Computing How-Provenance for SPARQL Queries via Query Rewriting

Daniel Hernández
Aalborg University, Denmark
danielh@cs.aau.dk

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

Katja Hose
Aalborg University, Denmark
khose@cs.aau.dk

ABSTRACT

Over the past few years, we have witnessed the emergence of large knowledge graphs built by extracting and combining information from multiple sources. This has propelled many advances in query processing over knowledge graphs, however the aspect of providing provenance explanations for query results has so far been mostly neglected. We therefore propose a novel method, SPARQLprov, based on query rewriting, to compute how-provenance polynomials for SPARQL queries over knowledge graphs. Contrary to existing works, SPARQLprov is system-agnostic and can be applied to standard and already deployed SPARQL engines without the need of customized extensions. We rely on spm-semirings to compute polynomial annotations that respect the property of commutation with homomorphisms on monotonic and non-monotonic SPARQL queries without aggregate functions. Our evaluation on real and synthetic data shows that SPARQLprov over standard engines incurs an acceptable runtime overhead w.r.t. the original query, competing with state-of-the-art solutions for how-provenance computation.

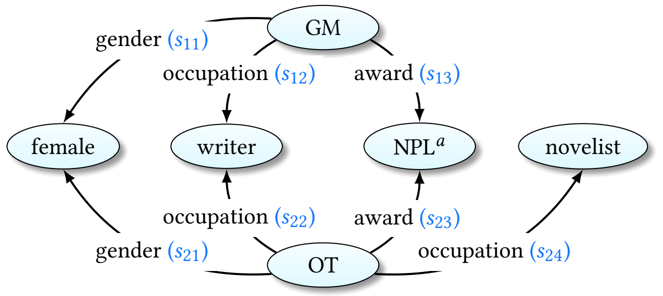
PVLDB Reference Format:
Daniel Hernández, Luis Galárraga, and Katja Hose. Computing How-Provenance for SPARQL Queries via Query Rewriting. PVLDB, 14(13): 3389-3401, 2021.
doi:10.14778/3484224.3484235

1 INTRODUCTION

The last few years have seen the emergence of large *knowledge graphs* (KGs): collections of triples ⟨ *subject*, *relation*, *object* ⟩, modeled using a graph abstraction [8] and constructed by extracting and integrating information from multiple providers. KGs are typically stored using the W3C standard RDF (Resource Description Framework) [7] and queried using SPARQL [24]. They find applications in multiple data-centric AI tasks, e.g., search engines, question answering, and smart assistants. This has not only given rise to a handful of academic [26, 28, 29, 34, 42] and industrial [9, 13, 15, 35, 40, 41] projects, but has also propelled many advances in graph stores and RDF/SPARQL engines.

Despite the current progress in RDF/SPARQL processing, the research in query provenance has particularly received little attention. At the same time, query provenance is essential for some applications given the central role of data integration in KG construction and maintenance. The provenance of a query result is an expression that encodes the "sources", i.e., relationships/triples, and processes that led to its computation. Reification is a prerequisite for provenance, as we need to identify the relationships in a KG to annotate query answers. For example, the triples in the RDF graph of Figure 1 have been reified with identifiers of the form s_{ij}.

Notable paradigms for query provenance are *lineage*, *why*-, and *how*-provenance [10]. Consider, for instance, the query that asks the KG in Figure 1 for people with an occupation who were awarded a Nobel Prize in Literature (NPL). The answers to this query are Gabriela Mistral (GM) and Olga Tokarczuk (OT). Lineage tells us the sources in the graph that contribute to the answer, e.g., s_{22}, s_{23}, and s_{24} for OT. Why-provenance tells us which sources contributed simultaneously to a solution in the query result set [11]. For OT, why-provenance is defined by the set $\{\{s_{22}, s_{23}\}, \{s_{23}, s_{24}\}\}$ that tells us that OT can be obtained by combining either s_{22} and s_{23}, or s_{22} and s_{24}. How-provenance extends why-provenance by structuring the sources of an answer as elements in the commutative semiring of polynomials with natural quotients $(\mathbb{N}[X], \oplus, \otimes, 0, 1)$ where elements of X identify relationships [22]. For instance, the answer OT is annotated with a polynomial of the form $s_{23} \otimes (s_{22} \oplus s_{24})$, where the operators $\otimes$ and $\oplus$ denote conjunctive and disjunctive derivation processes. The key property of commutative semirings is that they commute with homomorphisms. This property makes polynomials solid proofs that can be used in multiple applications [18].



[a]Nobel Prize in Literature

Figure 1: Running example knowledge graph

It has been shown that commutative semirings cannot explain the provenance of *non-monotonic queries*, that is, queries where an answer can cease to be an answer after adding data to the database [19, 20]. To guarantee this property, Geerts et al. [20] propose the spm-semiring formalism. Spm-semirings extend polynomial semirings with a difference operation ⊖ that can model non-monotonic SPARQL operators ("spm" stands for SPARQL minus). Geerts et al. prove that spm-semirings can model provenance

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484235

for monotonic and non-monotonic SPARQL queries, however they do not show how to compute such provenance.

This paper builds upon spm-semirings and proposes a method to compute how-provenance annotations for the answers of SPARQL queries. We target SPARQL because of its widespread use, its relevance for prominent public knowledge graphs [28, 29, 42], and its status as a W3C standard. Differently from state-of-the-art solutions [4, 21, 43], our approach holds the commutation with homomorphisms property for non-monotonic SPARQL queries.

Our techniques rely on query rewriting. This design decision makes our approach system-agnostic, and thus applicable to any graph store with a SPARQL interface. This notably includes a large variety of publicly available interfaces on the Web of Data, i.e., SPARQL endpoints. In summary, our contributions are:

- A novel query rewriting approach, SPARQLprov, that delivers query solutions annotated with how-provenance polynomials for SPARQL queries including the clauses AND, UNION, MINUS, OPTIONAL, FILTER, BIND, and SELECT.
- An approach to annotate solutions of SPARQL aggregate queries with lineage expressions.
- An extensive experimental evaluation of the runtime overhead and scalability of SPARQLprov on real and synthetic data when using different triple stores and reification schemes. We also compare SPARQLprov to two state-of-the-art how-provenance solutions, namely TripleProv [43] and GProM [4].

The remainder of this paper is structured as follows. Section 2 surveys the related work in provenance for query results, with focus on SPARQL queries. Section 3 then discusses preliminaries, whereas Section 4 elaborates on our contributions. Our evaluation is detailed in Section 5. Finally, Section 6 concludes the paper with an outlook to future work.

2 RELATED WORK

Algebraic Structures for Provenance. The use of semirings to model the management of annotated data was pioneered by the work of Green et al. [22]. This work uses a *commutative semiring* to annotate answers of queries in Datalog and the positive fragment of relational algebra (i.e., *selection-project-join-union* queries). The *non-monotonic* operators (left-outer join and relational difference) are not expressible in the semiring framework [22] and require an extension of the algebraic structure with a *monus* operator to account for the relational difference [19]. This extended structure is called m-semiring and is used by the ProvSQL system [38].

The work of Damasio et al. [12] shows that it is possible to compute provenance annotations on the m-semiring for SPARQL queries by rewriting the queries into relational algebra. However, this model does not allow for concise polynomials for SPARQL, which hampers its practical usability [20]. On those grounds, Geerts et al. [20] introduce the spm-semirings formalism that guarantees concise explanations for SPARQL queries including the non-monotonic operators (OPTIONAL and MINUS), and that is compatible with existing annotation frameworks for semantic data [14, 25]. Our work builds upon spm-semirings and devises a concrete method to compute how-provenance via query rewriting.

SPARQL Engines with Provenance Support. Wylot et al. [43] proposed *TripleProv*, which computes how-provenance annotations in the semiring model for select queries with basic graph patterns and the operators UNION and OPTIONAL. However, TripleProv does not guarantee the property of commutation with homomorphisms for queries with OPTIONAL due to its reliance on the semiring model. Furthermore, TripleProv counts on its own customized engine that stores data as molecules, i.e., star patterns. Hence, it cannot be deployed on top of existing engines in contrast to our approach.

By relying on query rewriting, solutions such as Perm [21] and GProM [4] can be deployed on top of existing engines. Nevertheless, their annotations are not always suitable for SPARQL queries because these solutions are designed for relational databases. Moreover, both GProM and Perm rely on the semirings framework, thus their annotations do not commute with homomorphisms for non-monotonic queries.

Other solutions exploit provenance annotations for specific applications. Avgoustaki et al. [6] propose a framework to capture the triple and attribute provenance polynomials of data added via SPARQL monotonic insert updates. Halpin et al. [23] use SPARQL insert updates to implement a Git-like version control system for RDF. A recent work [18] proposes a strategy to maintain views of SPARQL queries using how-provenance annotations on answers of a set of target queries in the presence of updates.

3 PRELIMINARIES

RDF. Assume four countable infinite pairwise disjoint sets $\mathbf{I}$, $\mathbf{B}$, $\mathbf{L}$, and $\mathbf{V}$, called *IRIs*, *blank nodes*, *literals*, and *variables*. An *term* is an element in $\mathbf{T} = \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. An *RDF triple* is a triple $(s, p, o) \in \mathbf{I} \cup \mathbf{B} \times \mathbf{I} \times \mathbf{T}$ where s is the *subject*, p the *predicate* and o the *object*. An *RDF graph* (or just a graph) is a set of RDF triples.

SPARQL. A *SPARQL selection formula* is defined recursively as follows. If $t_1, t_2 \in \mathbf{V} \cup \mathbf{T}$, and $x \in \mathbf{V}$, then $(t_1 = t_2)$ and $\text{bound}(x)$ are atomic selection formulas. If φ_1 and φ_2 are SPARQL selection formulas, then the expressions $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $\neg\varphi_1$ are also SPARQL selection formulas.

A *SPARQL query* is defined recursively as follows. A triple from $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{T} \cup \mathbf{V})$ is a query called a *triple pattern*[1]. A set of triple patterns is called a *basic graph pattern* (BGP). If P, P_1 and P_2 are queries, and φ is a selection formula then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ OPTIONAL } P_2)$, $(P_1 \text{ OPTIONAL}_{\varphi} P_2)$, $(P_1 \text{ MINUS } P_2)$, $(P_1 \text{ DIFF } P_2)$, $(P_1 \text{ DIFF}_{\varphi} P_2)$, and $(P \text{ FILTER } \varphi)$ are queries[2].

Assume a set $\mathbf{F}$ disjoint with $\mathbf{T} \cup \mathbf{V}$, called the set of regular function names, such that each function name $f \in \mathbf{F}$ is associated to a function $\hat{f} : (\mathbf{T} \cup \{\text{error}\})^n \rightarrow \mathbf{T} \cup \{\text{error}\}$, n is a natural number, and "error" is an element that is not a term and denotes unbound values or errors. If P is a query, $f \in \mathbf{F}$, $(t_1, \ldots, t_n) \in (\mathbf{T} \cup \mathbf{V})^n$, and $x \in \mathbf{V}$, then $(P \text{ BIND } (f(t_1, \ldots, t_1) \text{ AS } x))$ is a query. Let $\mathbf{A}$ be the set of aggregate functions, such that each function name $g \in \mathbf{A}$ is associated to a commutative monoid $(\mathbf{T}, +_g, 0_g)$. If W is a finite set of variables, $g_1, \ldots, g_n \in \mathbf{A}$, $x_1, \ldots, x_n, y_1, \ldots, y_n \in \mathbf{V}$, and P is a query, then $(\text{SELECT } W \text{ WHERE } P)$ and $(\text{SELECT } W\ (g(x_1) \text{ AS } y_1) \ldots (g_n(x_n) \text{ AS } y_n) \text{ WHERE } P \text{ GROUP BY } W)$ are queries.

[1]For simplicity, we do not include blank nodes in triple queries. They are expressible by introducing fresh variables and projecting them out with the SELECT operator.
[2]The operator DIFF refers to the difference operator considered by Geerts et. al [20], and even though it is not officially part of SPARQL, it is implemented in some existing engines (e.g., Virtuoso).

Let us denote by $\mathbf{G}$ the set of all possible RDF graphs, and by $\mathbf{P}$ the set of all possible SPARQL queries. A *solution mapping* (or just a *mapping*) is a partial function $\mu : \mathbf{V} \to \mathbf{T}$ where the domain of μ, $\mathrm{dom}(\mu)$, is the subset of $\mathbf{V}$ where μ is defined. The set of mappings is denoted by $\mathbf{M}$. The evaluation of a query $P \in \mathbf{P}$ on a graph $G \in \mathbf{G}$ is defined as a function $[\![P]\!]_G$ that returns a multiset of mappings $\mu \in \mathbf{M}$ according to the semantics defined in [3, 32, 36]. Given a basic graph pattern P, and a mapping μ whose domain includes all variables in P, we write $\mu(P)$ to denote the RDF graph resulting from replacing each variable $?x$ occurring in P by $\mu(?x)$. We write $\mu(?x) = \text{error}$ to denote that variable $?x$ is not in the domain of mapping μ. Given a set of variables W, we write $\mu|_W$ to denote the mapping resulting from limiting the domain of μ to $\mathrm{dom}(\mu) \cap W$. The *domain of a query* $P \in \mathbf{P}$, denoted by $\mathrm{dom}(P)$ in this paper and called *in-scope variables* by the specification [24, §18.2.1], is a set of variables defined recursively on the structures of queries to satisfy $\mathrm{dom}(\mu) \subseteq \mathrm{dom}(P)$ for every solution $\mu \in [\![P]\!]_G$. Two mappings μ_1 and μ_2 are said to be compatible – denoted by $\mu_1 \sim \mu_2$ – if $\mu_1(?\mathtt{x}) = \mu_2(?\mathtt{x})$ for every variable $?\mathtt{x} \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$. We write $\mu \models \varphi$ to denote that formula φ is evaluated as true on mapping μ (see [36]). We write $\mu_\emptyset$ to denote the mapping with an empty domain. Note that two mappings with disjoint domains are compatible, and the empty mapping is compatible with every mapping. Given a mapping μ and a selection formula φ, we write $1_{\mu \models \varphi}$ to denote the value 1 in case that $\mu \models \varphi$, and 0 otherwise. Given a triple query (s, p, o) and a mapping μ, we write $\mu(s, p, o)$ to denote the triple resulting from replacing $?\mathtt{x}$ with $\mu(?\mathtt{x})$ for each variable $?\mathtt{x}$ in the triple query.

We omit the semantics of SPARQL for space reasons and refer the reader to [3, 32, 33, 36].

Structures for provenance in SPARQL. Our goal is to annotate solution mappings from SPARQL queries with how-provenance polynomials. Those polynomials are built upon an algebraic structure called *spm-semiring*. In the following we introduce the notions of spm-semirings and describe their use.

A *commutative monoid* $\mathcal{M}$ is an algebraic structure $(M, +_\mathcal{M}, 0_\mathcal{M})$ where M is a non-empty set closed under a commutative and associate binary operation $+_\mathcal{M}$, and $0_\mathcal{M}$ is an identity for $+_\mathcal{M}$. A *commutative semiring* $\mathcal{K}$ is a structure $(K, +_\mathcal{K}, \times_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ where $(K, +_\mathcal{K}, 0_\mathcal{K})$ and $(K, \times_\mathcal{K}, 1_\mathcal{K})$ are commutative monoids, $\times_\mathcal{K}$ is distributive over $+_\mathcal{K}$, and $0_\mathcal{K} \times_\mathcal{K} x = 0$. A *spm-semiring* $\mathcal{K}$ is an algebraic structure [20] $(K, +_\mathcal{K}, \times_\mathcal{K}, -_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ where $(K, +_\mathcal{K}, \times_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ is a commutative semiring, and the following axioms hold: $x -_\mathcal{K} x = 0$; $x -_\mathcal{K} (y +_\mathcal{K} z) = (x -_\mathcal{K} y) -_\mathcal{K} z$; $x \times_\mathcal{K} (y -_\mathcal{K} z) = (x \times_\mathcal{K} y) -_\mathcal{K} z$; and $(x -_\mathcal{K} (x -_\mathcal{K} y)) +_\mathcal{K} (x -_\mathcal{K} y) = x$.

Examples of spm-semirings are: (i) The Boolean spm-semiring $(\mathbb{B}, \vee, \wedge, \nrightarrow, \bot, \top)$, where $\mathbb{B} = \{\bot, \top\}$, $\vee, \wedge$, and $\nrightarrow$ denote the logical connectives disjunction, conjunction, and material nonimplication[3]; (ii) the spm-semiring of natural numbers $(\mathbb{N}, +, \times, \ominus, 0, 1)$, where $+$ and $\times$ are the usual sum and product of natural numbers, and $\ominus$ is the operation defined as $x \ominus y = x$ if $y = 0$, and $x \ominus y = 0$ if $y \neq 0$; (iii) the provenance semiring $(\mathbb{N}[X], \oplus, \otimes, \ominus, 0, 1)$ where X is a finite set (representing sources) and $\mathbb{N}[X]$ is the set of equivalence classes of the expressions in the closure of the set $\mathbb{N} \cup X$ under the operations $\oplus$, $\otimes$, and $\ominus$.

[3]The material nonimplication is defined by the identity $x \nrightarrow y = x \wedge \neg y$.

If we conveniently annotate RDF triples and solution mappings with the elements of an spm-semiring, we can extend the SPARQL algebra with provenance annotations. Given a set A and an spm-semiring $\mathcal{K}$, an annotation function $f : A \to K$ is called a *$\mathcal{K}$-set* over A if the set $\mathrm{supp}(f) = \{x \in A \mid f(x) \neq 0\}$ – called the *support* of f – is finite. If $a \in A$, we call $f(a)$ its *$\mathcal{K}$-value*. If A is a set of mappings, we say f is a *$\mathcal{K}$-relation* and denote it by Ω, whereas if A is a set of triples, we say the annotation function is a *$\mathcal{K}$-graph*, which we denote by G.

$\mathcal{K}$-relations extend the notions of sets and bags of mappings defined for the answers of SPARQL queries on RDF graphs. For example, we can annotate mappings with elements in the Boolean spm-semiring $\mathbb{B}$ to represent sets of mappings, or with elements in the spm-semiring of natural numbers $\mathbb{N}$ to represent multisets of mappings. In the first case, the annotations tell us if a mapping belongs to a set; in the second case, annotations encode multiplicities.

An *spm-semiring homomorphism* is a mapping $h : \mathcal{K} \to \mathcal{K}'$ where $\mathcal{K}$ and $\mathcal{K}'$ are spm-semirings, $h(0_\mathcal{K}) = 0_{\mathcal{K}'}$, $h(1_\mathcal{K}) = 1_{\mathcal{K}'}$, $h(x +_\mathcal{K} y) = h(x) +_{\mathcal{K}'} h(y)$, $h(x \times_\mathcal{K} y) = h(x) \times_{\mathcal{K}'} h(y)$, and $h(x -_\mathcal{K} y) = h(x) -_{\mathcal{K}'} h(y)$. Given a $\mathcal{K}$-set S over a set A, and a homomorphism $h : \mathcal{K} \to \mathcal{K}'$, we write $h(S)$ to denote the $\mathcal{K}'$-set S' such that for every element $a \in A$ it holds that $S'(a) = h(S(a))$.

Recall that $\mathcal{K}$-graphs and $\mathcal{K}$-relations provide semantics for the answers of SPARQL queries, however the semantics of the queries themselves must also be adapted to return $\mathcal{K}$-relations. Geerts et al. [20] propose the following definition:

Definition 3.1. Given an spm-semiring $\mathcal{K}$, a query $Q \in \mathbf{P}$ consiting of a combination of triple patterns with the operators AND, UNION, DIFF, OPTIONAL, FILTER, and SELECT, and a $\mathcal{K}$-graph G, we write $(\!|P|\!)_G$ to denote the $\mathcal{K}$-relation defined recursively as follows:

$$(\!|(s,p,o)|\!)_G(\mu) = G(\mu(s,p,o)),$$
$$(\!|\textsc{select}\ W\ \textsc{where}\ P|\!)_G(\mu) = \textstyle\sum_{\mu'|_W=\mu} (\!|P|\!)_G(\mu'),$$
$$(\!|P\ \textsc{filter}\ \varphi|\!)_G(\mu) = (\!|P|\!)_G(\mu) \times_\mathcal{K} 1_{\mu\models\varphi},$$
$$(\!|P_1\ \textsc{union}\ P_2|\!)_G(\mu) = (\!|P_1|\!)_G(\mu) +_\mathcal{K} (\!|P_2|\!)_G(\mu),$$
$$(\!|P_1\ \textsc{and}\ P_2|\!)_G(\mu) = \textstyle\sum_{\mu=\mu_1\cup\mu_2} ((\!|P_1|\!)_G(\mu_1) \times_\mathcal{K} (\!|P_2|\!)_G(\mu_2)),$$
$$(\!|P_1\ \textsc{diff}\ P_2|\!)_G(\mu) = (\!|P_1|\!)_G(\mu) -_\mathcal{K} (\textstyle\sum_{\mu'\sim\mu} (\!|P_2|\!)_G(\mu')),$$
$$(\!|P_1\ \textsc{optional}\ P_2|\!)_G(\mu) = (\!|P_1\ \textsc{and}\ P_2|\!)_G(\mu) +_\mathcal{K} (\!|P_1\ \textsc{diff}\ P_2|\!)_G(\mu),$$

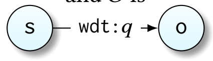
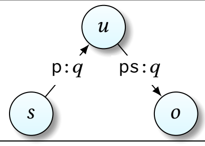
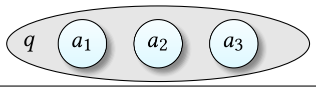
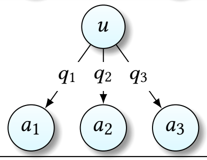
where $\sum$ denotes sums using the sum operation $+_\mathcal{K}$. The algebra defined above is called *$\mathcal{K}$-annotated SPARQL algebra*, which is shown to hold the property of commutation of homorphisms according to Geerts et al. [20].

Definition 3.2. Given an spm-semiring homomorphism $h : \mathcal{K} \to \mathcal{K}'$, a query Q is said to hold the commutation with homomorphisms property if for any graph G it satisfies $h((\!|Q|\!)_G) = (\!|Q|\!)_{h(G)}$.

Commutation with homomorphisms is a desired property because it allows specializing abstract provenance polynomials for different applications such as multiplicities, trust scores, cost, deletion propagation, and probabilistic databases [2]. For example, consider the $\mathcal{K}$-graph G depicted in Figure 1, and the query

$$Q = \textsc{select}\ ?\mathtt{person}\ \textsc{where}\ (?\mathtt{person}, \mathtt{award}, \mathtt{NPL})\ \textsc{diff}\ (?\mathtt{person}, \mathtt{occupation}, \mathtt{novelist})$$

Table 1: Reification scheme examples. On the left: a binary relation (top) and a 3-ary relation (bottom). On the right: reification using the Wikidata scheme (top) and the Direct Mapping reification for n-ary relationships (bottom).

Relationship r and graph G	Graph G^r = Reify(r, u)
r = wdt:$q(s, o)$, and G is 	
$r = q(a_1, a_2, a_3)$, and G is 	

and let μ_{M} = {?person ↦ GM} and μ_{T} = {?person ↦ OT} be two mappings corresponding to the solutions of the first triple pattern. Because GM is a winner of the Nobel Prize in Literature, but she is not a novelist, it holds that $(\!|Q|\!)_G(\mu_{\mathsf{M}}) = s_{13} \ominus 0$. Similarly, $(\!|Q|\!)_G(\mu_{\mathsf{T}}) = s_{23} \ominus s_{24}$. By applying an spm-semiring homomorphism $h : \mathbb{N}[X] \to \mathbb{B}$ (to the Boolean semiring) such that $h(s_{13}) = \top$, $h(s_{23}) = \top$, and $h(s_{24}) = \top$, the provenance polynomials explain why, according to the SPARQL set semantics, GM is an answer of Q and why OT is not. Indeed, $h(s_{13} \ominus 0) = \top \nrightarrow \bot = \top$ and $h(s_{23} \ominus s_{24}) = \top \nrightarrow \top = \bot$.

Reification Schemes. In order to support provenance annotations for query answers, we need to use a surrogate entity u to refer to each relationship r (e.g., assign identifiers to triples in an RDF graph). Those surrogate entities define the set X that is the building block of our provenance annotations in the spm-semiring $\mathbb{N}[X]$. The process of encoding the data using these surrogate entities is called reification. RDF and SPARQL allow multiple ways to reify the data [16, 27]. A reification scheme is a function Reify that receives an n-relationship r and a term $u \in \mathbf{I}$, and returns a graph Reify(r, u), called the *encoding of r reified as u*. A reification scheme defines an encoding of $\mathbb{N}[X]$-graphs G as regular graphs G^r (i.e., without annotations). We write Reify$^{-1}(G^r)$ to denote the $\mathbb{N}[X]$ graph defined as follows[4]:

$$\mathsf{Reify}^{-1}(G^r)((s, p, o)) = \bigoplus\nolimits_{\mu \in [\![\mathsf{Reify}(p(s,o),?\mathsf{u})]\!]_{G^r}} \mu(?\mathsf{u}).$$

Table 1 depicts two RDF reification schemes. The Wikidata scheme is shown in the first row. Wikidata uses the prefix wdt for the original predicates, i.e., predicates in a non-reified relationship such as wdt:q (left cell). The prefix wdt is replaced by the prefixes p and ps to create the new relationships p:q, ps:q that link the new reification entity u with the subject and the object of the original relationship (right cell). The second row illustrates a scheme to codify n-ary relations in RDF (depicted as an hyper-edge on the left), and is similar to the codification used to represent relational databases in RDF according to the direct mapping strategy [39].

[4]We allow for a sum of sources because one triple can represent several relationships. For instance, Wikidata codifies that Bachelet was President of Chile with a triple for which there are two reifying entities. Each entity describes one of two periods in which Bachelet held that position [27].

Reification schemes are generalized for basic graph pattern by admitting variables in the input relationships and basic graph patterns in the output. For example, the relationship queries R_1 = wdt:q(?x, o) and R_2 = q(b_1, ?x, b_3) that result from introducing variables in the relationships in Table 1 are reified as follows:

Reify(R_1, ?u) = (?x, p:q, ?u) AND (?u, ps:q, o),
Reify(R_2, ?u) = (?u, q:1, b_1) AND (?u, q:2, ?x) AND (?u, q:3, b_3).

4 COMPUTING HOW-PROVENANCE

We now present our method, SPARQLprov, to compute how-provenance for the answers of SPARQL queries. Given a query Q and a graph G, our goal is to annotate Q's answers with polynomials in the spm-semiring $\mathbb{N}[X]$, where X is the set of identifiers assigned to the triples in G via reification. We do so by rewriting Q into a new query Q' so that its evaluation on G returns mappings in the support of Q annotated with how-provenance annotations. Those annotations are encoded as part of the result of Q', i.e., as plain relations, and must be decoded into spm-semiring polynomials, that is, into $\mathbb{N}[X]$-relations.

This section is organized in four parts. In Section 4.1, we extend the $\mathbb{N}[X]$-annotated SPARQL algebra to account for some missing operators. Then, Section 4.2 describes how to compute how-provenance for SPARQL queries with monotonic operators. This is followed by an extension for non-monotonic operators in Section 4.3. These extensions exclude queries with aggregation, which we describe in Section 4.4.

4.1 Extending the K-annotated SPARQL Algebra

The $\mathcal{K}$-annotated SPARQL algebra proposed by Geerts et al. [20] – and described in Section 3 – is not defined for all the SPARQL operators considered in this paper. For instance, the empty basic graph pattern and the operator OPTIONAL$_\varphi$ (defined in SPARQL 1.0) were not studied by Geerts et al. Moreover, the operators MINUS and BIND succeed the publication of the spm-semirings formalism. In this section we extend the $\mathcal{K}$-annotated SPARQL algebra to account for the missing operators, except for aggregate queries because they are not expressible with the spm-semiring [2].

Given a $\mathcal{K}$-graph G, the semantics of the empty basic graph pattern are given by the identities $(\!|\{\}|\!)_G(\mu_\emptyset) = 1_{\mathcal{K}}$ and $(\!|\{\}|\!)_G(\mu) = 0_{\mathcal{K}}$ if $\mu \neq \mu_\emptyset$, where $\mu_\emptyset$ is the empty mapping. We highlight that producing answers from no data is a distinctive feature of SPARQL with respect to relational algebra, since in relational algebra all answers are generated from at least one tuple in the database.

Given a $\mathcal{K}$-graph G and a non-aggregate SPARQL query P, the semantics of the BIND operator are given by the identity $(\!|P \text{ BIND } f(t_1, \ldots, t_n) \text{ AS } y|\!)_G = (\!|P|\!)_G$. Intuitively, the operation BIND does not modify the provenance of answers of query P because the additional values are implicit in them.

In regards to the operators DIFF$_\varphi$, OPTIONAL$_\varphi$, and MINUS, it is known that they are expressible in terms of the operators originally supported by the $\mathcal{K}$-annotated SPARQL algebra [33]. We can therefore extend the annotated algebra for these operators by relying on these translations, that are omitted here due to the limited space.

It can be shown by induction on the structure of queries that the algebra resulting from the extension of the $\mathcal{K}$-annotated SPARQL algebra described in this section holds the commutation with homomorphisms property.

4.2 Provenance for monotonic operators

Before formalizing our approach , we illustrate the principle with an example. The technique described in the example is implicit in state-of-the-art provenance systems, such as GProM [4] and HUKA [18]. Also, this example aims to familiarize the reader with the encoding of polynomials in tables.

Example 4.1. Consider graph G in Figure 1 and the following query Q asking for female writers or novelists awarded with the Nobel Prize in Literature.

```
SELECT ?person
WHERE ((?person, occupation, writer) UNION
        (?person, occupation, novelist)) AND
       (?person, gender, female) AND (?person, awarded, NPL)
```

The result of query Q under the $\mathbb{N}[X]$-annotated SPARQL algebra is the following $\mathbb{N}[X]$-relation denoted by Ω:

$$\Omega = \left[\begin{array}{c||c} \text{?person} & \Omega(\mu) \\ \hline \text{GM} & s_{12} \otimes s_{11} \otimes s_{13} \\ \text{OT} & (s_{22} \oplus s_{24}) \otimes s_{21} \otimes s_{23} \end{array}\right],$$

where μ denotes each mapping in the support of Ω, and the double vertical bars separate the domain and range of Ω. We can encode this $\mathbb{N}[X]$-relation via the following plain relation Ω^r:

$$\Omega^r = \left[\begin{array}{c|cccc} \text{?person} & ?\Sigma\otimes 1\odot & ?\Sigma\otimes 2\odot & ?\Sigma\otimes 3\odot & ?\Sigma\otimes 4\odot \\ \hline \text{GM} & s_{12} & & s_{11} & s_{13} \\ \text{OT} & s_{22} & & s_{21} & s_{23} \\ \text{OT} & & s_{24} & s_{21} & s_{23} \end{array}\right],$$

where the single vertical rule separates query solutions from provenance elements. The variables on the right side of the line tell us how to decode this table into $\mathbb{N}[X]$ polynomials. We illustrate this process in the following:

(1) For each row of the table, multiply all non-unbound values on the right of the vertical rule and record them under the variable $?\Sigma\otimes$. The result is the following table.

$$\left[\begin{array}{c||c} \text{?person} & ?\Sigma\otimes \\ \hline \text{GM} & s_{12} \otimes s_{11} \otimes s_{13} \\ \text{OT} & s_{22} \otimes s_{21} \otimes s_{23} \\ \text{OT} & s_{24} \otimes s_{21} \otimes s_{23} \end{array}\right].$$

(2) Sum the values on the right of the double vertical bar grouped by the query answers (on the left), and record the results in variable $?\Sigma$. The result is the following table:

$$\left[\begin{array}{c||c} \text{?person} & ?\Sigma \\ \hline \text{GM} & s_{12} \otimes s_{11} \otimes s_{13} \\ \text{OT} & (s_{22} \otimes s_{21} \otimes s_{23}) \oplus (s_{24} \otimes s_{21} \otimes s_{23}) \end{array}\right].$$

By the distributivity of the product over the sum, we observe that the polynomials under the variable $?\Sigma$ in the table above are equivalent to the polynomials in the $\mathbb{N}[X]$-relation Ω (recall that the elements of $\mathbb{N}[X]$ are equivalence classes for the polynomial expressions). In the following, we formalize this process of encoding Ω into a regular relation Ω^r via query rewriting.

Definition 4.2 (Semiring decoding). Assume two disjoint sets of variables $\mathbf{V}_S$ and $\mathbf{V}_P$, called solution and provenance variables. Let Ω^r be a set of mappings, where for each mapping $\mu \in \Omega$ it holds that $\text{dom}(\mu) \subseteq \mathbf{V}_S \cup \mathbf{V}_P$. The *semiring decoding* of Ω^r is the $\mathbb{N}[X]$-relation Ω defined as follows:

$$\Omega(\mu') = \bigoplus_{\mu \in \Omega,\, \mu' = \mu|_{\mathbf{V}_S}} \left(\bigotimes_{x \in \text{dom}(\mu) \cap \mathbf{V}_P} \mu(x) \right).$$

Determining a query rewriting $Q \mapsto Q^r$ such that the rewritten query Q^r produces a regular relation Ω^r, requires the use of reification to obtain the sources associated to each relationship.

Definition 4.3 (Semiring query rewriting). Let Reify be a reification scheme, and Q be a query that consists of a combination of the empty basic graph pattern and triple patterns with the operators SELECT, UNION, AND, FILTER, and BIND. Then, the *semiring rewriting* for query Q is the query rewriting that produces the query Q^r resulting from the following modifications in query Q:

(1) each empty basic graph pattern is replaced by the query {} BIND (1 AS $?\Sigma\otimes i\odot$),
(2) each triple pattern (s, p, o) is replaced by the query $\text{Reify}(p(s, o), ?\Sigma\otimes i\odot)$,
(3) the variables $?\Sigma\otimes i\odot$ from steps 1 and 2 are added to the select clause.

The value of i is chosen consistently (in steps 1 and 2) to guarantee a fresh variable $?\Sigma\otimes i\odot$ in each replacement.

Example 4.4. The application of the semiring query rewriting on the query of Example 4.1 produces the following query:

```
SELECT ?person ?Σ⊗1⊙ ?Σ⊗2⊙ ?Σ⊗3⊙ ?Σ⊗4⊙
WHERE ( Reify(occupation(?person, writer), ?Σ⊗1⊙) UNION
          Reify(occupation(?person, novelist), ?Σ⊗2⊙)) AND
        Reify(gender(?person, female), ?Σ⊗3⊙) AND
        Reify(awarded(?person, NPL), ?Σ⊗4⊙).
```

By simple inspection, one can verify that the rewritten query in Example 4.4 returns the table that encodes the $\mathbb{N}[X]$-relation that results from evaluating the original query. It can be shown by induction that the semiring rewriting simulates the annotated algebra. More formally, given a reification scheme Reify; a graph G^r annotated using Reify; a query Q consisting of a combination of the empty basic graph pattern and triple patterns with the operators SELECT, UNION, AND, FILTER, and BIND; a query Q^r resulting from the semiring rewriting on query Q; and the $\mathbb{N}[X]$-relation Ω' that results from the semiring decoding of the output of Q^r in graph G^r, it holds that $\Omega' = (\!|Q|\!)_{\text{Reify}^{-1}(G)}$.

4.3 Provenance for non-monotonic operators

In this section we present our approach for query rewriting to compute how-provenance for queries with non-monotonic SPARQL operators. We build upon the spm-semirings framework. We first point out the challenges of encoding polynomials in an spm-semiring using regular relations. We then propose an encoding that overcomes those difficulties, and a query rewriting to obtain such an encoding.

4.3.1 Challenges to encode spm-semiring polynomials. The encoding from $\mathbb{N}[X]$-relations to regular relations defined for monotonic operators fails for non-monotonic operations such as OPTIONAL or DIFF. We illustrate this issue by means of the following example.

Example 4.5. Consider the $\mathbb{N}[X]$-graph G depicted in Figure 1 and the following query asking for females without an occupation.

$$Q = \text{SELECT } \texttt{?person} \text{ WHERE } (\texttt{?person}, \texttt{gender}, \texttt{female}) \text{ DIFF } (\texttt{?person}, \texttt{occupation}, \texttt{?occup})$$

Under the annotated algebra, query Q returns the $\mathbb{N}[X]$-relation

$$\Omega = \left[\begin{array}{c||c} \texttt{?person} & \Omega(\mu) \\ \hline \texttt{GM} & s_{11} \ominus s_{12} \\ \texttt{OT} & s_{21} \ominus (s_{22} \oplus s_{24}) \end{array}\right].$$

Since the polynomials in Ω have the operator $\ominus$, we cannot use Definition 4.3. A query rewriting for a query P_1 DIFF P_2 requires returning the provenance of solutions μ of P_1, and optionally, the provenance of solutions of P_2 that are compatible with μ. This observation yields the following query Q^r

$$\begin{array}{ll} \text{SELECT} & \texttt{?person } ?\Sigma\ominus1\odot\ ?\Sigma\ominus2\odot \\ \text{WHERE} & \text{Reify}(\texttt{gender}(\texttt{?person}, \texttt{female}), ?\Sigma\ominus1\odot) \\ & \quad \text{OPTIONAL Reify}(\texttt{occupation}(\texttt{?person}, \texttt{?occup}), ?\Sigma\ominus2\odot), \end{array}$$

which returns the following table:

$$\Omega^r = \left[\begin{array}{c|cc} \texttt{?person} & ?\Sigma\ominus1\odot & ?\Sigma\ominus2\odot \\ \hline \texttt{GM} & s_{11} & s_{12} \\ \texttt{OT} & s_{21} & s_{22} \\ \texttt{OT} & s_{21} & s_{24} \end{array}\right].$$

If we apply the naive decoding that computes a polynomial for each row, and then sums the polynomials of the same solution, we get the following $\mathbb{N}[X]$-relation:

$$\Omega' = \left[\begin{array}{c||c} \texttt{?person} & ?\Sigma \\ \hline \texttt{GM} & s_{11} \ominus s_{12} \\ \texttt{OT} & (s_{21} \ominus s_{22}) \oplus (s_{21} \ominus s_{24}) \end{array}\right].$$

The polynomials for the mapping $\mu_{\texttt{OT}} = \{\texttt{?person} \mapsto \texttt{OT}\}$ in the resulting table and in Ω are not equivalent because $\ominus$ is not left-distributive over $\oplus$, i.e., $x \ominus (y \oplus z) \neq (x \ominus y) \oplus (x \ominus z)$. This shows that we cannot treat the monotonic operator $\otimes$ and the non-monotonic operator $\ominus$ in the same way. We therefore need a more sophisticated encoding to distinguish them.

4.3.2 *Encoding for spm-semirings polynomials.*

Hitherto, to encode an $\mathbb{N}[X]$-relation Ω with a set of mappings Ω^r, we have distinguished the variables that are part of the solution mapping from the variables that codify the provenance. We first define the set of provenance variables in a more formal fashion.

Definition 4.6. Provenance variables are named with sequences defined by the following generation rule:

$$\mathbf{V}_P \coloneqq \Sigma \mid \oplus \mid \otimes \mid \ominus \mid \odot \mid (\Sigma \mid \oplus N \mid \otimes N \mid \ominus(1 \mid 2))\mathbf{V}_P,$$

where N denotes a positive natural number. We assume that the set $\mathbf{V}$ (of variables) is the disjoint union of the sets $\mathbf{V}_S$ and $\mathbf{V}_P$, where variables in $\mathbf{V}_S$ are called *solution variables.*

As Example 4.5 suggests, the names of the provenance variables encode information. Therefore, it is necessary to identify the components of a variable, such as the prefix. Given a provenance variable $?x$, we write $\mathbf{V}_{?x}$ to denote the subset of $\mathbf{V}_P$ consisting of the variables with prefix x, e.g., $?\oplus\ominus \in \mathbf{V}_{?\oplus}$. Given a set of mappings Ω, a variable $?x$, a set of variables V, and a term c, the expression $\Omega|_{?x=c}$ denotes the set of mappings $\{\mu \in \Omega \mid \mu(?x) = c\}$, $\Omega(?x)$ denotes the set of terms $\{\mu(?x) \mid \mu \in \Omega\}$, and $\Omega[V]$ denotes the set of mappings $\{\mu|_V \mid \mu \in \Omega\}$.

Syntactically speaking, polynomial expressions in $\mathbb{N}[X]$ are trees where internal nodes are operations (i.e., $\oplus$, $\otimes$, and $\ominus$), and leafs are 0, 1, or polynomial variables. For instance, the polynomial expression $(x \oplus y) \otimes 1$ corresponds to the following tree:

$$\begin{tikzcd} & & \otimes \arrow[dl, dash, "1"'] \arrow[dr, dash, "2"] & \\ & \oplus \arrow[dl, dash, "1"'] \arrow[dr, dash, "2"] & & 1 \\ x & & y & \end{tikzcd}$$

Edges are labeled with 1 and 2 to distinguish branches.

To encode this tree as a table, we use a column for each node, and identify these columns with the variables $?\otimes$, $?\otimes1\oplus$, $?\otimes1\oplus1\odot$, $?\otimes1\oplus2\odot$, and $?\otimes2\odot$. The sequence of symbols in variable names identifies the node by its path from the root – including nodes and edge labels. For instance, since the path to node y is the sequence $(\otimes, 1, \oplus, 2)$, we encode this node with the variable $?\otimes1\oplus2\odot$. The symbol $\odot$ at the end of the variable indicates that node y is a leaf. In the same spirit, the tree for the sub-expression $x \oplus y$ is codified with the variables $?\oplus$, $?\oplus1\odot$, and $?\oplus2\odot$. The process of taking a sub-expression in a table is defined as follows.

Definition 4.7 (Subtables). Given a finite set of mappings Ω, a finite set of provenance variables V, and a variable $?x\circ \in V$ that consists of a prefix x and a symbol $\circ \in \{\Sigma, \oplus, \otimes, \ominus, \odot\}$, the *subtable* of Ω over $?x\circ$, denoted $\text{subt}(\Omega, ?x)$ is the set of mappings that results from replacing prefix $?x$ by $?$ in the set of mappings $\Omega[\mathbf{V}_{?x}]$ and excluding mappings μ where $?\circ$ is unbound. Similarly, the *subtable* of V over $?x\circ$, denoted $\text{subt}(V, ?x\circ)$, is the set of variables that results from replacing the prefix $?x$ by $?$ in $V \cap \mathbf{V}_{?x\circ}$. We write $\text{subt}((\Omega, V), ?x\circ)$ to denote the pair $(\text{subt}(\Omega, ?x\circ), \text{subt}(V, ?x\circ))$.

Example 4.8. The extraction of the expression $x \oplus y$ from the table that codifies the aforementioned expression $(x \oplus y) \otimes z$ is implemented with the function subt as follows:[5]

$$\text{subt}(\left[\begin{array}{ccccc} ?\otimes & ?\otimes1\oplus & ?\otimes1\oplus1\odot & ?\otimes1\oplus2\odot & ?\otimes2\odot \\ \hline a & b & x & y & 1 \end{array}\right], ?\otimes1\oplus)) = \left[\begin{array}{ccc} ?\oplus & ?\oplus1\odot & ?\oplus2\odot \\ \hline b & x & y \end{array}\right].$$

Recall that our goal is to define a query rewriting such that the rewritten queries produce sets of mappings that can be decoded as $\mathbb{N}[X]$-relations. We call the *provenance table* the output of such a rewritten query. Function subt is defined to retrieve sub-expressions of a polynomial in a provenance table. Since function subt needs to record the set of variables that define the columns of the table (recall that sets of mappings do not record the variables that are unbound), provenance tables are defined as pairs (Ω, V) where Ω is the set of mappings and V is a finite set of variables including all variables in Ω. Additionally, provenance tables have to satisfy a particular structure, defined recursively as combinations of provenance tables.

[5]Terms a and b can be ignored for now (they are explained in detail in Section 4.3.3).

Definition 4.9 (Provenance tables). The set of *provenance tables* is the minimal set of mappings defined recursively as follows:

(1) A pair $(\{\mu\}, \{?\odot\})$ where μ is a mapping, and $\mathrm{dom}(\mu) \subseteq \{?\odot\}$ is a provenance table.
(2) A pair (Ω^t, V), where Ω^t is a set of mappings for $x \in \{\oplus, \otimes, \ominus\}$, V is a finite subset of $\mathbf{V}_{?x}$ and V includes the provenance variables $\{?x\}$, $?\mathbf{V}_{x1y}$, and $\mathbf{V}_{?x2z}$ with $y, z \in \{\Sigma, \oplus, \otimes, \ominus, \odot\}$, is a provenance table if either $\Omega^t = \{\mu_\emptyset\}$ or Ω^t satisfies the following conditions: (a) Each mapping $\mu \in \Omega^t$ satisfies $\mathrm{dom}(\mu) \subseteq V$; (b) there exists a term c such that $\mu(?x) = c$ for each $\mu \in \Omega^t$; (c) for $k \in \{1, 2\}$ and $u \in \{y, z\}$, there exists a disjoint union $\Omega^t[\mathbf{V}_{?x\mathrm{k}u}] = \Omega^t_1 \cup \cdots \cup \Omega^t_n$ such that, for $1 \leq i \leq n$, $\mathrm{subt}((\Omega^t_i, V), ?x\mathrm{k}))$ is a provenance table.
(3) A pair (Ω^t, V), where Ω^t is a set of mappings, V is a finite subset of $\mathbf{V}_{?\Sigma}$ that includes variable $?\Sigma$ and a variable $?\Sigma y$ with $y \in \{\Sigma, \oplus, \otimes, \ominus, \odot\}$, is a provenance table if either $\Omega^t = \{\mu_\emptyset\}$ or Ω^t satisfies the following conditions: (a) Each mapping $\mu \in \Omega^t$ satisfies $\mathrm{dom}(\mu) \subseteq V$; (b) there exists a disjoint union $\Omega^t[\mathbf{V}_{?\Sigma y}] = \Omega^t_1 \cup \cdots \cup \Omega^t_n$ such that, for $1 \leq i \leq n$, $\mathrm{subt}((\Omega^t_i, V), ?\Sigma)$ is a provenance table.

We are now ready to define the main notion of our query rewriting, that is, how we decode provenance tables as $\mathbb{N}[X]$-relations.

Definition 4.10 (Spm-semiring decoding). Given a provenance table T, the *spm-semiring decoding* of T, denoted $\langle\!\langle T \rangle\!\rangle$, is the provenance polynomial defined recursively as follows:

(1) If T is a pair $(\{\mu\}, \{?\odot\})$, where μ is a mapping, then $\langle\!\langle T \rangle\!\rangle = \mu(?\odot)$ if $\odot \in \mathrm{dom}(\mu)$; otherwise $\langle\!\langle T \rangle\!\rangle = 0$.
(2) If T is a pair (Ω^t, V), where the common prefix of variables in V is $?\circ$ for $\circ \in \{\oplus, \ominus\}$, and V includes variables $?\circ$, $?\circ 1y$, and $?\circ 2z$ with $y, z \in \{\Sigma, \oplus, \otimes, \ominus, \odot\}$, then $\langle\!\langle T \rangle\!\rangle = \langle\!\langle T_1 \rangle\!\rangle \circ \langle\!\langle T_2 \rangle\!\rangle$ where $T_1 = \mathrm{subt}(T, ?\circ 1y)$ and $T_2 = \mathrm{subt}(T, ?\circ 2z)$.
(3) If T is a pair (Ω^t, V), where the common prefix of variables in V is $?\otimes$, and V includes the variables $?\otimes$, $?\otimes 1y$ and $?\otimes 2z$ with $y, z \in \{\Sigma, \oplus, \otimes, \ominus, \odot\}$, then $\langle\!\langle T \rangle\!\rangle = \bigoplus_{a \in \Omega^t(?\otimes 1y),\, b \in \Omega^t(?\otimes 2z)} \langle\!\langle T_1 \rangle\!\rangle \otimes \langle\!\langle T_2 \rangle\!\rangle$ where $T_1 = \mathrm{subt}(T|_{?\otimes 1y = a}, ?\otimes 1y)$ and $T_2 = \mathrm{subt}(T|_{?\otimes 2z = b}, ?\otimes 2z)$.
(4) If T is a pair (Ω^t, V), where the common prefix of variables in V is $?\Sigma$, and V includes the variables $?\Sigma$ and $?\Sigma y$ with $y \in \{\Sigma, \oplus, \otimes, \ominus, \odot\}$, then $\langle\!\langle T \rangle\!\rangle = \bigoplus_{a \in \Omega^t(?\Sigma y)} \langle\!\langle T_a \rangle\!\rangle$ where $T_a = \mathrm{subt}(T|_{?\Sigma y = a}, ?\Sigma y)$.

Let Ω^r be a set of mappings, and V be a finite set of variables that include all variables in Ω^r. Given a mapping $\mu \in \Omega^r[\mathbf{V}_S]$, let $\Omega^r(\mu)$ be the set of mappings $\mu' \in \Omega^r$ that agree with μ in the domain $\mathbf{V}_S$ (i.e., for every variable $?x \in \mathbf{V}_S$, either $\mu(?x) = \mu'(x)$ or $?x$ does not belong to the domains of μ and μ'). If $T_\mu = \Omega^r(\mu)[V \cap \mathbf{V}_P]$ is a provenance table for each mapping $\mu \in \Omega^r[\mathbf{V}_S]$, then the *spm-semiring decoding* for Ω^r is the $\mathbb{N}[X]$-relation Ω where $\Omega(\mu) = \langle\!\langle T_\mu \rangle\!\rangle$ if $\mu \in \Omega^r[\mathbf{V}_S]$, and $\Omega(\mu) = 0$ if $\mu \notin \Omega^r[\mathbf{V}_S]$.

Example 4.11. Consider the $\mathbb{N}[X]$-graph G in Figure 1, the following query Q (from Example 4.5), and the set of mappings Ω^r:

$$Q = \textsc{select}\ \texttt{?person}\ \textsc{where}\ (\texttt{?person}, \texttt{gender}, \texttt{female})\ \textsc{diff}\ (\texttt{?person}, \texttt{occupation}, \texttt{?occup}),$$

$$\Omega^r = \left[\begin{array}{c|cccc} \texttt{?person} & ?\ominus & ?\ominus 1\odot & ?\ominus 2\Sigma & ?\ominus 2\Sigma\odot \\ \hline \texttt{GM} & a & s_{11} & c & s_{12} \\ \texttt{OT} & b & s_{21} & d & s_{22} \\ \texttt{OT} & b & s_{21} & d & s_{24} \end{array}\right],$$

where a, b, c, and d can be ignored for now – they are explained in detail in the next section. This set of mappings is interpreted as the following $\mathbb{N}[X]$-relation:

$$\Omega = \left[\begin{array}{c||c} \texttt{?person} & \\ \hline \texttt{GM} & \langle\!\langle \left[\begin{array}{cccc} ?\ominus & ?\ominus 1\odot & ?\ominus 2\Sigma & ?\ominus 2\Sigma\odot \\ \hline a & s_{11} & c & s_{12} \end{array}\right] \rangle\!\rangle \\ \texttt{OT} & \langle\!\langle \left[\begin{array}{cccc} ?\ominus & ?\ominus 1\odot & ?\ominus 2\Sigma & ?\ominus 2\Sigma\odot \\ \hline b & s_{21} & d & s_{22} \\ b & s_{21} & d & s_{24} \end{array}\right] \rangle\!\rangle \end{array}\right].$$

We can see that Ω is the $\mathbb{N}[X]$-relation obtained by the $\mathbb{N}[X]$-annotated SPARQL algebra using the definition of $\langle\!\langle \cdot \rangle\!\rangle$. We next describe how the provenance polynomial for `OT` is decoded:

$$\langle\!\langle \left[\begin{array}{c} ?\odot \\ \hline s_{21} \end{array}\right] \rangle\!\rangle \ominus \langle\!\langle \left[\begin{array}{cc} \Sigma & \Sigma\odot \\ \hline d & s_{22} \\ d & s_{24} \end{array}\right] \rangle\!\rangle = s_{21} \ominus \left(\langle\!\langle \left[\begin{array}{c} ?\odot \\ \hline s_{22} \end{array}\right] \rangle\!\rangle \oplus \langle\!\langle \left[\begin{array}{c} ?\odot \\ \hline s_{24} \end{array}\right] \rangle\!\rangle \right) =$$
$s_{21} \ominus (s_{22} \oplus s_{24})$.

4.3.3 *Query rewriting for non-monotonic queries.*

We now present a query rewriting that is compatible with our spm-semiring decoding, that is, it produces a new query Q^r that evaluates into a regular relation Ω^r that encodes the $\mathbb{N}[X]$-relation result of the original query Q under the $\mathbb{N}[X]$-annotated SPARQL algebra.

The set of mappings Ω^r used in Example 4.11 to encode an $\mathbb{N}[X]$-relation Ω includes the terms a and b which are used to identify mappings associated to the same solution. We can associate mappings such that $\mu(?\ominus) = a$ to solution `GM` and the mappings where $\mu(?\ominus) = b$ to solution `OT`. We define these terms a and b as a function β of the values of the mapping μ and the variable $?\ominus$. For instance, $a = \beta(\mu, ?\ominus) =$ `<http://example.org/Σ?person=GM&occupation=writer>`. We next define such a function β.

Given a provenance variable $?w$ and a list of variables $X = (?x_1, \ldots, ?x_n)$ in lexicographic order, we write $\mathtt{B}(X, ?w)$ to denote the operation $\textsc{bind}\,(E\ \textsc{as}\ ?w)$ where E is the following built-in SPARQL expression supported by standard triple stores:
`IRI(CONCAT("http://example.org/`w`?",`A_1`,"&",...,"&",`A_n`))`,
where for $1 \leq i \leq n$, A_i is the expression:
`COALESCE(CONCAT("&`x_i`=",ENCODE_FOR_URI(xsd:string(?`x_i`))),"")`.
Given a mapping μ such that $\mathrm{dom}(\mu) \subseteq X$, we define $\beta(\mu, ?w)$ as the result of evaluating $\mathtt{B}(X, ?w)$ on mapping μ. Given a set of variables $W = \{?w_1, \ldots, ?w_m\}$, we write $\mathtt{B}(X, W)$ as an abbreviation for $\mathtt{B}(X, ?w_1) \cdots \mathtt{B}(X, ?w_m)$.

The rewriting of operator $\textsc{diff}$ poses some challenges that require additional work. Example 4.5 suggests that a query $Q =$

P_1 DIFF P_2 is rewritten as $Q^r = (\textsc{select}\ \mathrm{dom}(P_1) \cup X\ \textsc{where}\ P_1^r\ \textsc{optional}\ P_2^r)$, where P_1^r and P_2^r are the rewritten queries for P_1 and P_2, and X is the set of provenance variables added in queries P_1^r and P_2^r. The key of this rewriting is that the provenance variables generated in query P_2^r are projected, but not the values of the solutions of P_2. However in general, this rewriting does not work because it can still return bindings from solutions of P_2. For instance, if $\mathrm{dom}(P_1) = \mathrm{dom}(P_2) = \{?x, ?y\}$, $\mathrm{supp}((\!|P_1|\!)_G) = \{?x \mapsto a\}$, and $\mathrm{supp}((\!|P_2|\!)_G) = \{?x \mapsto a, ?y \mapsto b\}$, then the aforementioned rewritten query Q^r returns an annotated mapping $\{?x \mapsto a, ?y \mapsto b\}$ instead of mapping $\{?x \mapsto a\}$. To solve this issue we substitute each "problematic" variable $?y$ with a fresh variable $?y'$, and add a condition to check that the values of variables $?y$ and $?y'$ are compatible. We call *variable substitution*, a partial function $\nu : \mathbf{V}_S \to \mathbf{V}_S$, and we write $\nu(Q)$ to denote the result of substituting in Q every variable $?x \in \mathrm{dom}(\nu)$ by $\nu(?x)$. We write C_ν to denote the formula $\bigwedge_{?x \in \mathrm{dom}(\nu)} (\neg\, \mathrm{bound}(?x) \vee \neg\, \mathrm{bound}(\nu(?x)) \vee ?x = \nu(?x))$. Hence, the aforementioned query Q is rewritten into Q^r, where pattern P_1^r OPTIONAL P_2^r is substituted with pattern P_1^r $\textsc{optional}_{C_\nu}$ $\nu(P_2^r)$, and ν substitutes the problematic variables with fresh variables (observe that both patterns are identical if there are no problematic variables). The variable substitution ν depends on determining what variables are problematic. A variable $?x$ is said to be *problematic* if there exists a graph G such that there is a mapping $\mu_1 \in \mathrm{supp}((\!|P_1|\!)_G)$ where $?x$ is unbound. We cannot find the exact set of problematic variables because, in general, the problem of determining whether or not a variable is potentially unbound is undecidable, but a sound approximate method can be used to discard variables that are always bound, called *strongly bound* [5].

Now we are ready to present the query rewriting to annotate answers of monotonic and non-monotonic SPARQL queries with how-provenance. The algorithm assumes a SPARQL reification scheme Reify as parameter. To ease readability, we assume that Reify reifies binary relationship queries, even though the query rewriting can be generalized to relationship queries of higher arity. Additionally, we define the query rewriting for queries written in terms of the operators SELECT, AND, UNION, DIFF, FILTER, and BIND, called *primitive* in what follows. Queries with the operators MINUS, OPTIONAL, $\textsc{optional}_\varphi$, and $\textsc{diff}_\varphi$ can be normalized to use the aforementioned set of operators (see Section 4.1).

Definition 4.12. Let Q, P_1, and P_2 be SPARQL queries consisting of empty basic graph patterns, triple patterns, and primitive operators. Let $z\mathbf{V}_P$ be a prefix of a provenance variable, ν a variable substitution that substitutes with fresh variables the variables in $\mathrm{dom}(P_1) \cap \mathrm{dom}(P_2)$ that are not strongly bound in P_1, and Reify a reification scheme. Then, the rewritten query for Q and variable z, denoted $\alpha(Q, z)$, is defined recursively as follows:

(1) If Q is an empty basic graph pattern, then $\alpha(Q, z)$ is the query $\{\}\ \mathsf{B}(1, z\odot)$.
(2) If Q is a triple pattern (s, p, o), then $\alpha(Q, z)$ is the query $\mathsf{Reify}(p(s, o), z\Sigma\odot)\ \mathsf{B}(\mathrm{dom}(Q), z\Sigma)$.
(3) If Q is P_1 AND P_2, then $\alpha(Q, z)$ is the query $(\alpha(P_1, z\otimes 1)\ \textsc{and}\ \alpha(P_2, z\otimes 2))\ \mathsf{B}(\mathrm{dom}(Q), z\otimes)$.
(4) If Q is P_1 UNION P_2, then $\alpha(Q, z)$ is the query $(\alpha(P_1, z\oplus 1)\ \textsc{union}\ \alpha(P_2, z\oplus 2))\ \mathsf{B}(\mathrm{dom}(Q), z\oplus)$.
(5) If Q is P_1 DIFF P_2, then $\alpha(Q, z)$ is the query
$$\begin{array}{l}(\ \textsc{select}\ W \\ \quad \textsc{where}\ (\alpha(P_1, z\ominus 1)\ \textsc{optional}_{C_\nu}\ \alpha(\nu(P_2), z\ominus 2\Sigma)) \\ \quad \mathsf{B}(\mathrm{dom}(Q), \{z\ominus, z\ominus 2\Sigma\})\end{array},$$
such that W is the set of variables $(\mathrm{dom}(\alpha(P_1, z\ominus 1)) \cup \mathrm{dom}(\alpha(\mu(P_2), z\ominus 2\Sigma))) \setminus \mathrm{dom}(\mu(P_2))$.
(6) If Q is SELECT W WHERE P_1, then $\alpha(Q, z)$ is the query $\textsc{select}\ W \cup (\mathrm{dom}(\alpha(P_1, z\Sigma)) \setminus \mathrm{dom}(P_1)) \cup \{z\Sigma\}$ $\textsc{where}\ \alpha(P_1, z\Sigma))\ \mathsf{B}(\mathrm{dom}(Q), z\Sigma)$
(7) If Q is P_1 FILTER φ, then $\alpha(Q, z)$ is the query $\alpha(P_1, z)\ \textsc{filter}\ \varphi$.
(8) If Q is P_1 BIND $(E$ AS $x)$, then $\alpha(Q, z)$ is the query $\alpha(P_1, z)\ \textsc{bind}\ (E\ \textsc{as}\ x)$.

We write $\alpha(Q)$ as an abbreviation of $\alpha(Q, ?)$.

The correctness of the query rewriting is given as follows.

THEOREM 4.13. *Let G be a graph, Q be a SPARQL query consisting of empty basic graph patterns, triple patterns, and the primitive operators, $Q^r = \alpha(Q)$ be the rewritten query for Q according to Definition 4.17, then $[\![Q]\!]_G$ encodes the $\mathbb{N}[X]$-relation $(\!|Q|\!)_G$.*

Example 4.14. Let Q be the query of Example 4.5 asking for females without occupation. The rewritten version of query Q is:

$$\begin{aligned}\alpha(Q) = {} & \textsc{select}\ \mathtt{?person}\ ?\Sigma \\ & \qquad ?\Sigma\ominus\ ?\Sigma\ominus 1\Sigma\ ?\Sigma\ominus 1\Sigma\odot\ ?\Sigma\ominus 2\Sigma\ ?\Sigma\ominus 2\Sigma\Sigma\ ?\Sigma\ominus 2\Sigma\Sigma\odot \\ & \textsc{where}\ \alpha((\mathtt{?person}, \mathtt{gender}, \mathtt{female})\ \textsc{diff} \\ & \qquad (\mathtt{?person}, \mathtt{occupation}, \mathtt{?occup}), ?\Sigma) \\ & \qquad \mathsf{B}(\{\mathtt{?person}\}, ?\Sigma) \\ = {} & \textsc{select}\ \mathtt{?person}\ ?\Sigma \\ & \qquad ?\Sigma\ominus\ ?\Sigma\ominus 1\Sigma\ ?\Sigma\ominus 1\Sigma\odot\ ?\Sigma\ominus 2\Sigma\ ?\Sigma\ominus 2\Sigma\Sigma\ ?\Sigma\ominus 2\Sigma\Sigma\odot \\ & \textsc{where}\ (\alpha((\mathtt{?person}, \mathtt{gender}, \mathtt{female}), ?\Sigma\ominus 1)\ \textsc{optional} \\ & \qquad \alpha((\mathtt{?person}, \mathtt{occupation}, \mathtt{?occup}), ?\Sigma\ominus 2\Sigma), ?\Sigma) \\ & \qquad \mathsf{B}(\{\mathtt{?person}\}, \{?\Sigma\ominus, ?\Sigma\ominus 1\Sigma, ?\Sigma\}).\end{aligned}$$

We omit the remaining recursive steps of the query rewriting due to space limitations.

4.4 Provenance for Aggregate Queries

The query rewriting described in Definition 4.17 does not include aggregate queries, which however are of great interest for OLAP (OnLine Analytical Processing) applications [17, 30, 31]. We therefore extend the algebra covered by Geerts [20] by defining a lineage-based provenance model for aggregate queries. Lineage annotations break the property of commutation with homomorphisms. In [2] it is suggested, however, that respecting this property comes at the expense of encoding the resulting aggregate values in the solutions of queries as expressions that can be evaluated via a homomorphism. This complefixies the output the query rewriting. Providing how-provenance annotations that commute with homomorphisms, and that at the same time do not require to encode the query results, is an interesting avenue for future work.

Definition 4.15. The lineage-spm-semiring is the structure $(\mathbb{N}[X], \oplus, \otimes, \ominus, \delta, 0, 1)$ where $(\mathbb{N}[X], \oplus, \otimes, \ominus, 0, 1)$ is a provenance spm-semiring and δ is a unary operation. Let F be the aggregate expression $(g_1(x_1)\ \textsc{as}\ y_1) \ldots (g_n(x_n)\ \textsc{as}\ y_n)$. The *lineage-annotated algebra* is the result of extending the $\mathbb{N}[X]$-annotated SPARQL algebra as follows: $(\!|\textsc{select}\ W\ F\ \textsc{where}\ P\ \textsc{group by}\ W|\!)_G(\mu) = \delta\left(\bigoplus_{\mu'|_W=\mu|_W} (\!|P|\!)_G(\mu')\right)$.

Example 4.16. Let $G = \{(a, \texttt{:price}, 1) \mapsto k_1, (a, \texttt{:price}, 2) \mapsto k_2\}$ be an $\mathbb{N}[X]$-graph recording products and transaction prices, and Q be the following SPARQL query asking for the total taxes (10%) paid for each product:

```
SELECT ?product (sum(?tax) AS ?totalTax)
WHERE ((?product,:price,?price) BIND (?price*0.1 AS ?tax))
GROUP BY ?product.
```

The evaluation of Q in G has a unique answer $\mu = \{\texttt{?product} \mapsto a, \texttt{?totalTax} \mapsto 0.3\}$. Intuitively and according to the proposed semantics, the provenance of a mapping μ resulting from an aggregation is defined in terms of the provenance of the mappings μ' that are aggregated to generate μ. This intuition is formalized by Definition 4.15, from which it follows that $(\!|Q|\!)_G(\mu) = \delta(\bigoplus_{\mu'|_{\texttt{?product}}=\mu} (\!|P|\!)_G(\mu')) = \delta(k_1 \oplus k_3)$ where the operator δ keeps track of the sources that participate in a single aggregation, P is the pattern in the WHERE clause of the query above.

The rewriting of aggregate queries requires a special regard. The main idea behind the rewriting of an aggregate query $Q = (\textsc{select}\ W\ (f(x)\ \textsc{as}\ y)\ \ \textsc{where}\ \ P)$ is that for each answer μ of Q we have to compute both, the value c of the aggregate function $f(x)$ and the polynomial p that explains the mapping μ. We follow the approach described in [21] for relational algebra, which calculates the aggregate function and the polynomial that explains μ separately and then combines both results with a natural join to ensure that values of the aggregate function and polynomials correspond to the same mapping μ. However, this approach cannot directly be applied to SPARQL because compatible mappings are joinable, which may result in wrong combinations of aggregate values and provenance polynomials. It follows that we need a join based on a more strict compatibility notion that accepts equal values instead of compatible values. This compatibility is called *cautious compatibility* in [37] and formalized as follows. Let Q_1 and Q_2 be SPARQL queries, ν be a variable substitution for the variables in $\text{dom}(Q_1) \cap \text{dom}(Q_2)$ that are not strongly bound in Q_1 and Q_2. Then, we write $Q_1\ \textsc{and}_=\ Q_2$ to denote the SPARQL query $\textsc{select}\ W\ \textsc{where}\ (Q_1\ \textsc{and}\ \nu(Q_2))\ \textsc{filter}\ \varphi_\nu$, where $W = \text{dom}(Q_1) \cup \text{dom}(Q_2)$ and φ_ν is the formula $\bigwedge_{?x \in \text{dom}(\nu)} (?x = \nu(?x) \vee (\neg\, \text{bound}(?x) \wedge \neg\, \text{bound}(\nu(?x)))$. Definition 4.17 formalizes the extension of our query rewriting for aggregate queries.

Definition 4.17. Let W be a set of variables; A be an aggregate function expression; P be a SPARQL query consisting of empty basic graph patterns, triple patterns, the primitive operators, and aggregate functions; z be a provenance variable prefix; and Q be the query $(\textsc{select}\ W\ A\ \ \textsc{where}\ P)$, then $\alpha(Q, z)$ is the query
$\alpha((\textsc{select}\ W\ \ \textsc{where}\ P), z)\ \ \textsc{and}_=\ \ (\textsc{select}\ W\ A\ \textsc{where}\ P)$.

5 EVALUATION

We evaluate the viability of SPARQLprov to compute how-provenance by measuring the runtime overhead w.r.t. the original query. The evaluation is carried out on Virtuoso and Fuseki, two standard RDF/SPARQL engines, using real and synthetic data. In Section 5.1, we evaluate the response times of each phase of SPARQLprov (rewriting, execution, and decoding) as well as the impact of the reification scheme used to model the data. In Section 5.2, we conduct a scalability analysis and consider queries with aggregates. Section 5.3 evaluates our approach on a real-world dataset, i.e., Wikidata [42]. Finally, in Section 5.4 we compare our solution with two state-of-the-art approaches for how-provenance.

All experiments were conducted on a machine with an AMD EPYC 7281 16-Core Processor, 256GB of RAM, and an 8 TB HDD disk. We use Virtuoso (v. 7.2.5.1) and Fuseki (v. 3.17.0 with TDB1 as storage driver) to test SPARQLprov. For all experiments, we set a timeout of 300 seconds, and report the average response time of the queries over 5 executions after a warm-up phase. Our implementation as well as the experimental data are available at https://relweb.cs.aau.dk/sparqlprov/.

5.1 Algorithm phases and reification scheme

For this evaluation we use Watdiv [1], a state-of-the-art performance benchmark for RDF/SPARQL engines. Watdiv offers a data generator to build synthetic datasets of different sizes, and 20 query templates each containing 10 instantiated queries. The query templates are divided in four categories; linear queries (L), star queries (S), snowflake-shaped queries (F), and complex queries (C). We also introduce 5 new query templates to account for non-monotonic queries (O). These queries were constructed by surrounding one of the triple patterns in the linear queries with an OPTIONAL clause.

5.1.1 Rewriting and decoding. As explained in Section 4, SPARQLprov operates in three stages: (i) query rewriting, (ii) execution of the rewritten query on the host engine, and (iii) decoding of the answer into how-provenance polynomials. Our experimental results show that phase (ii) dominates the other phases in terms of runtime. In particular the runtime of phase (i) is negligible, whereas for phase (iii) runtime is mainly determined by the result size. Based on these observations, we focus the remainder of our runtime analysis on phase (ii), i.e., the execution of the provenance query itself.

5.1.2 Query execution. Query runtime is obviously influenced by the used reification scheme. Hence, we measure query execution times on the three most popular reification schemes: named graphs, Wikidata, and standard reification. We also hold a copy of the data without reification to determine the overhead caused by the reification schemes. We then measure the execution times of several versions of the benchmark queries on the data:

- P the provenance query created by our rewriting approach (as defined in Definition 4.12) on the respective reification scheme,
- R the query without provenance rewriting on the respective reification scheme, and
- B the original query executed on the original (non-reified) data.

As illustrated in the figure below, we can break down the execution time of a provenance query P into three components (i) the baseline share ($\frac{B}{P}$) of executing the original query B on non-reified data, (ii) the reification overhead ($\frac{R-B}{P}$) of executing query R over the reified data, and (iii) the provenance overhead ($\frac{P-R}{P}$) of computing query P over the reified data.

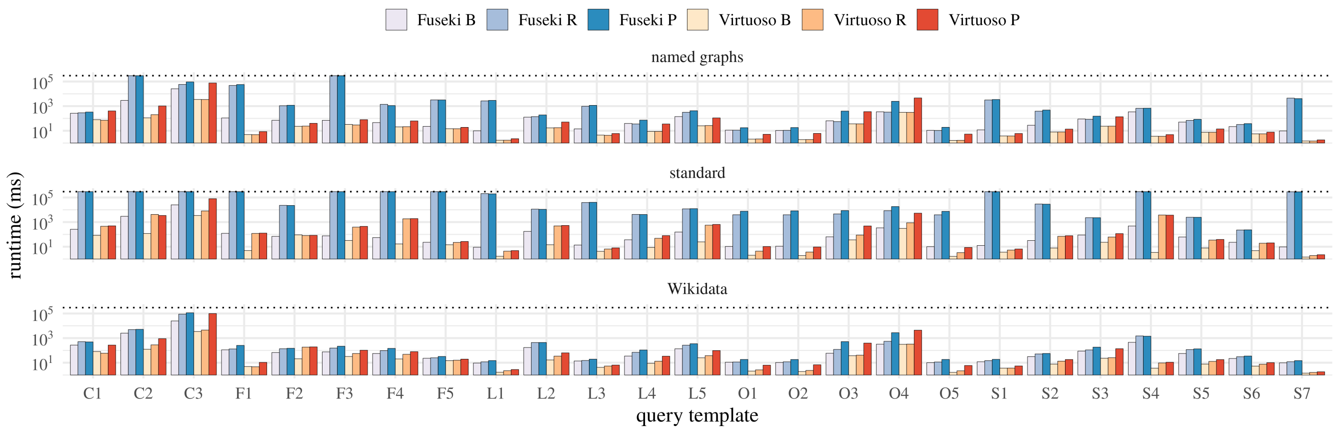


Figure 2: Watdiv query execution times per reification scheme and query template. The top dotted line represents the timeout.

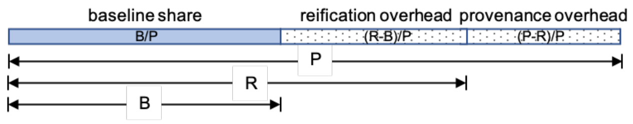


Figure 2 shows the average query runtimes of the provenance queries P per reification scheme grouped by template on the 100M triples Watdiv dataset. We first observe that Virtuoso outperforms Fuseki in most of the queries, and that the performance gap is more pronounced for the standard reification scheme regardless of the query template. We also highlight that reification alone already causes a significant overhead in comparison to executing the original query on the non-reified dataset. On average the reification overhead amounts to 23% on Virtuoso and 47% on Fuseki.

If we distinguish between queries with monotonic and non-monotonic operators, we find that reification induces a bigger overhead for queries with non-monotonic operators only (on average 27% on Virtuoso and 56% on Fuseki) than it does compared to those with monotonic operators (on average 7% on Virtuoso and 19% on Fuseki). This happens because non-monotonic operators translate into more complex rewritten queries.

We also find that the effect of reification is particularly pronounced for the standard reification scheme. We observe that on average, 82% of the provenance queries' runtime corresponds to the overhead induced by the reification scheme in Fuseki (52% for Virtuoso). In contrast, the average overhead for the named graphs (0.3% Virtuoso, 44% Fuseki) and the Wikidata (18% Virtuoso, 30% Fuseki) reification schemes are lower. This can be explained by the fact that reification introduces additional triples that complexify query execution. Besides, standard reification introduces more triples than any of the other schemes.

Although in general Virtuoso outperforms Fuseki, there are some cases where the provenance query is slower on Virtuoso. This can be explained by the choice of sub-optimal plans by Virtuoso's query planner. For instance, for complex query C3, Virtuoso sub-optimally pushes down the calculation and addition of the provenance variables before the joins. This leads to many superfluous calculations; many of the intermediate rows extended with provenance variables are pruned by the subsequent joins. As for S3, Virtuoso resorts to a query plan containing a costly nested query with a large number of intermediate results.

All in all, the average provenance overhead for Virtuoso is 40% for queries without non-monotonic operators and 73% with non-monotonic operators, compared to provenance overheads of 14% and 52% for Fuseki. In the latter case, the provenance overhead is comparably low because, unlike Virtuoso, Fuseki adds new columns to intermediate results very efficiently. Our experiments did not reveal any particular influence of the query type on performance in general. We rather found that performance is influenced by result sizes and selectivity of the triple patterns as well as whether non-monotonic operators are involved or not – such operators are significantly more expensive to evaluate.

5.2 Scalability and aggregation

To evaluate the runtime overhead of our approach as data size increases, we use the TPC-H benchmark (v. 2.18.0). We generate datasets using the TPC-H data generator with scale factors of the form $10^{i/4-2}$ for $i = \{1, 2, \ldots, 8\}$. We wrote a conversion tool to convert TPC-H data to RDF. Each row of a table in a TPC-H dataset corresponds to an n-ary relation in RDF, which we reify according to the *direct mapping scheme* in Table 1. The numbers of triples in the resulting RDF datasets range from 1.2M to 123M.

The TPC-H query generator comes with 22 query templates with variables that are randomly instantiated to produce concrete queries. We omit query templates 4, 13, 15, 17, 18, 20, 21, and 22 from the benchmark because they include features outside the SPARQL fragment studied in this paper.

For each SQL query template we generate an equivalent SPARQL query template (*base aggregate*) as well as a corresponding version without aggregation (*base non-aggregate*). For each of these two queries, we then apply our rewriting approach and generate corresponding provenance queries: *provenance aggregate* and *provenance non-aggregate*. By considering two groups of base queries (base aggregate and base non-aggregate) we can study the overhead of computing provenance with and without aggregates. We generate 10 queries for each query template and each scale factor.

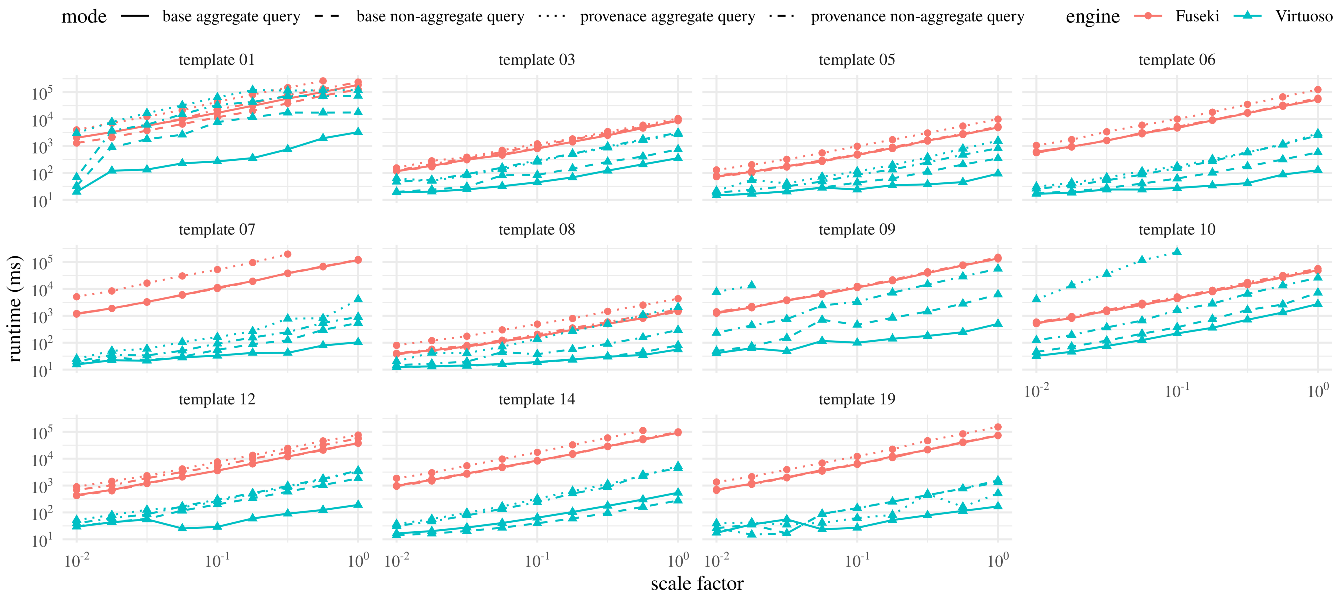


Figure 3: TPC-H execution times per query template as scale factor grows.

Figure 3 depicts the evolution of the runtime of the aforementioned queries in Fuseki and Virtuoso as the scale factor increases. We highlight that runtime grows linearly with data size, and that Virtuoso consistently outperforms Fuseki (less clearly in the case of template 1 though). Even though Virtuoso achieves overall lower runtimes than Fuseki, the overhead of aggregation in the base queries is more pronounced for Virtuoso. For provenance queries, this overhead is less significant (templates 3, 5–7, 12, 14, and 19).

In general, the runtime overhead of provenance is insensitive to the scale factor and the number of answers in Fuseki. For Virtuoso, in contrast, this overhead depends largely on the query template. For example, some of the provenance queries without aggregation from template 1 reach Virtuoso's maximal number of answers (2^{20}) from scale factor $10^{6/4-2}$, putting a ceiling in query runtime and leading to a constant overhead w.r.t. the base query. In other cases (e.g., template 19), the overhead fluctuates because the number of answers does not increase monotonically with the scale factor for some queries. This suggests that Virtuoso's performance is more sensitive to the number of answers than Fuseki's.

On average, the provenance overheads are 88% and 63% for aggregate and non-aggregate queries in Virtuoso, and 16% and 0% in Fuseki (excluding the queries with timeouts). The overhead is significantly larger for aggregate queries due to the joins introduced by our rewriting (see Definition 4.17).

5.3 Real data

We evaluate SPARQLprov on the RDF Wikidata dump from 27-01-2020 that contains 942M relationships encoded with the Wikidata reification scheme (see Table 1). We generated three types of queries: *star*, *union*, and *minus*. The star queries were randomly generated based on the procedure described in [27]. To generate queries with UNION and MINUS operators, we first created two star queries Q_1 and Q_2 such that Q_2 is the result of renaming the variables that are not in the select clause of Q_1 with fresh names. These two patterns produce the union query Q_1 UNION Q_2 and the minus query Q_1 MINUS Q_2. For each of the aforementioned query types, we generate 20 queries and apply our rewriting approach to generate the corresponding provenance queries.

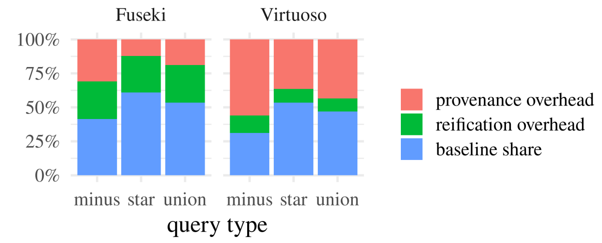


Figure 4: Wikidata time overhead.

Our evaluation results (Figure 4) show that our approach induces an average provenance overhead of 36% for star queries, 43% for union queries, and 56% for minus queries in Virtuoso (Fuseki: 12% star, 19% union, 30% minus). The provenance overhead is larger for the non-monotonic minus queries. This is not surprising since we compute explanations for, on average, 3599 bindings in the support of the minus queries – even though the queries do not have actual results. The general provenance overhead is smaller for Fuseki, because this engine extends the intermediate results with provenance variables more efficiently than Virtuoso. These results show that our approach can compute how-provenance on top of a standard engine on large real-world datasets with a reasonable runtime overhead.

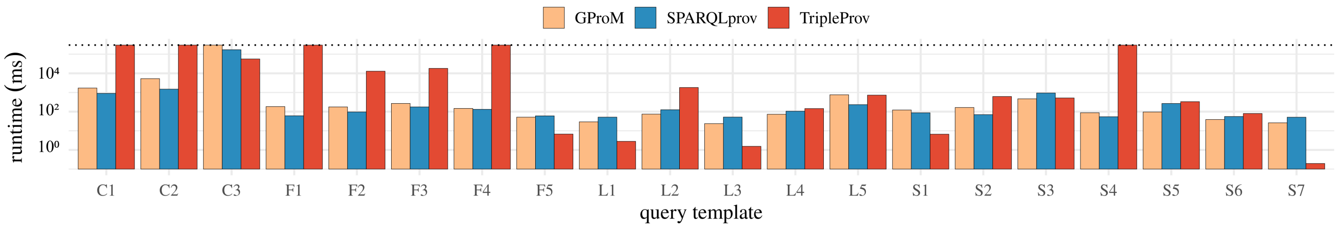


Figure 5: Watdiv query execution times per engine and query template. The top dotted line represents the timeout.

5.4 Comparison with other systems

We also compare SPARQLprov with TripleProv [43], a specialized how-provenance solution for SPARQL, and GProM [4], a system to compute how-provenance for SQL queries.

5.4.1 Runtime evaluation. Since TripleProv is a pure main-memory storage engine, these experiments were executed using an in-memory file system to guarantee a fair comparison. For GProM, we translated the WatDiv queries into SQL and modeled the RDF data using one table per predicate as this translation yields the best performance for GProM (we use PostgreSQL as backend). We limit our evaluation to monotonic queries, i.e., queries without OPTIONAL and DIFF, since GProM and TripleProv build upon semirings and therefore do not produce sound polynomials for such queries [20]. For SPARQLprov we use Virtuoso with named graphs reification.

Figure 5 shows the measured runtimes. In general, SPARQLprov outperforms TripleProv except for queries C3, F5, L3, and S7. C3 is a star query without constants, which benefits from TripleProv's architecture – based on star patterns called *molecules*. For the remaining queries, the difference in runtime is at most 2ms, and can be explained by the fact that TripleProv, unlike SPARQLprov, lacks the runtime overhead associated with query parsing, planning, rewriting, and HTTP communication. Despite this advantage, TripleProv times out for queries C1, C2, F1, F4, and S1, which require to retrieve information from multiple molecules and looping through many intermediate results.

Overall SPARQLprov outperforms GProM. Otherwise, the runtime difference is less than 0.5s despite the efficient RDF representation used by GProM, where predicates are omitted, and tables names identify predicate relations.

These results show that SPARQLprov competes with state-of-the-art approaches in terms of runtime while being system-agnostic.

5.4.2 Qualitative evaluation. We also compare the annotations computed by SPARQLprov and those obtained from GProM and TripleProv. Our first example query is obtained by instantiating template L3 from the WatDiv benchmark:

$$\text{SELECT } ?v_0 \text{ WHERE } ((?v_0, \texttt{:likes}, ?v_1) \text{ AND } (?v_0, \texttt{:subscribes}, \texttt{:Website2579}))$$

SPARQLprov returns a table with 12 rows and 5 columns with the following structure:

$?v_0$	$?\Sigma$	$\Sigma\otimes$	$\Sigma\otimes 1\odot$	$\Sigma\otimes 2\odot$
:User50045	a	b	u_1	u_3
:User50045	a	c	u_2	u_3

Due to space limitations we present the rows for a single answer, and denote the values of provenance variables by u_1, u_2, a, b, c. This table is interpreted as the polynomial expression $(u_1 \otimes u_3) \oplus (u_2 \otimes u_3)$, which is the same answer provided by GProM. TripleProv returns an equivalent factorized expression: $(u_1 \oplus u_2) \otimes u_3$.

Consider also query O5 as introduced in Section 5.1.2:

$$\text{SELECT } ?v_0 \text{ WHERE } ((?v_0, \texttt{:subscribes}, \texttt{:Website2579}) \text{ OPTIONAL } (?v_0, \texttt{:likes}, ?v_1))$$

The rewritten query produced by SPARQLprov returns the following table (due to space restrictions some columns are omitted):

$?v_0$	$\Sigma\oplus 1\otimes 1\odot$	$\Sigma\oplus 1\otimes 2\odot$	$\Sigma\oplus 2\ominus 1\odot$	$\Sigma\oplus 2\ominus 2\Sigma\odot$
:User50045	u_3	u_1		
:User50045	u_3	u_2		
:User50045			u_3	u_1
:User50045			u_3	u_2

The answer $\{?v_0 \mapsto \texttt{:User50045}\}$ is thus annotated with the polynomial expression $u_3 \otimes u_1 \oplus u_3 \otimes u_1 \oplus u_3 \ominus (u_1 \oplus u_2)$, which captures the semantics of OPTIONAL. On the other hand, GProM and TripleProv produce the expression $u_3 \otimes (u_1 \oplus u_2)$, which is incorrect because it does not adhere to the property of commutation with homomorphisms (Definition 3.2).

6 CONCLUSIONS

We have proposed an approach to compute how-provenance polynomials for SPARQL query results via query rewriting. This makes our approach directly applicable to any engine with a SPARQL interface. Unlike existing solutions, our approach can compute how-provenance explanations that commute with homomorphisms for monotonic and non-monotonic queries. It can also provide lineage annotations for queries with aggregates.

As shown by our experiments, our query rewriting layer deployed on standard RDF/SPARQL engines can compute provenance for large real-world datasets such as Wikidata. Furthermore, our evaluation on the TPC-H benchmark suggests that our approach is viable for computing provenance explanations in an OLAP setting where queries with aggregates are very common.

ACKNOWLEDGMENTS

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensen Foundation.

REFERENCES

[1] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *The Semantic Web - ISWC 2014 Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 8796. Springer, 197–212.

[2] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for Aggregate queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*. ACM, 153–164.

[3] Renzo Angles and Claudio Gutiérrez. 2016. The Multiset Semantics of SPARQL Patterns. In *The Semantic Web – ISWC Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 9981. 20–36.

[4] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Engineering Bulletin* 41, 1 (2018), 51–62.

[5] Carlos Buil Aranda, Marcelo Arenas, and Óscar Corcho. 2011. Semantics and Optimization of the SPARQL 1.1 Federation Extension. In *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 6644. Springer, 1–15.

[6] Argyro Avgoustaki, Giorgos Flouris, Irini Fundulaki, and Dimitris Plexousakis. 2016. Provenance Management for Evolving RDF Datasets. In *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC Proceedings (Lecture Notes in Computer Science)*, Vol. 9678. Springer, 575–592.

[7] Dave Beckett. 2004. RDF/XML Syntax Specification (Revised). W3C Recommendation. http://www.w3.org/TR/rdf-syntax-grammar/

[8] Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. 2018. Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371). *Dagstuhl Reports* 8, 9 (2018), 29–111.

[9] Spencer Chang. 2018. Scaling Knowledge Access and Retrieval at Airbnb. AirBnB Medium Blog. https://medium.com/airbnb-engineering/scaling-knowledge-access-and-retrieval-at-airbnb-665b6ba21e95 (Accessed on 16/09/2021).

[10] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474.

[11] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)* 25, 2 (2000), 179–227.

[12] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. 2012. Provenance for SPARQL Queries. In *The Semantic Web – ISWC Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 7649. Springer, 625–640.

[13] Deepika Devarajan. 2017. Happy Birthday Watson Discovery. IBM Cloud Blog. https://www.ibm.com/cloud/blog/announcements/happy-birthday-watson-discovery (Accessed on 16/09/2021).

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

[15] Xin Luna Dong, Xiang He, Andrey Kan, Xian Li, Yan Liang, Jun Ma, Yifan Ethan Xu, Chenwei Zhang, Tong Zhao, Gabriel Blanco Saldana, Saurabh Deshpande, Alexandre Michetti Manduca, Jay Ren, Surender Pal Singh, Fan Xiao, Haw-Shiuan Chang, Giannis Karamanolakis, Yuning Mao, Yaqing Wang, Christos Faloutsos, Andrew McCallum, and Jiawei Han. 2020. AutoKnow: Self-Driving Knowledge Collection for Products of Thousands of Types. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*. ACM, 2724–2734.

[16] Johannes Frey, Kay Müller, Sebastian Hellmann, Erhard Rahm, and Maria-Esther Vidal. 2019. Evaluation of Metadata Representations in RDF Stores. *Semantic Web* 10, 2 (2019), 205–229.

[17] Luis Galárraga, Kim Ahlstrøm, Katja Hose, and Torben Bach Pedersen. 2018. Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets. In *ISWC*. 547–565.

[18] Garima Gaur, Arnab Bhattacharya, and Srikanta Bedathur. 2020. How and Why is An Answer (Still) Correct? Maintaining Provenance in Dynamic Knowledge Graphs. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management*. ACM, 405–414.

[19] Floris Geerts and Antonella Poggi. 2010. On database query languages for K-relations. *Journal of Applied Logic* 8, 2 (2010), 173–185.

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

[21] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 25th International Conference on Data Engineering, ICDE*. IEEE Computer Society, 174–185.

[22] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 31–40.

[23] Harry Halpin and James Cheney. 2014. Dynamic Provenance for SPARQL Updates. In *The Semantic Web – ISWC Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 8796. Springer, 425–440.

[24] Steve Harris and Ady Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/

[25] Olaf Hartig. 2009. Querying Trust in RDF Data with tSPARQL. In *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC Proceedings (Lecture Notes in Computer Science)*, Vol. 5554. Springer, 5–20.

[26] Nicolas Heist and Heiko Paulheim. 2019. Uncovering the Semantics of Wikipedia Categories. In *The Semantic Web - ISWC Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 11778. Springer, 219–236.

[27] Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. 2016. Querying Wikidata: Comparing SPARQL, Relational and Graph Databases. In *The Semantic Web – ISWC, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 9982. 88–103.

[28] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. 2011. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. In *Proceedings of the 20th International Conference on World Wide Web, WWW (Companion Volume)*. ACM, 229–232.

[29] Wei Hu, Honglei Qiu, JiaCheng Huang, and Michel Dumontier. 2017. BioSearch: A Semantic Search Engine for Bio2RDF. *Database - The Journal of Biological Databases and Curation* (2017).

[30] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2014. Towards Exploratory OLAP Over Linked Open Data - A Case Study. In *BIRTE*. 114–132.

[31] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2016. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In *ISWC*. 341–359.

[32] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. 2016. Semantics and Expressive Power of Subqueries and Aggregates in SPARQL 1.1. In *Proceedings of the 25th International Conference on World Wide Web, WWW*. ACM, 227–238.

[33] Roman Kontchakov and Egor V. Kostylev. 2016. On Expressibility of Non-Monotone Operators in SPARQL. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR*. AAAI Press, 369–379.

[34] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.

[35] Natalya Fridman Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale knowledge graphs: lessons and challenges. *Commun. ACM* 62, 8 (2019), 36–43.

[36] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 16:1–16:45.

[37] Axel Polleres. 2007. From SPARQL to Rules (and back). In *Proceedings of the 16th International Conference on World Wide Web, WWW*. ACM, 787–796.

[38] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2034–2037.

[39] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. 2012. On directly mapping relational databases to RDF and OWL. In *Proceedings of the 21st World Wide Web Conference 2012, WWW*. ACM, 649–658.

[40] Saurabh Shrivastava. 2017. Bring rich knowledge of people, places, things and local businesses to your apps. BingBlogs. https://blogs.bing.com/search-quality-insights/2017-07/bring-rich-knowledge-of-people-places-things-and-local-businesses-to-your-apps/ (Accessed on 16/09/2021).

[41] Amit Singhal. 2012. Introducing the Knowledge Graph: things, not strings. Google Blog. https://www.blog.google/products/search/introducing-knowledge-graph-things-not/ (Accessed on 16/09/2021).

[42] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledge Base. *Commun. ACM* 57, 10 (2014), 78–85.

[43] Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. 2014. TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store. In *23rd International World Wide Web Conference, WWW*. ACM, 455–466.