 (\oplus (x_3)))} \right], \\ \llbracket \beta(P_2, ?\text{prov} \oplus) \rrbracket_D &= \left[ \frac{?\text{city} \mid ?\text{prov} \oplus}{\begin{array}{l} \text{Ulm} \mid (\otimes y_2 (\oplus (x_3))) \\ \text{Budapest} \mid (\otimes y_2 (\oplus (x_6))) \end{array}} \right]. \end{aligned}$$

1136 Combining these results we obtain:

1137 
$$\llbracket \beta(Q_F, ?\text{prov}) \rrbracket_D = \left[ \frac{?\text{city} \mid ?\text{prov}}{\begin{array}{l} \text{Ulm} \mid (\oplus (\otimes y_1 (\oplus (x_3))) (\otimes y_2 (\oplus (x_3)))) \\ \text{Budapest} \mid (\oplus (\otimes y_2 (\oplus (x_6)))) \end{array}} \right].$$

1138 Thus, the values for variable  $?\text{prov}$  for solutions where variable  $?\text{city}$  takes the values **Ulm** and  
1139 **Budapest** are the Polish notation expressions for the polynomials  $(y_1 \otimes x_3) \oplus (y_2 \otimes x_3)$  and  $y_2 \otimes x_6$ .

## 4:32 Native Provenance Computation for Federated and Non-Federated SPARQL Queries

1140 Like in NPCS (see Theorem 33), the correctness of the Fed-NPCS's query rewriting is based  
1141 on the underlying algebra of queries evaluated over annotated datasets.

1142 ► **Theorem 64.** *Let Reify be a reification scheme, and  $\beta$  be the function described in Definition 62.*  
1143 *Then, function  $\beta$  is sound and complete for the reification scheme Reify.*

1144 **Proof.** It can be shown by induction on the structure of the query. Excluding the rule with the  
1145 SERVICE operator, the proof is the same as for Theorem 33. Let us review the new case and  
1146 assume a federated dataset  $D$  for the corresponding annotated dataset  $\Delta$ . If a query  $Q_F$  has the  
1147 form (SERVICE  $y$   $Q$ ) then  $\beta(Q_F, ?prov)$  is the query:

1148 
$$\beta(Q_F, ?prov) = (\text{SELECT } \text{inScope}(Q) \cup \{?prov\}$$
  
WHERE ((SERVICE  $y$   $\beta(Q, ?prov\otimes)$ ) BIND (ServiceOp( $y, ?prov\otimes$ ) AS  $?prov$ ))).

1149 First, to show that  $\beta$  is sound, assume that mapping  $\mu \cup \{?prov \mapsto y \otimes k\}$  is an answer to  
1150 query  $\beta(Q_F, ?prov)$ . By construction, mapping  $\mu \cup \{?prov \mapsto k\}$  is then an answer to query  
1151  $\beta(Q, ?prov\otimes)$ . By induction,  $(Q)_{\Delta, y}(\mu) = k$ . By the semantics of the SERVICE operator (see  
1152 Definition 47),  $(Q_F)_{\Delta, y}(\mu) = y \otimes k$ . Hence,  $\beta$  is sound.

1153 Second, to show that  $\beta$  is complete, we use the inverse reasoning. By induction, the answers to  
1154 query  $\beta(Q, ?prov\otimes)$  include all mappings  $\mu \cup \{?prov\otimes \mapsto k\}$  such that  $(Q)_{\Delta, y}(\mu) = k$  and  $k \neq 0$ .  
1155 Since  $y \otimes 0 = 0$ , the answers to query  $\beta(Q_F, ?prov)$  include all solutions  $\mu \cup \{?prov \mapsto y \otimes k\}$   
1156 such that  $y \otimes k \neq 0$ . ◀

1157 So far, we have presented a sound and complete base query rewriting to compute provenance for  
1158 queries evaluated on federated SPARQL endpoints. However, this base query rewriting generates  
1159 some redundant operations that can be further simplified. Fed-NPCS applies the same techniques  
1160 used by NPCS and described in Section 4.4 for optimization.

1161 ► **Example 65.** Consider the query  $Q_F$  from Example 63. Wrapping up, the rewritten query  
1162  $\beta(Q_F, ?prov)$  is

1163 
$$(\text{SELECT } ?city \text{ (ProvAggSum(?prov}\oplus) \text{ AS ?prov)}$$
  
WHERE ((SELECT ?city ?prov $\oplus$   
WHERE ((SERVICE  $y_1$   
(SELECT ?city (ProvAggSum(?prov $\oplus\otimes\oplus)$  AS ?prov $\oplus\otimes$ )  
WHERE Reify((Danube, crosses, ?city), ?prov $\oplus\otimes\oplus$ )  
GROUP BY ?city))  
BIND (ServiceOp( $y_1, ?prov\oplus\otimes$ ) AS ?prov $\oplus\otimes$ )))  
UNION  
(SELECT ?city ?prov $\oplus$   
WHERE ((SERVICE  $y_2$   
(SELECT ?city (ProvAggSum(?prov $\oplus\otimes\oplus)$  AS ?prov $\oplus\otimes$ )  
WHERE Reify((Danube, crosses, ?city), ?prov $\oplus\otimes\oplus$ )  
GROUP BY ?city))  
BIND (ServiceOp( $y_2, ?prov\oplus\otimes$ ) AS ?prov $\oplus\otimes$ ))))  
GROUP BY ?city).

1164 Following the optimization rules, we can rewrite this query as follows:

```
1165 (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).
```

1166 In the spirit of the simplification rule no. 2 in Definition 35 (for chains of UNION operators),  
1167 we can remove two SELECT operations and obtained a simpler provenance query.

1168 To summarize, our query rewriting framework, Fed-NPCS, extends NPCCS to federated queries by  
1169 introducing the  $\otimes$  operator to track the provenance of intermediate results obtained from different  
1170 SPARQL endpoints. The extension preserves the soundness and completeness of NPCCS, ensuring  
1171 correct results even when querying across multiple sources. By leveraging existing base rewriting  
1172 rules and adapting them to handle federations, our approach provides a robust mechanism for  
1173 executing and combining federated SPARQL queries with annotated provenance information.

## 1174 6 Evaluation

1175 Our evaluation is structured in two parts. In the first part, i.e., Section 6.1, we evaluate NPCCS,  
1176 our how-provenance solution for SPARQL queries on centralized knowledge graphs using two  
1177 state-of-the-art benchmark datasets and two SPARQL engines. Then, Section 6.2 provides an  
1178 evaluation of Fed-NPCS on the FedShop federation benchmark [17] that provides federations of  
1179 different sizes.

### 1180 6.1 Evaluation on Centralized KGs (NPCCS)

1181 We conducted an extensive evaluation of NPCCS's viability for computing how-provenance by  
1182 assessing the runtime overhead incurred by the rewritten queries on centralized settings. This is  
1183 measured by comparing the runtime between the original queries without provenance annotations  
1184 and the queries obtained with our approach.

#### 1185 6.1.1 Experimental Setup

1186 **Environment.** NPCCS was implemented in Java, using the Java Development Kit (JDK) version  
1187 11. All the experiments were conducted on a computer with an AMD EPYC 7281 16-core processor,  
1188 256GB of RAM, and an 8 TB HDD disk running Ubuntu 18.04.6 LTS. We evaluated NPCCS on  
1189 two widely used RDF/SPARQL engines with support for RDF-star, namely GraphDB<sup>3</sup> (version  
1190 10.2.0) and Stardog<sup>4</sup> (version 9.1.0). For all our experiments, we set a timeout of 350 seconds for  
1191 individual query executions, to ensure consistent results, and reported the average response time  
1192 of the queries over five executions in a cold setting, i.e., after clearing the disk cache.

1193 **Competitor.** We compare NPCCS with SPARQLprov [30], a state-of-the-art solution for  
1194 how-provenance in SPARQL, which is also based on query rewriting. We used the implementation  
1195 provided with the paper [30], and extended it to support the RDF-star reification scheme.

<sup>3</sup> <https://graphdb.ontotext.com/>

<sup>4</sup> <https://www.stardog.com/>

1196 SPARQLprov and NPCS compute the same provenance polynomials since they both rely on  
1197 spm-semirings.

1198 **Synthetic Workload.** We employed the Watdiv [4] performance benchmark specifically  
1199 designed for RDF/SPARQL engines. Watdiv provides a data generator that can produce synthetic  
1200 datasets of varying sizes. Additionally, WatDiv includes 20 SELECT query templates, each  
1201 comprising 10 instantiated queries. The query templates are categorized into four types: linear  
1202 queries (L), star queries (S), snowflake-shaped queries (F), and complex queries (C). They are  
1203 all monotonic queries. We therefore introduced five additional non-monotonic query templates  
1204 (O) as proposed by [30]. These non-monotonic queries were created by enclosing one of the triple  
1205 patterns in the linear queries with an OPTIONAL clause. The triple pattern to be enclosed was  
1206 chosen randomly to ensure a diverse set of non-monotonic query templates.

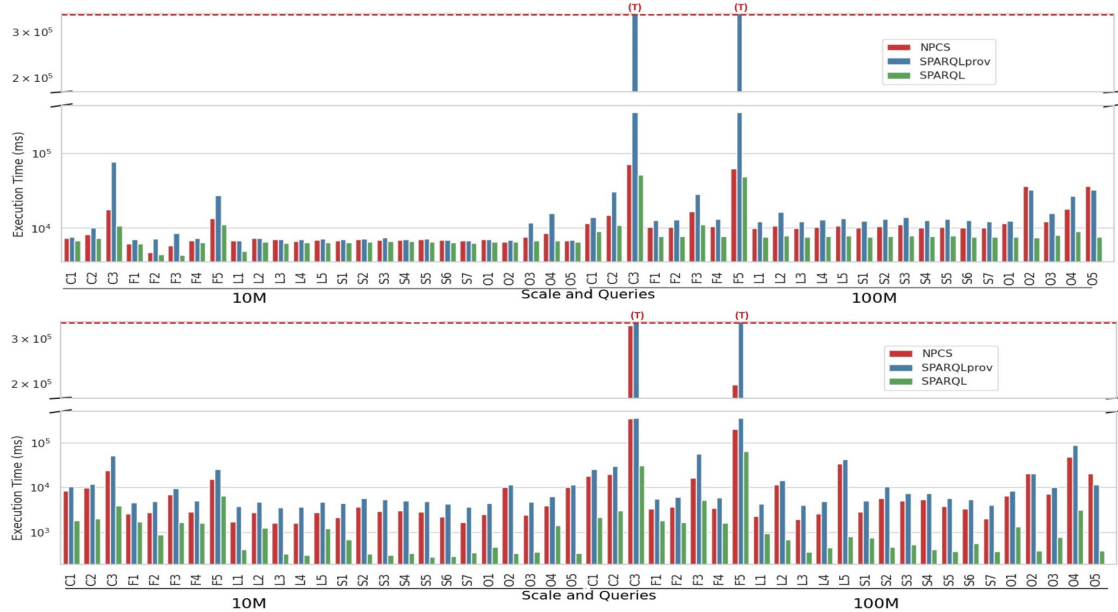
1207 We evaluated NPCS on the 10M-triple and 100M-triple Watdiv datasets that we reified using  
1208 the RDF-star and named graphs reification schemes. We excluded the standard reification from  
1209 the evaluation as it exhibits the worst performance according to [30]. Moreover, we created a  
1210 200M-triple dataset by duplicating every triple of the 100M-triple dataset and assigning a second  
1211 provenance identifier to the duplicates. This dataset simulates a challenging case where triples  
1212 have been extracted from more than one source.

1213 **Real Workload.** We tested NPCS and SPARQLprov on the WDBench benchmark [5], which  
1214 provides real-world data. The benchmark uses 15.2 billion triples encoded using the Wikidata  
1215 reification scheme from a 2023 Wikidata dump. The benchmark provides more than 800 queries  
1216 consisting of simple BGPs, some of them with OPTIONAL clauses. We took a sample of 150 queries  
1217 consisting of 50 single-triple-pattern queries, 50 non-monotonic queries (with OPTIONAL), and 50  
1218 monotonic queries with more than one triple pattern. The queries were randomly chosen.

## 1219 6.1.2 Results

1220 **Synthetic Workload.** Figure 2 compares the execution times of the original query with those of  
1221 the rewritten queries produced by NPCS and our competitor SPARQLprov on RDF-star data  
1222 when using GraphDB and Stardog. We measure the runtimes on the 10M and 100M Watdiv  
1223 datasets. We first notice that in all cases, rewriting the query to compute how-provenance incurs a  
1224 performance overhead regarding the execution of the original query. Not surprisingly, the overhead  
1225 increases with data size, but its behaviour also depends on the query engine. For instance, NPCS's  
1226 overhead ranges from 20% to 30% in GraphDB, and from 25% to 50% in Stardog. Figure 2  
1227 highlights that runtime across query templates exhibits higher variability in Stardog compared  
1228 to GraphDB. Regardless of the data size and the query engine, templates C3 and F5 are by far  
1229 the most challenging, and make our competitor SPARQLprov timeout on both GraphDB and  
1230 Stardog. The complexity of C3 is explained by its large number of intermediate results, whereas  
1231 for F5 it is caused by the large number of solutions.

1232 When we compare the query rewriting strategies, we notice the NPCS consistently outperforms  
1233 SPARQLprov in 98 out of our 100 studied cases. Also, NPCS is on average 25 times faster than  
1234 SPARQLprov. One can explain this performance difference by the fact that NPCS is a fully native  
1235 SPARQL solution, whereas SPARQLprov relies on a post-hoc decoding phase to compute the  
1236 provenance polynomials. Like NPCS, SPARQLprov rewrites the query to extract provenance  
1237 information. Unlike our approach, SPARQLprov encodes the structure of the how-provenance  
1238 annotations in additional columns in the result set. Those additional columns can be numerous  
1239 and encode the structure of the provenance polynomials. Decoding that information requires  
1240 running additional group and aggregation operations. Hence, the runtime of this decoding phase  
1241 is proportional to the number of query solutions times the maximal depth of the operator trees  
1242 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.

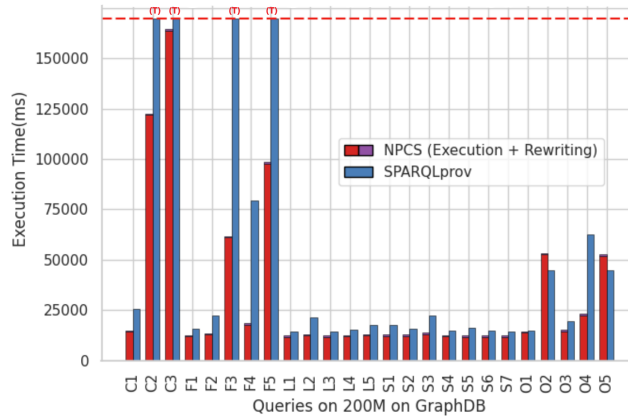
1243 F5, which is by far the template with the highest number of query solutions (173.6K solutions on  
 1244 average). NPCS, in contrast, carries out the grouping operations during query evaluation, which  
 1245 not only leverages the engine optimizations for grouping, but also makes it easier to deploy in  
 1246 real-world settings.

1247 Despite NPCS’s clear runtime advantage, SPARQLprov can exhibit comparable or better  
 1248 performance on very selective queries. This is demonstrated by the runtimes for queries O1, O2,  
 1249 and O5. In cases such as query templates O2 and O5 on GraphDB, NPCS’s strategy of evaluating  
 1250 grouping operations in the SPARQL engine does not pay off. This is so because the queries and  
 1251 their constituent triple patterns are very selective.

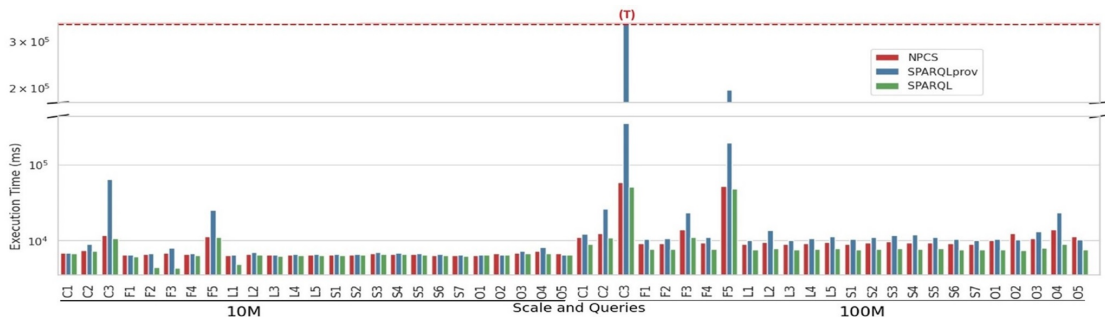
1252 Figure 3 shows the results for the rewritten queries on the 200M dataset on GraphDB. We  
 1253 observe similar trends as in the 100M scenario, except that SPARQLprov also times out on query  
 1254 templates C2 and F4. We omit the results for Stardog as they exhibit similar behavior as in the  
 1255 100M dataset.

1256 Finally, we evaluate NPCS on a different reification scheme, namely the popular named graphs  
 1257 strategy. The results are depicted in Figure 4 for the 10M and 100M datasets on GraphDB. We  
 1258 observe the same trends as for the RDF-star reification, that is, NPCS outperforms SPARQLprov  
 1259 consistently in 48 out of 50 studied cases. This shows that our approach is insensitive to the data  
 1260 reification scheme, which makes it applicable to any standard RDF/SPARQL engine. Similar  
 1261 results are observed for Stardog.

1262 **Real Workload.** We evaluate NPCS on WDBench. Figure 5 shows the results for GraphDB  
 1263 and Stardog. Each dot in the plot represents the execution of a query, either the original query or  
 1264 a rewritten query by NPCS or by SPARQLprov. Queries are plotted on the x-axis by increasing  
 1265 number of solutions, and the y-axis represents the execution time. We verify the same trend  
 1266 for both engines, namely that SPARQLprov’s query rewriting induces a much larger overhead  
 1267 than NPCS’s. While the overhead increases with the number of query results for both methods,  
 1268 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

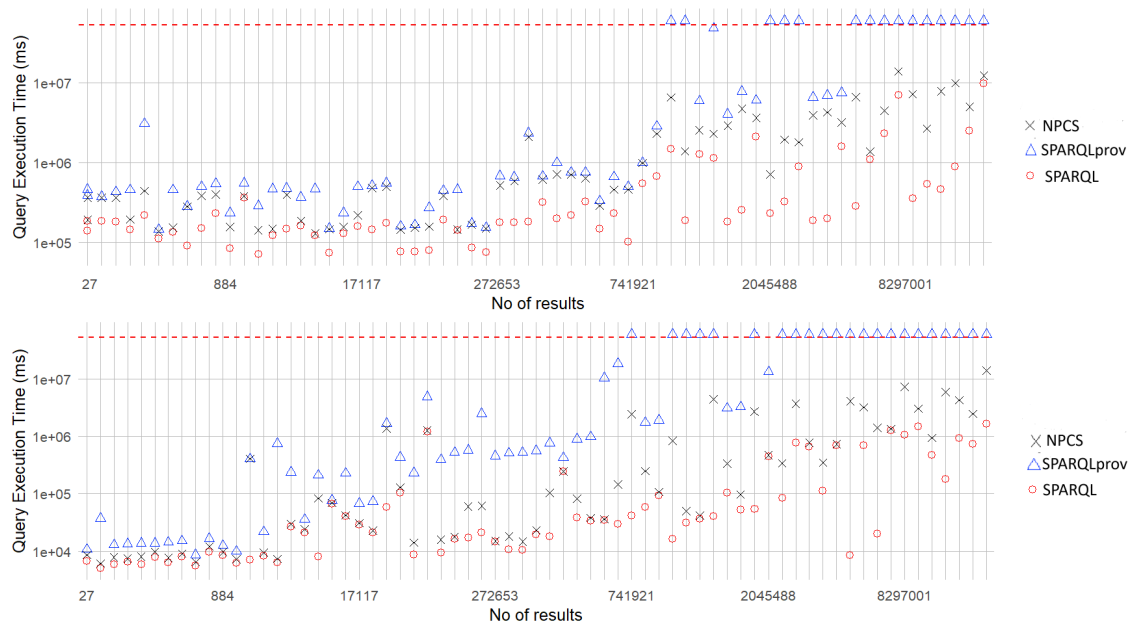
1269 of results is above 700K on Stardog (on GraphDB timeouts start after 1M solutions). We also  
 1270 observe that for queries with a few thousand results executed on Stardog, NPCS’s overhead can  
 1271 be minimal.

## 1272 6.2 Evaluation on a Federated Setting (Fed-NPCS)

### 1273 6.2.1 Experimental Setup

1274 **Environment.** Fed-NPCS was implemented in Java, using the Java Development Kit (JDK)  
 1275 version 11. All the experiments were conducted on the same hardware environment used in the  
 1276 centralized evaluation: a computer with an AMD EPYC 7281 16-core processor, 256GB of RAM,  
 1277 and an 8 TB HDD disk running Ubuntu 18.04.6 LTS. In this evaluation we tested Fed-NPCS on  
 1278 GraphDB<sup>5</sup> (version 10.2.0) as it exhibits better performance than Stardog in Figure 5. To simulate  
 1279 a federated environment, we deployed multiple endpoints hosting different RDF datasets in a  
 1280 distributed manner. Each dataset was hosted on separate instances of the engine, configured to  
 1281 act as individual SPARQL services accessible via the `SERVICE` keyword. The evaluation considered  
 1282 varying the number of distributed endpoints, ranging from 20 to 200, to assess the scalability  
 1283 and efficiency of the rewritten federated queries. As with the centralized evaluation, a timeout of  
 1284 350 seconds was enforced to individual query executions. For consistency, we report the average  
 1285 response time over five executions in a cold setting (with the disk cache cleared).

<sup>5</sup> <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

1286 **Query Workload.** We used the FedShop benchmark [17], which is specifically designed to  
 1287 evaluate the scalability of federated RDF/SPARQL query engines. The FedShop schema replicates  
 1288 a virtual catalog across autonomous vendors and rating sites, linking local and global entities  
 1289 using `owl:sameAs` statements, and generates federations of varying sizes using schema-based data  
 1290 generators while maintaining local independence and global interoperability. FedShop provides  
 1291 pre-configured federations and a comprehensive query workload, which we employed for our  
 1292 evaluation. We generated 10 federations  $F(N)$  with sizes  $N = \{20, 40, \dots, 200\}$ . Each federation  
 1293 has as many vendors as reviewing sites. For instance,  $F(200)$  consists of 100 vendors and 100  
 1294 review sites. All necessary instructions to install, configure, and run the benchmark are available  
 1295 on the FedShop GitHub repository<sup>6</sup>.

1296 The FedShop’s query generator was used to create a workload of 120 queries, each derived from  
 1297 12 template queries and executed across our 10 federations – from  $F(20)$  to  $F(200)$  – providing  
 1298 a robust testing environment. For sub-query decomposition and source selection, the FedShop  
 1299 benchmark relies on FedX [42], which follows a unions-first-joins-later (i.e., unions-over-joins)  
 1300 decomposition refined with FedX’s exclusive-groups optimization. FedShop provides pre-computed  
 1301 Reference Source Assignments (RSA) on top of this decomposition, which we used as the basis for  
 1302 our evaluation.

1303 The FedShop benchmark is designed to assess the performance and scalability of query  
 1304 federation engines across multiple sources. It categorizes queries based on their data location  
 1305 patterns, namely into, single-domain, multi-domain, and cross-domain queries. These are based  
 1306 on the queries’ join variables and on how they combine data from the different sources, and can  
 1307 be explained by the structure of the provenance explanations of their solutions. Single-domain  
 1308 queries are queries without global join variables and with a bounded subject in at least one triple  
 1309 pattern. Since that bounded subject appears only in one of the endpoints, this allows the query

<sup>6</sup> <https://github.com/GDD-Nantes/FedShop.git>

1310 to be executed on a single endpoint using one SERVICE clause, as seen for query templates Q9,  
 1311 Q11, and Q12. The provenance of their solutions consists of polynomials with a single  $\otimes$  operator,  
 1312 e.g.,  $e_1 \otimes (s_1 \otimes s_2)$  with endpoint  $e_1$ . Multi-domain queries also lack global join variables but  
 1313 do not include bounded subjects, requiring all triple patterns to be grouped into one SERVICE  
 1314 clause, with results potentially retrieved from multiple endpoints. Unlike single-domain queries the  
 1315 resulting provenance explanations include summations of polynomials where each term in the sum  
 1316 is labeled with a different source using the  $\oplus$  operator, e.g.,  $e_1 \otimes (s_1 \otimes s_2) \oplus e_2 \otimes (s_3 \otimes s_4)$ . This  
 1317 is the case for query templates Q1, Q2, Q3, Q4, Q6, Q8, and Q10. Cross-domain decomposition  
 1318 involves queries with global join variables linking data across multiple datasets, which requires  
 1319 multiple SERVICE clauses that combine results from several endpoints, as in query templates Q5  
 1320 and Q7. This translates into how-provenance explanations that contain products labeled with  
 1321 different sources, e.g.,  $e_1 \otimes (s_1 \otimes s_2) \otimes e_2 \otimes (s_3 \otimes s_4)$ .

1322 For our evaluation, we relied entirely on FedShop’s query generation, decomposition, and  
 1323 execution framework, enabling a straightforward comparison across varying federation sizes. The  
 1324 benchmark efficiently manages the minimal source selection over federated datasets and provides  
 1325 valuable insights into the performance of our approach. This corresponds to Step 1 in Section 5.4.2;  
 1326 we therefore evaluate the overhead of rewriting the federated plan to augment its results with  
 1327 how-provenance (Step 2).

## 1328 6.2.2 Results

1329 We evaluated the performance of Fed-NPCS using the FedShop benchmark on a subset of 12  
 1330 queries. Out of these, four were optional queries (Q2, Q3, Q7, and Q8). Each query consists  
 1331 of nine subqueries, and each subquery was executed 10 times, corresponding to the number of  
 1332 endpoints (from 20 to 200) involved in the execution. Since there is no direct competitor to  
 1333 Fed-NPCS, the only meaningful comparison is with a baseline federated SPARQL plan. This  
 1334 comparison allows us to assess the performance overhead introduced by provenance in a federated  
 1335 setting and to evaluate the scalability of Fed-NPCS with an increasing number of endpoints.

1336 Fed-NPCS demonstrated higher execution times compared to SPARQL in most cases, which can  
 1337 be attributed to the additional overhead of query rewriting. As shown in Figure 8, this overhead  
 1338 consists of two components: (i) query rewriting time, which is minimal across all evaluated  
 1339 configurations (typically <1% of total runtime at larger scales), and (ii) query execution time on  
 1340 the reified knowledge graph, which dominates the total runtime. To avoid visual distortion that  
 1341 arises from stacked bar charts on a logarithmic scale, Figure 8 presents these two components as  
 1342 percentage contributions on a linear scale, where rewriting time represents at most 15% of total  
 1343 runtime (WatDiv–GraphDB named graph reification at 10M triples) and decreases substantially  
 1344 as data scale increases. However, as the number of endpoints increased, Fed-NPCS showed a more  
 1345 stable performance pattern, particularly at handling a larger number of results. Below, we provide  
 1346 the performance outcomes for selected queries.

1347 **Impact of the number of endpoints.** Figure 6 illustrates the execution time in milliseconds  
 1348 for both the SPARQL and Fed-NPCS as the number of endpoints varies from 20 to 200. Each  
 1349 data point represents the average execution time for 12 queries across five runs<sup>7</sup>.

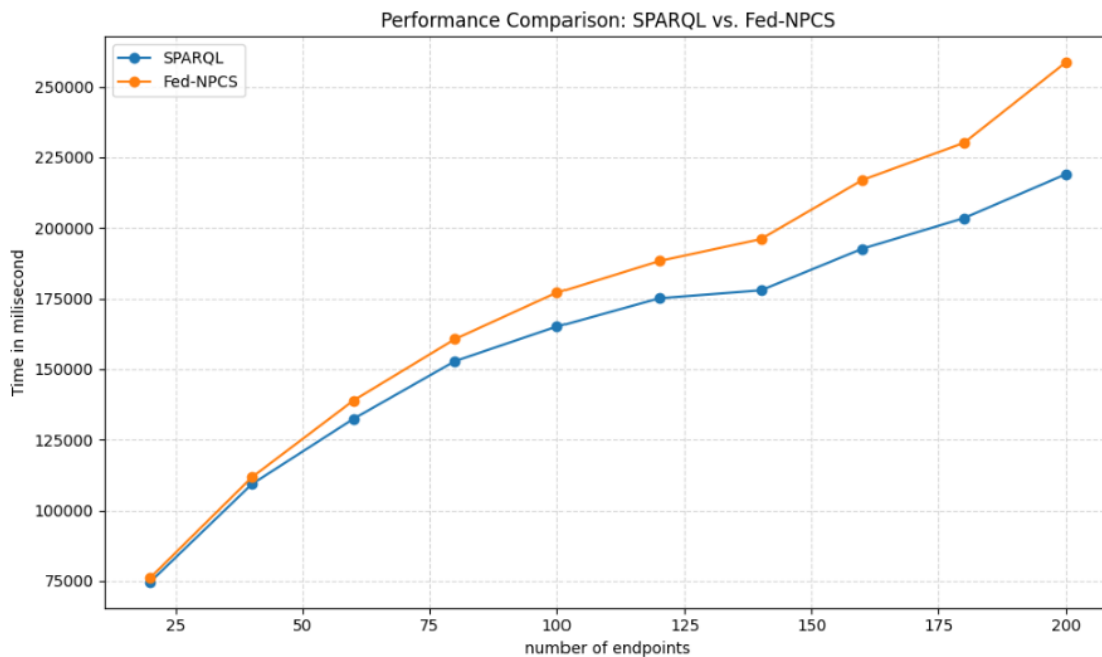
1350 The results show a clear trend: as the number of endpoints increases, both SPARQL and  
 1351 Fed-NPCS experience a gradual increase in execution time. Not surprisingly Fed-NPCS takes  
 1352 longer to execute than SPARQL across all federation sizes due to the additional overhead associated

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

1353 with provenance computation, which requires extra resources for tracking and processing query  
 1354 provenance data.

1355 Despite this overhead, Fed-NPCS exhibits a relatively stable and predictable growth pattern  
 1356 in execution time. Both systems demonstrate near-linear scalability as the endpoint count rises.  
 1357 Notably, the gap between SPARQL and Fed-NPCS widens slightly at higher endpoint counts  
 1358 as query processing on larger federations incurs the exchange of more query results – both  
 1359 intermediate and final. This overhead reaches 10.3% for the 200-source federation.

1360 In short, while Fed-NPCS incurs higher execution costs than SPARQL due to provenance  
 1361 computation, its consistent scalability suggests it can still be viable for large federations where  
 1362 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.



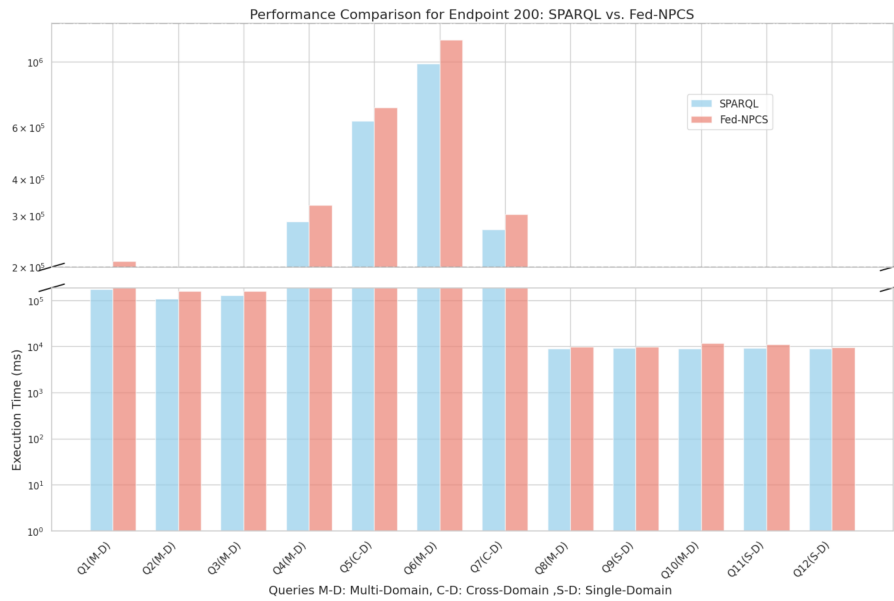
1363 **Figure 6** Average execution time of the different FedShop query templates (y-axis) vs. the size of the  
 1364 federation (x-axis).

1363 **Individual query analysis.** Figure 7 presents the execution time comparison between the  
 1364 baseline SPARQL engine and Fed-NPCS across 12 queries on 200 endpoints. The y-axis is in  
 1365 logarithmic scale and is also split to accommodate for the wide range of execution times.  
 1366

1367 Notably, for complex queries (such as Q5 and Q6), Fed-NPCS exhibits substantially higher  
 1368 execution times compared to plain SPARQL. However, for less intensive queries (like Q8-Q12 that  
 1369 are not cross-domain), the difference in execution times is less pronounced.

1370 The performance of individual queries reflects varying levels of complexity and data distribution.  
 1371 Notably, Q6 shows the highest execution time among all queries, as it is a multi-domain query  
 1372 involving the largest number of results. The high volume of intermediate results significantly  
 1373 contributes to the execution time for both SPARQL and Fed-NPCS, though the latter incurs  
 1374 additional overhead due to provenance computation.

1375 The cross-domain queries, such as Q5 and Q7, also exhibit longer execution times as they  
 1376 involve extensive data joining across federated endpoints. The performance of the multi-domain  
 1377 queries Q1-Q4 and Q8, is mainly explained by their respective numbers of intermediate and final



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

1378 results. For example, queries with more intermediate results tend to require more computation,  
 1379 leading to higher execution times.

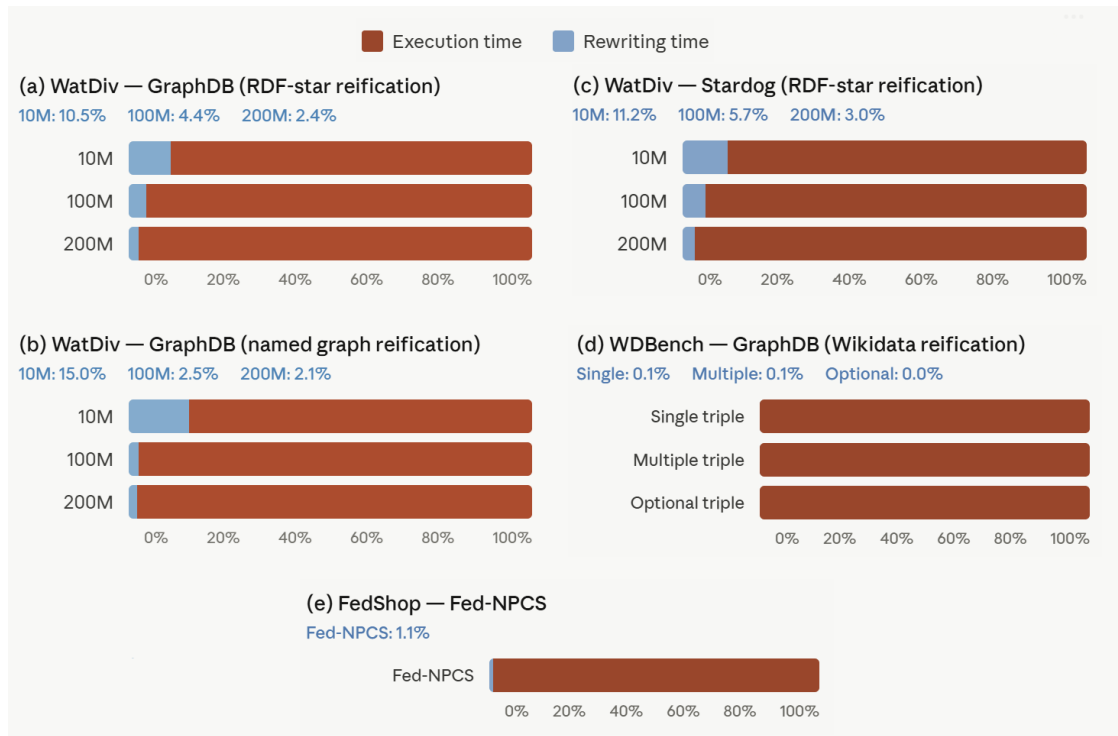
1380 Conversely, the single-domain queries, such as Q10, Q11, and Q12, are simpler and thus exhibit  
 1381 the shortest execution times. Both SPARQL and Fed-NPCS perform comparably on these queries,  
 1382 with minimal overhead observed in Fed-NPCS. This indicates that, while the Fed-NPCS introduces  
 1383 additional computation, its impact is less pronounced for simpler queries that do not involve  
 1384 complex joins or provenance tracking across domains.

1385 Overall, while the Fed-NPCS consistently incurs additional overhead due to provenance  
 1386 computation, the resulting system scales to complex multi-domain and cross-domain queries. This  
 1387 makes it a suitable choice for large-scale federated environments where provenance tracking is a  
 1388 priority.

## 1389 **7** Conclusions

1390 We have proposed NPCCS, a novel query rewriting method to compute how-provenance annotations  
 1391 for SPARQL query results. To the best of our knowledge, NPCCS is the first 100% SPARQL-based  
 1392 solution for how-provenance. NPCCS can be easily applied to standard and already deployed  
 1393 RDF/SPARQL engines, without the need for customized extensions or post-processing steps.  
 1394 Moreover, we introduce NPCCS’s federated counterpart, Fed-NPCS, which ports this idea to  
 1395 federations of SPARQL endpoints; this is also a novel extension, as, to our knowledge, there is  
 1396 currently no other engine allowing the computation of provenance information over federations.

1397 Our experimental evaluation on synthetic and real data shows that NPCCS’s native SPARQL  
 1398 rewriting outperforms the state of the art in how-provenance for SPARQL queries. The performance  
 1399 gains provided by our method allow us to compute provenance annotations for millions of query  
 1400 results on knowledge graphs with billions of triples. This makes NPCCS attractive for ETL processes  
 1401 on large volumes of data—a common scenario for multi-source KG construction and OLAP for  
 1402 KGs [19, 31–33, 35]. But NPCCS’s performance also makes it feasible to compute how-provenance



■ **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.

1403 explanations in federated settings. As shown by our evaluation of Fed-NPCS on the FedShop  
 1404 benchmark, our approach scales to federations with up to 200 sources. We expect this work to  
 1405 open new avenues in the fields of federated query optimization, access control, and data quality.

1406 In this paper the result of a SPARQL query has a column with character strings which represent  
 1407 expressions in  $\text{ProvExp}(X)$  in Polish notation. For example, a mapping  $\mu$  can be annotated with  
 1408 an expression  $(\oplus u_1 (\otimes u_2 u_3))$  where  $u_1$ ,  $u_2$ , and  $u_3$  are URLs identifying statements. To apply  
 1409 an homomorphism to a spm-semiring  $\mathcal{K}$  we can replace these URLs with the respective elements  
 1410 in  $\mathcal{K}$  and then apply the corresponding operations on  $\mathcal{K}$ . This procedure is less efficient than  
 1411 using built-in operations during the query evaluation. For example, for the natural numbers  
 1412 spm-semiring, we can redefine the operations  $\text{ProvAggSum}$ ,  $\text{ProvProd}$ , and  $\text{ProvDiff}$  as follows:

$$\begin{aligned}
 1413 \quad & \text{ProvAggSum}(\?x) = \text{sum}(\?x), \\
 & \text{ProvProd}(\?x_1, \dots, \?x_n) = (\?x_1 * \dots * \?x_n), \\
 & \text{ProvDiff}(\?x_1, \?x_2) = \text{if}(\?x_2 = 0, \?x_1, 0).
 \end{aligned}$$

1414 If a structure different from  $\mathbb{N}$  were used, we could still have redefined these operations based  
 1415 on custom functions supported by the SPARQL engine. Indeed, several SPARQL engines allow  
 1416 the definition of custom functions. Therefore, our query rewriting can be used as the base for an  
 1417 efficient application of homomorphisms. We plan the study of such an extension as future work.

1418 As another future work we intend to work on lazy approaches for how-provenance computation,  
 1419 that is, approaches where provenance is computed for a user-specified set of solutions. This avoids  
 1420 the execution of expensive queries for results that are not of interest of the user. Finally, we  
 1421 argue that the field of query processing and provenance for SPARQL would benefit from the  
 1422 development of new benchmarks specifically designed for provenance evaluation. While existing

1423 benchmarks such as FedShop [17] provide query workloads executable under different federation  
 1424 configurations, they do not directly support the evaluation of provenance computation—e.g.,  
 1425 they lack ground-truth provenance annotations or queries designed to stress-test provenance  
 1426 systems. New benchmarks addressing these gaps would enable a more systematic comparison of  
 1427 provenance-aware systems across both centralized and federated settings.

## 1428 Supplementary Material Statement

1429 The source code of NPCS and Fed-NPCS, along with scripts to recreate the experimental setup, all  
 1430 required libraries, queries, and results can be found at [https://github.com/ZubariaForthAcc/](https://github.com/ZubariaForthAcc/NPCS_Fed-NPCS)  
 1431 [NPCS\\_Fed-NPCS](https://github.com/ZubariaForthAcc/NPCS_Fed-NPCS), and in a long-term preservation archive at the University of Stuttgart [10].

---

## References

- 1 Maribel Acosta, Olaf Hartig, and Juan Sequeda. Federated RDF query processing. In Albert Zomaya, Javid Taheri, and Sherif Sakr, editors, *Encyclopedia of Big Data Technologies*, 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. doi:10.1007/978-3-642-25073-6\_2.
- 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. doi:10.1145/3589334.3645704.
- 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. doi:10.1007/978-3-319-11964-9\_13.
- 5 Renzo Angles, Carlos Buil-Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoc. Wdbench: A wikidata graph query benchmark. In Ulrike Sattler, Aidan Hogan, C. Maria Keet, Valentina Presutti, João Paulo A. Almeida, Hideaki Takeda, Pierre Monnin, Giuseppe Pirrò, and Claudia d’Amato, editors, *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. doi:10.1007/978-3-031-19433-7\_41.
- 6 Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *The Semantic Web – ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, volume 5318 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2008. URL: [https://doi.org/10.1007/978-3-540-88564-1\\_8](https://doi.org/10.1007/978-3-540-88564-1_8), doi:10.1007/978-3-540-88564-1\_8.
- 7 Renzo Angles and Claudio Gutierrez. The multiset semantics of SPARQL patterns. In Paul Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil, editors, *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 Eng. Bull.*, 41(1):51–62, 2018. URL: <http://sites.computer.org/debull/A18mar/p51.pdf>.
- 9 Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irimi Fundulaki, and Katja Hose. NPCS: native provenance computation for SPARQL. In Tat-Seng Chua, Chong-Wah Ngo, Ravi Kumar, Hady W. Lauw, and Roy Ka-Wei Lee, editors, *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, pages 2085–2093. ACM, 2024. doi:10.1145/3589334.3645557.
- 10 Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irimi Fundulaki, and Katja Hose. Code for Native Provenance Computation for Federated and Non-Federated SPARQL Queries. Dataset, DaRUS, V1, 2026. doi:10.18419/DARUS-5944.
- 11 Carlos Buil-Aranda. *Federated query processing for the semantic web*. PhD thesis, Technical University of Madrid, Spain, 2014. doi:10.3233/978-1-61499-350-6-i.
- 12 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.
- 13 Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory –*

- ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330, Berlin, Heidelberg, 2001. Springer. URL: [https://doi.org/10.1007/3-540-44503-X\\_20](https://doi.org/10.1007/3-540-44503-X_20), doi:10.1007/3-540-44503-X\_20.
- 14 Angelos Charalambidis, Antonis Troumpoukis, and Stasinou Konstantopoulos. SemaGrow: optimizing federated SPARQL queries. In *SEMANTiCS*, pages 121–128. ACM, 2015. doi:10.1145/2814864.2814886.
  - 15 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.
  - 16 Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Provenance for SPARQL queries. In Philippe Cudré-Mauroux, Jeff Hefflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *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. doi:10.1007/978-3-642-35176-1\_39.
  - 17 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.
  - 18 Renata Queiroz Dividino, Sergej Sizov, Steffen Staab, and Bernhard Schueler. Querying for provenance, trust, uncertainty and other meta knowledge in RDF. *J. Web Semant.*, 7(3):204–219, 2009. doi:10.1016/j.websem.2009.07.004.
  - 19 Luis Galárraga, Kim Ahlstrøm Jakobsen, Katja Hose, and Torben Bach Pedersen. Answering provenance-aware queries on RDF data cubes under memory budgets. In Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, volume 11136 of *Lecture Notes in Computer Science*, pages 547–565. Springer, 2018. URL: [https://doi.org/10.1007/978-3-030-00671-6\\_32](https://doi.org/10.1007/978-3-030-00671-6_32), doi:10.1007/978-3-030-00671-6\_32.
  - 20 Garima Gaur, Arnab Bhattacharya, and Srikanta Bedathur. How and why is an answer (still) correct? maintaining provenance in dynamic knowledge graphs. In Mathieu d’Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux, editors, *CIKM ’20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, pages 405–414. ACM, ACM, 2020. doi:10.1145/3340531.3411958.
  - 21 Floris Geerts and Antonella Poggi. On database query languages for k-relations. *J. Appl. Log.*, 8(2):173–185, 2010. doi:10.1016/j.jal.2009.09.001.
  - 22 Floris Geerts, Thomas Unger, Grigoris Karvounarakis, Irini Fundulaki, and Vassilis Christophides. Algebraic structures for capturing the provenance of SPARQL queries. *J. ACM*, 63(1):7:1–7:63, 2016. doi:10.1145/2810037.
  - 23 Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *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. URL: [https://ceur-ws.org/Vol-782/GoerlitzAndStaab\\_COLD2011.pdf](https://ceur-ws.org/Vol-782/GoerlitzAndStaab_COLD2011.pdf).
  - 25 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *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*, 15(1):107–165, 2024. doi:10.3233/SW-223201.
  - 27 Olaf Hartig. Querying trust in RDF data with tsparql. In Lora Aroyo, Paolo Traverso, Fabio Ciravegna, Philipp Cimiano, Tom Heath, Eero Hyvönen, Riichiro Mizoguchi, Eyal Oren, Marta Sabou, and Elena Simperl, editors, *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. URL: [https://doi.org/10.1007/978-3-642-02121-3\\_5](https://doi.org/10.1007/978-3-642-02121-3_5), 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, dec 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 Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini, editors, *WWW ’22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, WWW ’22, pages 1047–1057, New York,

- NY, USA, 2022. ACM. 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. doi:10.1007/978-3-319-18818-8\_17.
  - 32 Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Optimizing aggregate SPARQL queries using materialized RDF views. In Paul Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil, editors, *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 341–359, 2016. URL: [https://doi.org/10.1007/978-3-319-46523-4\\_21](https://doi.org/10.1007/978-3-319-46523-4_21), doi:10.1007/978-3-319-46523-4\_21.
  - 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. doi:10.1007/978-3-319-50112-3\_14.
  - 34 Roman Kontchakov and Egor V. Kostylev. On expressibility of non-monotone operators in SPARQL. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 369–379. AAAI Press, 2016. URL: <https://aaai.org/papers/38-12889-on-expressibility-of-non-monotone-operators-in-sparql>.
  - 35 Matteo Lissandrini, Katja Hose, and Torben Bach Pedersen. Example-driven exploratory analytics over knowledge graphs. In Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini, editors, *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pages 105–117. OpenProceedings.org, 2023. doi:10.48786/edbt.2023.09.
  - 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. doi:10.1007/978-3-319-68288-4\_28.
  - 37 Bofeng Pan, Natalia Stakhanova, and Suprio Ray. Data provenance in security and privacy. *ACM Comput. Surv.*, 55(14s), jul 2023. doi:10.1145/3593294.
  - 38 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009. doi:10.1145/1567274.1567278.
  - 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. doi:10.1007/978-3-540-68234-9\_39.
  - 40 Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance for blockchain. *Proc. VLDB Endow.*, 12(9):975–988, may 2019. URL: <http://www.vldb.org/pvldb/vol12/p975-ruan.pdf>, 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. doi:10.1007/978-3-319-07443-6\_13.
  - 42 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. doi:10.1007/978-3-642-25073-6\_38.
  - 43 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 Spatial (SEMMESES)*, volume 3443. CEUR Workshop Proceedings (CEUR-WS.org), July 2023. URL: <https://ceur-ws.org/Vol-3443/ESWC%5f2023%5fTrusDeKW%5fpaper%5f3033.pdf>.
  - 44 Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. Tripleprov: efficient processing of lineage queries in a native RDF store. In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *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.