Incremental View Maintenance for SPARQL Queries: Adapting the Counting Algorithm

DORE STAQUET, Data Science Institute, Hasselt University and transnational University of Limburg, Diepenbeek, Belgium and Data & Analytics Competence Center, Flemish Institute for Technological Research, Mol, Belgium
BART BUELENS, Data & Analytics Competence Center, Flemish Institute for Technological Research, Mol, Belgium
JAN VAN DEN BUSSCHE, Data Science Institute, Hasselt University and transnational University of Limburg, Diepenbeek, Belgium

The counting algorithm is a classic approach to incremental view maintenance for queries on relational data. We adapt this algorithm to SPARQL queries on RDF (Resource Description Framework) datasets. In the decentralized Web, data are customarily stored in RDF, which is linked data that can be queried using SPARQL. The efficiency of retrieval of query results can be improved through maintaining views on the data incrementally as the underlying data change. SPARQL operators give rise to heterogeneous sets of solution mappings and involve multi-set semantics. We develop a theory of SPARQL algebra on annotated sets of solution mappings and show how to handle operators traditionally perceived as difficult, including LeftJoin, Diff, and Minus. We discuss the implementation of our methods—based on the counting algorithm—and assess the feasibility using data and queries from the Berlin SPARQL Benchmark. Performant incremental view maintenance will be instrumental when querying the decentralized web at scale.

CCS Concepts: • **Information systems** → *Resource Description Framework (RDF)*; **Query optimization**; **Query languages**.

Additional Key Words and Phrases: Linked Data, Semantic Web, Web3.0

1 Introduction

A materialized view is the stored result of a possibly complex query, posed against a database [14]. When updates are made to the database, the query result may change, and the view needs to be updated. This is the classic task of *incremental view maintenance*. In the relational database setting, incremental view maintenance is well understood, see for example the compendium volume [19] and monograph [9]. In this article, we propose an incremental view maintenance algorithm for SPARQL over RDF graphs.

Recent developments combining concepts from the semantic web and the decentralized web [22, 44]—sometimes referred to as Web3.0—instigate renewed interest in incremental view maintenance for RDF datasets. Personal Online Data stores (PODs) are proposed to be personal data vaults containing private data from individuals. These ideas are formalized in the Solid Project [37] and are continued as Linked Web Storage [48]. PODs are made available through HTTP as authenticated linked data containers with RDF resources. Data retrieval from

Authors' Contact Information: Dore Staquet, Data Science Institute, Hasselt University and transnational University of Limburg, Diepenbeek, Flanders, Belgium and Data & Analytics Competence Center, Flemish Institute for Technological Research, Mol, Flanders, Belgium; e-mail: dore.staquet@uhasselt.be; Bart Buelens, Data & Analytics Competence Center, Flemish Institute for Technological Research, Mol, Flanders, Belgium; e-mail: bart.buelens@vito.be; Jan Van den Bussche, Data Science Institute, Hasselt University and transnational University of Limburg, Diepenbeek, Belgium; e-mail: jan.vandenbussche@uhasselt.be.

ACM 1559-114X/2026/2-ART
https://doi.org/10.1145/3796549

PODs can occur directly through sharing and accessing of the relevant IRIs [47], or, in a more powerful fashion, through the querying of PODs using SPARQL [39].

In a world in which every citizen would control their own personal data POD [31], data volumes could grow large, with frequent updates to some parts of the data. Many different actors or parties could seek access to parts of peoples' data via informed consent mechanisms [12]. For example, in the region of Flanders, in Belgium, the We Are project [27, 46] envisages a POD for every citizen to govern their personal health data. In such scenarios, for efficiency reasons, it would be sensible for PODs to materialize different views, catering for different data consumers [10, 31, 41]. Such views will need to be incrementally maintained to keep them actual under data updates in the PODs. To illustrate, consider an RDF graph that describes patients, their diagnoses and prescribed medications, as well as the doctors treating them. One view may be defined by a query that retrieves the diagnoses and medications of a group of relevant patients–for example those residing in a particular hospital. When the graph is updated, such as when a new medication is added for a patient, the view should be updated accordingly. Recomputing the view in its entirety after each update is inefficient; instead, incremental view maintenance updates the view using only the changes that occurred.

In database systems, the materialization of views is typically motivated by performance considerations, particularly for query optimization. This applies to relational querying as well as RDF graph querying. Several authors have applied materialized RDF views for optimizing analytical SPARQL queries (see Ibragimov et al. [24] and the references therein). So far, materialized views in the SPARQL context have only been considered with fixed schemas, derived from basic graph patterns (BGP) or data cubes, or defined by construct-queries (see section 2). Support for views based on more general graph patterns would be generally useful, yet it has received little attention. Note that every type of SPARQL query: select-, construct-, or ask-query, is based on a general graph pattern, as are the *subqueries* of such queries.

In this article, we propose an incremental view maintenance algorithm for SPARQL graph patterns over RDF graphs.

While there are certainly commonalities with relational data querying (SQL), the SPARQL setting presents new problems due to *heterogeneity* of query results. The result of a graph pattern is not a relation over a fixed schema, but a set of *solution mappings*. When LeftJoin and Union operators are used in the pattern, the solution mappings may not all be defined on the same set of variables. SPARQL engines commonly display such partially defined solution mappings as relational tuples with null values in undefined attributes. Relational algebra under this representation does not behave in line with the SPARQL algebra on heterogeneous sets of solution mappings. For example, in SQL, the tuple $\{a\colon 4, b\colon 5, c\colon \text{null}\}$ does not join with the tuple $\{a\colon \text{null}, b\colon \text{null}, c\colon 6\}$. In contrast, in SPARQL, the solution mapping $\{?a\colon 4, ?b\colon 5\}$ trivially joins with the solution mapping $\{?c\colon 6\}$. Hence, incremental view maintenance for SPARQL is not simply solved by applying incremental view maintenance for relational queries with nulls.

We address the following research questions:

- (RQ1) How can incremental view maintenance techniques from the relational setting be adapted to SPARQL, given the multiset semantics and heterogeneous solution mappings that arise from SPARQL's algebra?
- (RQ2) A graph pattern in SPARQL is an expression composed of BGPs using the operators of the SPARQL algebra [3, 21]: Join, LeftJoin (known as OPTIONAL), Filter, Union, Minus, and Project (known as SELECT). How can these be handled in an incremental framework?
- (RQ3) To what extent can the adapted algorithm be feasibly implemented and evaluated on realistic RDF data and SPARQL queries?

We do not directly consider EXISTS or NOT EXISTS filters, because their semantics as given in the current standard can be ill-defined [25]. However, such filters can be expressed equivalently in terms of Semijoins or Diff

operations, which we do support. Our contributions pave the way for future work on aggregations and property paths, SPARQL features requiring specialized techniques.

Our approach consists of the following steps:

(i) A key aspect of the counting algorithm in the relational context is that it works with delta relations in which tuples are *annotated* with an integer value. These values can be negative, to express deletions. Regarding SPARQL, an algebra on annotated sets of solution mappings was already developed by Geerts et al. [15] for general "spm-semiring" annotations. We will use an equivalent formulation of their definitions, extended to express multi-set semantics.

(ii) Semiring annotations alone do not yet yield incremental view maintenance. To this end, we prove a set of algebraic equivalences, identified as $\mathcal{E}_1$ by Geerts et al., over annotated sets of solution mappings.[1] These are crucial to adapt the counting algorithm to patterns built from BGPs using Join, Union, Filter, and Project operators on annotated sets of solution mappings.

(iii) Finally and importantly, we adapt the counting algorithm's treatment of negation to deal with operators Diff, Minus, and LeftJoin.

The remainder of this article is organized as follows. Section 2 discusses related work in a semantic web context, in particular on incremental view maintenance. Section 3 develops the SPARQL algebra over annotated sets of solution mappings. Section 4 presents our algorithm. Section 5 discusses the results of our implementation based on a series of experiments. Section 6 concludes this article discussing challenges and directions for future work.

2 Related work

In the relational data context, the standard approach to incremental view maintenance is to provide, for each relational algebra operator O working on relational data D, an algorithm $\mathcal{A}_O$ that takes three inputs: D; the current result $O(D)$; and an update set Δ. The algorithm then produces $O(D')$ as output, where D' denotes D updated by Δ. Of course, the idea here is to design $\mathcal{A}_O$ so that it takes less time than computing $O(D')$ from scratch, i.e., from D' only. This approach dates back to the seminal work by Blakeley, Larson and Tompa [8], who did this for selections, projections and joins.

When the algorithm $\mathcal{A}_O$ is itself expressed in relational algebra, we obtain so-called *propagation rules* for operator O. By composing together the propagation rules for a view Q expressed in relational algebra, we obtain a more complex relational algebra expression, called the *incremental expression*, that does incremental maintenance for Q. Propagation rules for relational algebra were developed by Qian and Wiederhold [32]. Improved, so-called "minimal," rules were given later by Griffin et al. [18].

The LeftJoin (OPTIONAL) operator is very important for SPARQL applications; in the relational setting it is known as left *outer join*. Outer join is expressible in relational algebra, so propagation rules can be obtained from those of the relational algebra. As reported by Larson and Zhou, however, the incremental expressions become very complex, so they proposed more dedicated algorithms as an alternative to propagation rules [28]. Griffin and Kumar devise propagation rules for outer join in relational algebra extended with outer join and semijoin, but do not report on implementation, performance [16].

Propagation rules for the relational algebra with multiset semantics, including duplicate elimination, were proposed, on a theoretical level, by Griffin and Libkin [17]. Working with bag semantics as well is the algorithm behind DBToaster, a state-of-the-art system in incremental view maintenance in the relational setting [26]. Their algorithm uses higher-order propagation and is implemented using query compilation, achieving fast performance. Incremental view maintenance remains an active research topic in database systems [38].

[1] Geerts et al. did not prove these equivalences, but only proved conversely that, if we want $\mathcal{E}_1$ to hold, the annotation values must form a semiring.

To the best of our knowledge, there is little prior work specifically on incremental view maintenance for SPARQL patterns. Menendez et al. [29] consider a fragment of SPARQL construct-queries, geared to defining "linksets," a kind of data mappings. The schematic heterogeneity of general SPARQL patterns does not manifest itself in linksets, which have a fixed triple schema. Furthermore, their considered fragment has no OPTIONAL operator, and their incremental maintenance algorithm is specific to linksets. A related work originating from the same team [45] has a more general algorithm that maintains relational-to-RDF mappings. The language considered there is not SPARQL, however, but is essentially that of relational conjunctive queries. Schmedding [36] comes closest to our work in that he develops propagation rules for a limited set of operators over sets of solution mappings. He does not treat BGPs as a unit; he does not consider multiset semantics; he considers Diff (which he calls minus) but not SPARQL 1.1 standard Minus; he does not discuss implementation or experimental evaluation.

RDF graphs can be classified as graph data or, in earlier terminology, as semistructured data. Research on incremental view maintenance for graph queries has frequently focused on languages that are essentially relational (e.g., basic graph patterns), while the graph-specific context motivates the development of novel techniques. A similar observation holds for streaming settings, encompassing both graph-structured and relational data. Tailored solutions for incremental view maintenance in such specific settings are available for streaming RDF [49]. There is also work addressing regular path queries and recursion, which likewise necessitate specialized techniques. We consider these contributions to be beyond the scope of this paper. Instead, we focus on a salient feature of SPARQL patterns, namely schematic heterogeneity by OPTIONAL matching.

As explained in the Introduction, our approach relies on SPARQL algebra over annotated sets of solution mappings, often referred to as *semiring provenance*. While we emphasize semiring provenance as a basis to obtain a provably correct incremental maintenance algorithm, there is interesting work on the efficient computation of semiring provenance for SPARQL [4, 23]. That work is complementary to ours and indeed could be applied when developing a high-performance implementation of our method.

Specifically in the context of Solid [37], Taelman and Verborgh [40] develop an efficient link traversal SPARQL query processing engine for decentralized knowledge graphs like constituted by Solid. Another approach to query optimization for SPARQL queries on PODs is query rewriting, in which an original query gets rewritten to a potentially more efficient one [42]. Neither of these approaches consider views, nor their incremental maintenance, and hence are complementary in overall query optimization strategies. Optimization of SPARQL query execution over RDF data generally has received interest lately, for example Pang et al. [30] propose a novel BGP-tree to optimize query execution. Closely related yet again complementary to our work is the ESPRESSO framework [34], in which the authors propose a framework for searching a POD-based decentralized web architecture. They construct indices for PODs to optimize querying, while we propose to introduce materialized views for PODs.

3 SPARQL algebra on annotated sets of mappings

We recall the formal definition of annotated sets of solution mappings, and the semantics of SPARQL operators thereon. We assume two disjoint, infinitely enumerable universes of *nodes* and *variables*.[2]

3.1 SPARQL algebra

We begin by recalling the semantics of the SPARQL operators, which is defined in terms of solution mappings [21]. Our development follows the widely cited exposition by Arenas et al. [3].

[2]For simplicity of presentation, and following similar treatments, e.g., [25], we do not distinguish here between IRIs, blank nodes, and literals, and just refer to them collectively as nodes. Making our treatment fully conformant to the RDF and SPARQL specifications poses no additional difficulties.

A *solution mapping* (or just mapping for short) is a function $\mu : X \to U$, where X is a finite set of variables and U the universe of nodes. The set X is called the *schema* of μ, and we denote it by $\text{vars}(\mu)$.[3]

Two mappings μ_1 and μ_2 are *compatible*, denoted by $\mu_1 \sim \mu_2$, if they agree on their common variables; formally, if $\mu_1(x) = \mu_2(x)$ for every $x \in \text{vars}(\mu_1) \cap \text{vars}(\mu_2)$. When $\mu_1 \sim \mu_2$, their union $\mu_1 \cup \mu_2$ is a well-defined solution mapping.

Viewing variables as attributes in the relational database model [1], a solution mapping corresponds to a tuple in the relational model. A set of solution mappings that all have the same schema, corresponds to a relation over that schema. Here, we will work with sets of solution mappings that do not necessarily have the same schema.

We now recall the SPARQL algebra on sets of solution mappings [21]. Let Ω, Ω_1, and Ω_2 denote sets of solution mappings.

Filter Let e be a predicate on solution mappings, i.e., a function from solution mappings to {true, false}. The idea is that e serves as a filter condition. Now $\text{Filter}(e, \Omega) := \{\mu \in \Omega \mid e(\mu) = \text{true}\}$.

Join We define $\text{Join}(\Omega_1, \Omega_2) := \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \text{ and } \mu_1 \sim \mu_2\}$.

Diff We define $\text{Diff}(\Omega_1, \Omega_2) := \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 : \mu_1 \nsim \mu_2\}$.

LeftJoin We define $\text{LeftJoin}(\Omega_1, \Omega_2) := \text{Join}(\Omega_1, \Omega_2) \cup \text{Diff}(\Omega_1, \Omega_2)$.

Union Simply $\text{Union}(\Omega_1, \Omega_2) := \Omega_1 + \Omega_2$.

Minus We define $\text{Minus}(\Omega_1, \Omega_2) := \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 : \text{vars}(\mu_1) \cap \text{vars}(\mu_2) \neq \emptyset \Rightarrow \mu_1 \nsim \mu_2\}$.

Project Let X be a finite set of variables. Now $\text{Project}(\Omega, X) := \{\mu|_{\text{vars}(\mu) \cap X} \mid \mu \in \Omega\}$. Here, the notation $\mu|_Z$, for any subset Z of $\text{vars}(\mu)$, denotes the restriction of μ to the variables in Z.

In the SPARQL surface syntax, Join is denoted by a dot, and LeftJoin by the OPTIONAL keyword. Project corresponds to SELECT [21].

We will use the following symbols to denote the above operators:

operator:	Filter(e)	Join	Diff	LeftJoin	Union	Minus	Project(X)
symbol:	σ_e	$\bowtie$	$\ominus$	$\mathbin{⟕}$	$+$	$\boxminus$	π_X

Whenever, below, we use the standard minus ($-$) symbol with sets, we mean the standard set difference (which neither Diff nor Minus are).

While there are certainly commonalities with relational data querying (SQL), the SPARQL setting presents new problems due to *heterogeneity* of query results. The result of a graph pattern is not a relation over a fixed schema, but a set of *solution mappings*. When Optional (LeftJoin) and Union operators are used in the pattern, the solution mappings need not be defined on the same set of variables. Using the example introduced in section 1, assume there is a patient Alice who is diagnosed with diabetes and treated by Dr. Carol; patient Bob is diagnosed with hypertension and his treating physician is unknown, Figure 1 visually represents this data. Consider a query which retrieves each patient's diagnosis, and if available, their treating physician:

```
SELECT ?patient ?diagnosis ?physicianName
WHERE {
  ?patient :hasDiagnosis ?diagnosis .
  OPTIONAL {
    ?patient :treatedBy ?physician .
    ?physician :name ?physicianName .
  }
}
```

Evaluating this query produces the results in Table 1. The first solution mapping binds all three variables, wheras the second leaves ?physician undefined. SPARQL engines typically display such partially defined solution

[3]The usual notation for $\text{vars}(\mu)$ is $\text{dom}(\mu)$ [3, 21]; we find $\text{vars}(\mu)$ less confusing.

mappings as relational tuples with null values in undefined attributes. Relational algebra under this representation does not behave in line with the SPARQL algebra on heterogeneous sets of solution mappings. Now consider another query in which the doctors are matched with all patients they assist:

```
SELECT ?physicianName ?patient ?diagnosis
WHERE {
    ?physician a :doctor .
    ?physician :name ?physicianName .
    OPTIONAL {
        ?patient :treatedBy ?physician .
        ?patient :diagnosis ?diagnosis .
    }
}
```

The results of this query are given in Table 2. Constructing a query which joins both of these queries exposes a difference between SPARQL and SQL. Abbreviating the two previous queries as `P1` and `P2`, consider a joining query:

```
SELECT ?patient ?diagnosis ?physicianName
WHERE {
    {P1} .
    {P2}
}
```

The resulting set of solution mappings is shown in Table 3 on the left. SPARQL engines commonly report results of SPARQL queries as relational tables with null values where variables are undefined. If we treat the tables for `P1` and `P2` as relations with null values, their join in SQL would produce a quite different result as shown in Table 3.

This example highlights why incremental view maintenance in SPARQL cannot be simply reduced to the relational setting.

?patient	**?diagnosis**	**?physicianName**
:alice	:diabetes	"Dr Carol"
:bob	:hypertension	*undefined*

Table 1. Results of a query with `OPTIONAL`, showing heterogeneous solution mappings. The first mapping binds all variables, whereas the second leaves ?physicianName undefined. SPARQL engines typically render such cases with NULL values.

?physicianName	**?patient**	**?diagnosis**
"Dr. Carol"	:alice	:diabetes
"Dr. Hyde"	*undefined*	*undefined*

Table 2. Results of the query mapping a physician to all their patients. In the first mapping, all variables are mapped. But in the second mapping both ?patient and ?diagnosis are left as undefined.

3.2 K-sets of solution mappings

We next introduce *K-sets* of solution mappings. These are finite sets of solution mappings as above, where additionally each mapping has an annotation. In our treatment, annotations are always integers. So, formally,

?patient	?diagnosis	?physicianName
:alice	:diabetes	"Dr. Carol"
:bob	:hypertension	"Dr. Hyde"

patient	diagnosis	physicianName
:alice	:diabetes	"Dr. Carol"

Table 3. Different results of joining the previous tables. The left result shows the join according to SPARQL, whilst the right table shows the join according to SQL.

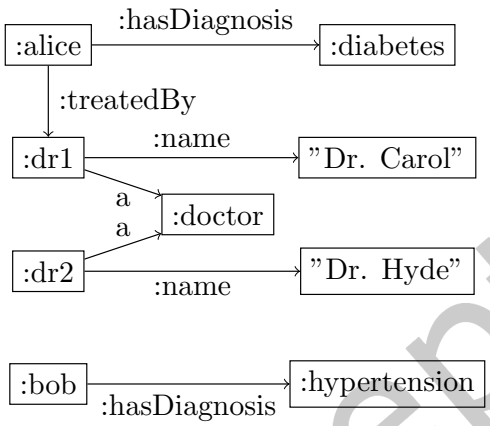


Fig. 1. RDF graph of two patients, one has a known doctor, the other has no known doctor. Two doctors are given with their respective names.

a K-set is a function $R : \Omega \to \mathbb{Z}$ where Ω is a finite set of solution mappings. An important remark for what follows is that when $R(\mu) = 0$ for a solution mapping $\mu \in \Omega$, we will treat μ as if it were not present in Ω.

We extend the SPARQL algebra to K-sets in the following manner. The above remark about zero annotations is reflected in the definitions of Diff and Minus. Let $R_1 : \Omega_1 \to \mathbb{Z}$, $R_2 : \Omega_2 \to \mathbb{Z}$, and $R : \Omega \to \mathbb{Z}$ denote K-sets.

Filter $\sigma_e(R) : \sigma_e(\Omega_1) \to \mathbb{Z} : \mu \mapsto R(\mu)$. The annotation of mappings satisfying the filter condition is simply preserved.

Join

$$R_1 \bowtie R_2 : \Omega_1 \bowtie \Omega_2 \to \mathbb{Z} : \mu \mapsto \sum_{\substack{(\mu_1,\mu_2)\in\Omega_1\times\Omega_2 \\ \mu_1\sim\mu_2 \\ \mu_1\cup\mu_2=\mu}} R_1(\mu_1) \cdot R_2(\mu_2)$$

The products of the annotations of all pairs of mappings that join to the same mapping are summed.

Union

$$R_1 + R_2 : \Omega_1 \cup \Omega_2 \to \mathbb{Z} : \mu \mapsto \begin{cases} R_1(\mu) + R_2(\mu) & \text{if } \mu \in \Omega_1 \cap \Omega_2; \\ R_1(\mu) & \text{if } \mu \in \Omega_1 - \Omega_2; \\ R_2(\mu) & \text{if } \mu \in \Omega_2 - \Omega_1. \end{cases}$$

When a mapping is in both sets, its two annotations are added; otherwise it simply keeps its annotation from where it comes from.

Diff $R_1 \ominus R_2 : \Omega' \to \mathbb{Z} : \mu \mapsto R_1(\mu)$, where

$$\Omega' = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 : R_2(\mu_2) \neq 0 \Rightarrow \mu_1 \nsim \mu_2\}.$$

Note that the annotation of an element μ in Ω' in $R_1 \ominus R_2$ is simply preserved from R_1.

LeftJoin $R_1 \mathbin{⟕} R_2$ is defined as $(R_1 \bowtie R_2) + (R_1 \ominus R_2)$ using the operations already introduced.

Minus $R_1 \boxminus R_2 : \Omega'' \to \mathbb{Z} : \mu \mapsto R_1(\mu)$, where

$$\Omega'' = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 : (R_2(\mu_2) \neq 0 \wedge \text{vars}(\mu_1) \cap \text{vars}(\mu_2) \neq \emptyset) \Rightarrow \mu_1 \nsim \mu_2\}.$$

As with Diff, the annotation from R_1 is preserved.

Project

$$\pi_X(R) : \pi_X(\Omega) \to \mathbb{Z} : \nu \mapsto \sum_{\substack{\mu \in \Omega \\ \mu|_{\text{vars}(\mu) \cap X} = \nu}} R(\nu).$$

The annotations of all mappings that yield the same projection are summed.

In the context of the examples used earlier, suppose a patient is associated with a single diagnosis but treated with two medications. Querying for each patient-diagnosis-medication triple then produces two distinct solution mappings. However, when the medication variable is projected away, both mappings collapse to the same patient-diagnosis pair. In this case we obtain one solution mapping with multiplicity $k = 2$.

3.3 Algebraic equivalences

The following theorem is a new result (see the Introduction for context) and provides the basis for the correctness of our incremental maintenance algorithm presented in Section 4.

Theorem 3.1. *Let R_1, R_2 and R_3 be K-sets. The following equalities hold:*

(1) $\sigma_e(R_1 + R_2) = \sigma_e(R_1) + \sigma_e(R_2)$.
(2) $(R_1 + R_2) \bowtie R_3 = (R_1 \bowtie R_3) + (R_2 \bowtie R_3)$.
(3) $\pi_X(R_1 + R_2) = \pi_X(R_1) + \pi_X(R_2)$.

The proof can be found in Appendix B.1.

Equally important are the properties of commutativity and associativity of Join and Union on K-sets. We leave them out of the above theorem as these properties are immediate from the definitions.

Absent in the above theorem are Diff, LeftJoin, and Minus. Incremental maintenance of patterns involving these nonmonotonic operators will be dealt with by a reduction to the relational setting, then using a technique from Gupta and Mumicks's counting algorithm.

3.4 SPARQL patterns over annotated RDF graphs

K-sets of solution mappings are obtained from RDF graphs initially by applying basic graph patterns (BGPs). These BGPs are the leaves in the parse tree of a SPARQL pattern; the tree is built up further by SPARQL algebra operators.

Formally, a BGP is a finite set of *triple patterns*, where a triple pattern is simply a triple composed of nodes and variables. Thus ensues the following formal definition of a *SPARQL pattern*:

(1) Every BGP is a SPARQL pattern.
(2) If P is a SPARQL pattern, e is a predicate on solution mappings, and X is a finite set of variables, then $\sigma_e(P)$ and $\pi_X(P)$ are SPARQL patterns too.
(3) If P_1 and P_2 are SPARQL patterns, then so are $P_1 \bowtie P_2$; $P_1 \ominus P_2$; $P_1 \mathbin{⟕} P_2$; $P_1 \cup P_2$; and $P_1 \boxminus P_2$.

It remains to formally state how the result of evaluating a pattern on an RDF graph is defined. Recall that an *RDF graph* (or simply graph) is a finite set of triples of nodes. To do incremental maintenance, we have to work with annotated RDF graphs, which we call *K-graphs*. Similarly to K-sets, a K-graph is a function $H : G \to \mathbb{Z}$,

where G is a graph. So, in H, each triple from G has an integer annotation. It will be convenient to view every graph G as a K-graph where every triple gets annotation value 1.

Let $T = (u, v, w)$ be a triple pattern, and let $X = \{u, v, w\} \cap V$. Note that the elements of X are the variables in T; the components of T not in X are node constants. For any solution mapping $\mu : X \to U$, let us agree for any node constant c that $\mu(c) = c$. Then for any K-graph $H : G \to \mathbb{Z}$ we can define $T(H)$ as the K-set $R : \Omega \to \mathbb{Z}$, where $\Omega = \{\mu : X \to U \mid (\mu(u), \mu(v), \mu(w)) \in G\}$, and $R(\mu) = H(\mu(u), \mu(v), \mu(w))$ for each $\mu \in \Omega$. So, every solution mapping inherits the annotation value of the triple in G that it matches.

Now for a BGP $P = \{T_1, \ldots, T_n\}$, we define the result of P on H to be the K-set $P(H) := T_1(H) \bowtie \cdots \bowtie T_n(H)$. All mappings in this $P(H)$ have the same schema, namely, the set of all variables that occur in P.

Any SPARQL pattern P can now be evaluated on a K-graph H in a bottom-up fashion, first evaluating the BGPs in the leaves, then applying to the resulting K-sets the SPARQL operators in the parents of the leaves, and so on. The final resulting K-set at the root is denoted by $P(H)$. Due to the interplay between Union, Project, Join, and LeftJoin operators, annotations can become greater and smaller. When all triples in H have a positive annotation, the same will hold for all solution mappings in $P(H)$. When all triples in H have an annotation value 1, the annotations in $P(H)$ correspond exactly to the multiplicities in the *multiset semantics* of SPARQL patterns.

Triple patterns in real BGPs can contain *blank nodes* [21]. These can be modeled in our formalism by replacing them by fresh variables and projecting on the original set of variables.

4 Incremental view maintenance for SPARQL patterns

We begin by defining the problem formally. Let P be a SPARQL pattern. An algorithm is said to do *incremental maintenance* for P if it has the following behavior:

State: The algorithm maintains an RDF graph G; the result $P(G)$ of evaluating P on G; and possibly additional auxiliary information.

Input: An update ΔG to G.

Output: A new state, consisting of the updated graph $G' = G + \Delta G$; the updated result $P(G')$; and updated auxiliary information.

A trivial such algorithm is the "from scratch" algorithm: it uses no auxiliary information and simply recomputes $P(G')$ each time. Our goal is to obtain a more efficient algorithm by maintaining auxiliary information allowing to update $P(G)$ directly. So the goal is to compute an update $\Delta P(G)$ such that $P(G + \Delta G) = P(G) + \Delta P(G)$.[4]

We can conveniently model updates to graphs using K-graphs. Specifically, an update ΔG to G is modeled as a K-graph $\Delta G : D \to \mathbb{Z}$ with the following special properties:

- ΔG gives each triple in D an annotation value 1 or -1, where 1 means insert and -1 means delete.
- The triples annotated with 1 should not yet be in G; those with -1 should be in G.

The updated graph $G + \Delta G$ is then defined as expected and equals

$$(G - \{t \in D \mid \Delta G(t) = -1\}) \cup \{t \in D \mid \Delta G(t) = 1\}.$$

4.1 The counting algorithm

Our approach in this article is based on elements from the *counting algorithm* [20] for incremental maintenance of nonrecursive Datalog programs in the relational model.

The counting algorithm provides a method to efficiently update materialized views when base relations change. A key idea is to maintain for each tuple a count of the distinct derivations in the view. This count is updated as changes in the base relations occur. The algorithm handles insertions and deletions. In case an insertion in

[4]Since ΔP depends not only on G but also on ΔG and possibly also on auxiliary state information S, it would be more correct to write $\Delta P(G, \Delta G, S)$, but we stick to $\Delta P(G)$ to keep notation light.

the base relations introduces a new derivation of a tuple in the view, the count is increased accordingly. If the insertion leads to a new tuple in the view, it is added to the table with an appropriate count. For deletions in the base relations, the affected tuples in the view are identified and their counts decremented. If a count drops to zero, the tuple is removed from the view.

By maintaining these counts correctly, the counting algorithm can identify the necessary view updates when base relations change, avoiding a complete recalculation of the view from scratch. This is typically an efficient approach in relational databases and minimizes computational costs.

To introduce the counting algorithm, we recall the following example from the original paper [20]:

$$Hop(x, y) \leftarrow Link(x, z), Link(z, y).$$

This is a view defined by a conjunctive query over a binary relation *Link*, defining a result relation *Hop*. If we interpret the *Link* relation as edges in a directed graph, the relation *Hop* will contain pairs (x, y) so that there is a path of length two from x to y. Typical applications of directed graphs include social networks (the "follows" relation), trophic networks in ecology ("who eats who"), or workflows (dependency relation).

To make this query incremental, we interpret all relations as K-relations. This means that every tuple is annotated with an integer. In the input relation *Link*, this integer is always simply 1. Updates to relation *Link* are collected in a relation $\Delta Link$ where new tuples to be inserted are again annotated with 1, and existing tuples to be deleted are annotated with -1. The updated relation $Link^{\nu}$ can be obtained as a sum $Link + \Delta Link$ of annotated relation. Here, the sum adds the annotations of tuples in the intersection; tuples appearing only in one of the two relations to be summed, simply keep their annotations.

$$Link'(x, y) \leftarrow Link(x, y).$$
$$Link'(x, y) \leftarrow \Delta Link(x, y).$$

The updates to relation *Hop* are then computed by a union of two conjunctive queries, referred to as delta rules:

$$\Delta Hop(x, y) \leftarrow \Delta Link(x, z), Link(z, y).$$
$$\Delta Hop(x, y) \leftarrow Link'(x, z), \Delta Link(z, y).$$

$$Hop'(x, y) \leftarrow Hop(x, y).$$
$$Hop'(x, y) \leftarrow \Delta Hop(x, y).$$

The counting algorithm prescribes a definite procedure to derive these delta rule from the conjunctive query; in general, if a conjunctive query has n subgoals then n delta rules are generated.

Crucial is that delta rules are interpreted over annotated relations. This means several things:

(1) When two tuples join, their annotations are multiplied.
(2) In the head of a rule, typically a projection happens on the joined tuples. (In the above example, variable z is projected away.) The annotation of a tuple in the result of the rule is defined to be the sum of all joined tuples that project to that result tuple.
(3) Finally, the results of the several rules are again summed, yielding the final annotated relation ΔHop.

The interesting property of this algorithm is that the sum $Hop' \leftarrow Hop + \Delta Hop$ produces exactly the tuples that would be returned if we computed the *Hop* query from scratch on the updated relation $Link'$. However, relation Hop' is now an annotated relation, where some tuples can have an annotation that is strictly higher than 1.

Example 4.1. To illustrate an example of this algorithm, consider the relation *Link* and $\Delta Link$. According to the algorithm, the next step produces an intermediary result of the different delta rules, corresponding to the $\Delta Hop(x, y) \leftarrow \Delta Link(x, z), Link(z, y)$ and $\Delta Hop(x, y) \leftarrow Link'(x, z), \Delta Link$. This results in the final relation of

Link

A	*B*	*k*
a	*b*	1
b	*c*	1

Hop

x	y	k
a	*c*	1

Δ*Link*

A	*B*	*k*
a	*b*	-1
c	*b*	1

Δ*Hop* ← Δ*Link*, *Link*

A	*B*	*k*
a	*c*	-1
c	*c*	1

Hop′

A	*B*	*k*
b	*b*	1
c	*c*	1

Link′

A	*B*	*k*
b	*c*	1
c	*b*	1

Δ*Hop* ← *Link*′, Δ*Link*

A	*B*	*k*
b	*b*	1

Table 4. Illustration of the counting algorithm on the *Hop* view.

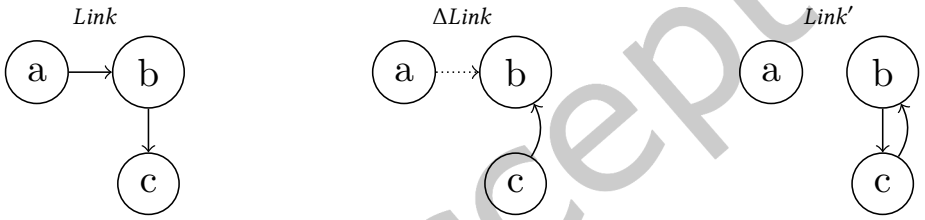


Fig. 2. *Link*, Δ*Link* and *Link*′ from Table 4 as graphs, In the Δ*Link* graph, the dotted line represents a deletion whilst the full edge represents an addition.

Hop′ which combines *Hop* and Δ*Hop* together. All relations and their values of this example are represented in Table 4 and shown in Figure 2.

Stating this example in the patient-doctor setting used earlier, assume that *a* and *c* are doctors, and *b* a patient. While we used unlabeled graphs for clarity, assume the relations :treats and :favoritedr for the triples in Figure 2 as follows. For *Link* we have the triples (:a, :treats, :b) and (b:, :favoritedr, :c). In Δ*Link* the first triple is deleted–doctor :a no longer treats patient :b–and a new one added, (:c, :treats, :b). Finally *Link*′ has two triples, (b:, :favoritedr, :c) and (:c, :treats, :b).

It is important to note that views can be composed. For example, consider the view:

$$TripleHop(x, y) \leftarrow Hop(x, z), Link(z, y)$$

To maintain *TripleHop* under updates Δ*Link*, we first compute Δ*Hop* and *Hop*′ as above. Using these, the delta rules for *TripleHop* can now be applied:

$$\Delta TripleHop(x, y) \leftarrow \Delta Hop(x, z), Link(z, y).$$
$$\Delta TripleHop(x, y) \leftarrow Hop'(x, z), \Delta Link(z, y).$$

Finally, we note that the counting algorithm also supports safe negation in rule bodies. We will discuss the delta rules for negation in Section 4.5.

4.2 Adapting the counting algorithm to SPARQL

We next illustrate by example how the counting algorithm can be useful for incremental maintenance of SPARQL patterns. We begin by observing that a BGP corresponds directly to a conjunctive query over a ternary relation G that holds the triples of the RDF graph.

4.2.1 BGP. For example, reconsider the *Hop* example as a SPARQL BGP. In real SPARQL syntax this is `{?x :link ?z. ?z :link ?y .}`. In our formalization this is $B = \{(x, \texttt{:link}, z), (z, \texttt{:link}, y)\}$. This corresponds to the following view:

$$B(x, z, y) \leftarrow G(x, \texttt{:link}, z), G(z, \texttt{:link}, y).$$

Applying the counting algorithm, B can be incrementally maintained by the following delta rules:

$$\Delta B(x, z, y) \leftarrow \Delta G(x, \texttt{:link}, z), G(z, \texttt{:link}, y).$$
$$\Delta B(x, z, y) \leftarrow G'(x, \texttt{:link}, z), \Delta G(z, \texttt{:link}, y).$$

In the SPARQL algebra on annotated sets of mappings, this becomes a sum of two joins:

$$\Delta B := (x, \texttt{:link}, z)(\Delta G) \bowtie (z, \texttt{:link}, y)(G) + (x, \texttt{:link}, z)(G + \Delta G) \bowtie (z, \texttt{:link}, y)(\Delta G).$$

4.2.2 Projection. As another example, let us look at the *Hop* example's projection in SPARQL terms. In SPARQL syntax this is `{ SELECT ?x ?y WHERE { B } }`, with B the above BGP. This corresponds to the following view:

$$P(x, y) \leftarrow B(x, z, y).$$

Such a projection is maintained by one simple delta rule:

$$\Delta P(x, y) \leftarrow \Delta B(x, z, y).$$

In the SPARQL algebra on annotated sets of mappings, this becomes a projection:

$$\Delta P := \pi_{x,y}(\Delta B)$$

4.2.3 Union. To illustrate the handling of union, consider the following pattern in real SPARQL syntax: Consider the union of two BGP's `{?x :link "Jan"} UNION {?y :link "Dore"}`. In our formalization this becomes $U = \{(x, \texttt{:link}, \texttt{"Jan"})\} + \{(y, \texttt{:link}, \texttt{"Dore"})\}$. Solution mappings in the result of this pattern can have different schemas, namely, $\{x\}$ and $\{y\}$. We cannot simply treat this as a view defined as a union of two conjunctive queries, as in:

$$U(x) \leftarrow G(x, \texttt{:link}, \texttt{"Jan"}).$$
$$U(y) \leftarrow G(y, \texttt{:link}, \texttt{"Dore"}).$$

This would lose the schemas of the solutions mappings. Indeed, since the choice of variables in Datalog-like rules is free, the above is equivalent to the following:

$$U(z) \leftarrow G(z, \texttt{:link}, \texttt{"Jan"}).$$
$$U(z) \leftarrow G(z, \texttt{:link}, \texttt{"Dore"}).$$

Instead, we need to introduce two views U_x and U_y as follows:

$$U_x(x) \leftarrow G(x, \texttt{:link}, \texttt{"Jan"}).$$
$$U_y(y) \leftarrow G(y, \texttt{:link}, \texttt{"Dore"}).$$

This illustrates a first complication in adapating the counting algorithm, deveveloped for the relational models, to the SPARQL context. In the relational model every relation has only one schema, but results of SPARQL patterns are heterogeneous.

Since U_x and U_y are separate views, they each have their own delta rule, in this case:

$$\Delta U_x(x) \leftarrow \Delta G(x, \texttt{:link}, \texttt{"Jan"}).$$
$$\Delta U_y(y) \leftarrow \Delta G(y, \texttt{:link}, \texttt{"Dore"}).$$

Finally, in the SPARQL algebra, we obtain:

$$\Delta U := U(G) + (x, \texttt{:link}, \text{"}Jan)(\Delta G) + (y, \texttt{:link}, \texttt{"Dore"})(\Delta G).$$

4.2.4 Join. Let us next join the previous union with a third triple pattern, in real SPARQL syntax: `{ U } . { ?z :link "Bart" }`. In our formalization this becomes $J = U \bowtie (z, \texttt{:link}, \texttt{"Bart"})$. Since U is heterogeneous, there are two corresponding views:

$$J_{xz}(x, z) \leftarrow U_x(x), G(z, \texttt{:link}, \texttt{"Bart"}).$$
$$J_{yz}(y, z) \leftarrow U_y(y), G(z, \texttt{:link}, \texttt{"Bart"}).$$

We apply the counting algorithm to both views as before:

$$\Delta J_{xz}(x, z) \leftarrow \Delta U_x(x), G(z, \texttt{:link}, \texttt{"Bart"}).$$
$$\Delta J_{xz}(x, z) \leftarrow U'_x(x), \Delta G(z, \texttt{:link}, \texttt{"Bart"}).$$
$$\Delta J_{yz}(y, z) \leftarrow \Delta U_y(y), G(z, \texttt{:link}, \texttt{"Bart"}).$$
$$\Delta J_{yz}(y, z) \leftarrow U'_y(y), \Delta G(z, \texttt{:link}, \texttt{"Bart"}).$$

In contrast, since the SPARQL algebra can deal with schematic heterogeneity, the delta expression for J is simply

$$\Delta J := \Delta U \bowtie (z, \texttt{:link}, \texttt{"Bart"})(G) + U' \bowtie (z, \texttt{:link}, \texttt{"Bart"})(\Delta G),$$

where $U' = U(G) + \Delta U$.

This shows that it is advantageous to redevelop the counting algorithm directly on the level of the SPARQL algebra, as this leads to simpler delta expressions. We will be takin on precisely that task in Sections 4.3, 4.4 and 4.5.

4.2.5 Minus. This corresponds to a view with negation:

$$M(x, y) \leftarrow G(x, \texttt{:link}, y), \neg G(y, \texttt{:link}, \texttt{"Dore"})$$

While the counting algorithm can handle negation, the treatment necessitates the introduction of a special operator. We defer the discussion to Section 4.5.

4.2.6 LeftJoin. Let us consider a leftjoin operation, which in SPARQL syntax is written as OPTIONAL, for example, `{?x :link ?y . OPTIONAL{?y :email ?z.}}` In our formalization this becomes this $L = \{(x, \texttt{:link}, y)\} ⟕ \{(y, \texttt{:email}, z)\}$.

The leftjoin operation corresponds to a union of two views as follows:

$$HasE(y) \leftarrow G(y, \texttt{:email}, z)$$
$$L(x, y) \leftarrow G(x, Link, y), G(y, \texttt{:email}, z)$$
$$L(x, y) \leftarrow G(x, Link, y), \neg HasE(y)$$

Hence, incremental maintenance of Leftjoin is a combination of incremental maintenance of unions and negations as already illustrated above.

4.3 Incremental maintenance of BGPs

Framing BGPs in the context of the counting algorithm for incremental maintenance of Datalog programs, these BGPs can be interpreted as negation-free Datalog rules.

Specifically, let $B = \{T_1, \ldots, T_n\}$ be a BGP, where $T_i = (u_i, v_i, w_i)$ for $i = 1, \ldots, n$. Let G be a relation name for the underlying RDF graph. Let $S = \{x_1, \ldots, x_k\}$ be the set of all variables in B. Then B is interpreted as the following Datalog rule:

$$B(x_1, \ldots, x_k) \leftarrow G(u_1, v_1, w_1), \ldots, G(u_n, v_n, w_n). \tag{$*$}$$

Example 4.2. Consider B, the BGP `{ ?x foaf:name ?y . ?x foaf:mbox ?z }`. In our formalization B is $\{(x, \texttt{foaf:name}, y), (x, \texttt{foaf:mbox}, z)\}$. Then B is interpreted as the Datalog rule

$$B(x, y, z) \leftarrow G(x, \texttt{foaf:name}, y),\ G(x, \texttt{foaf:mbox}, z). \qquad \square$$

To maintain rules like $(*)$ under updates, the counting algorithm introduces an ingenious set of so-called *delta rules*. Leveraging the correspondence between BGPs and rules, we can adapt these delta rules to the SPARQL setting. This results in our theorem as follows.

Theorem 4.3. *Let $B = \{T_1, T_2, T_3, \ldots, T_n\}$ be a BGP, let G be a graph, and let ΔG be an update to G. Let $G' = G + \Delta G$. Then $B(G') = B(G) + \Delta B(G)$, where*

$$\begin{aligned}
\Delta B(G) = {} & T_1(\Delta G) \bowtie T_2(G) \bowtie T_3(G) \bowtie \cdots \bowtie T_n(G) \\
& + T_1(G') \bowtie T_2(\Delta G) \bowtie T_3(G) \bowtie \cdots \bowtie T_n(G) \\
& + T_1(G') \bowtie T_2(G') \bowtie T_3(\Delta G) \bowtie \cdots \bowtie T_n(G) \\
& \vdots \\
& + T_1(G') \bowtie T_2(G') \bowtie T_3(G') \bowtie \cdots \bowtie T_n(\Delta G).
\end{aligned}$$

The proof for Theorem 4.3 can be found in Appendix B.2.

The above theorem provides an incremental algorithm for B. Every term in the incremental expression $\Delta B(G)$ has a factor involving ΔG. If ΔG is very small compared to G, each term will be small, and evaluating the incremental expression will be faster than evaluating $B(G')$ from scratch. Note that no auxiliary information is needed yet; this will be needed for more complex SPARQL patterns built from BGPs.

It is important to note that, just as in ΔG, solution mappings may receive a *negative* annotation from $\Delta B(G)$.

Example 4.4. For B in Example 4.2, we obtain the following delta rules (staying here with the Datalog notation for the sake of illustration):

$$\begin{aligned}
\Delta B(x, y, z) &\leftarrow \Delta G(x, \texttt{foaf:name}, y),\ G(x, \texttt{foaf:mbox}, z). \\
\Delta B(x, y, z) &\leftarrow G'(x, \texttt{foaf:name}, y),\ \Delta G(x, \texttt{foaf:mbox}, z).
\end{aligned}$$

4.4 Filter, Join, Union, Projection

More complex patterns are built from BGPs using the SPARQL algebra operators. When these operators are Filter, Join, Union, or Projection, incremental maintenance poses no additional problems:

Filter: Let P be of the form $\sigma_e(P_1)$. Applying our method recursively to P_1, we obtain a K-set $\Delta P_1(G)$. Now

$$\begin{aligned}
P(G + \Delta G) &= \sigma_e(P_1(G + \Delta G)) \\
&= \sigma_e(P_1(G) + \Delta P_1(G)) \\
&= \sigma_e(P_1(G)) + \sigma_e(\Delta P_1(G)) \qquad \text{by Theorem 3.1} \\
&= P(G) + \sigma_e(\Delta P_1(G)).
\end{aligned}$$

Hence, $\sigma_e(\Delta P_1(G))$ serves as $\Delta P(G)$. Note that $\Delta P_1(G)$ is now auxiliary information for the incremental maintenance algorithm.

Project: Completely analogous to Filter.

Union: Let P be of the form $P_1 + P_2$. Clearly,

$$\begin{aligned} P(G + \Delta G) &= (P_1 + P_2)(G') \\ &= P_1(G') + P_2(G') \\ &= P_1(G) + \Delta P_1(G) + P_2(G) + \Delta P_2(G) \\ &= P(G) + \Delta P_1(G) + \Delta P_2(G). \end{aligned}$$

Hence, $\Delta P_1(G) + \Delta P_2(G)$ serves as $\Delta P(G)$.

Join: Let P be of the form $P_1 \bowtie \cdots \bowtie P_n$. Assuming we have $\Delta P_i(G)$ for $i = 1, \ldots, n$, we can again apply a generalization of Theorem 4.3 (substituting P_i for T_i) to obtain an expression for $\Delta P(G)$.

Indeed, we can formulate and prove Theorem 4.3 more generally as follows:

Theorem 4.5. *Let P be a SPARQL pattern of the form $P_1 \bowtie \cdots \bowtie P_n$, let G be a graph, and let ΔG be an update to G. Let $G' = G + \Delta G$. Assume we already have K-sets $\Delta P_i(G)$ so that $P_i(G') = P_i(G) + \Delta P_i(G)$, for $i = 1, \ldots, n$. Then $P(G') = P(G) + \Delta P(G)$, where*

$$\begin{aligned} \Delta P(G) = {} & \Delta P_1(G) \bowtie P_2(G) \bowtie P_3(G) \bowtie \cdots \bowtie P_n(G) \\ & + P_1(G') \bowtie \Delta P_2(G) \bowtie P_3(G) \bowtie \cdots \bowtie P_n(G) \\ & + P_1(G') \bowtie P_2(G') \bowtie \Delta P_3(G) \bowtie \cdots \bowtie P_n(G) \\ & \vdots \\ & + P_1(G') \bowtie P_2(G') \bowtie P_3(G') \bowtie \cdots \bowtie \Delta P_n(G). \end{aligned}$$

The proof of Theorem 4.5 can be found in Appendix B.3.

4.5 Diff, Minus, LeftJoin

The counting algorithm supports rules with negation. We first explain their method, which works only for K-relations, not K-sets with schematic heterogeneity. We then explain how we can obtain incremental maintenance for Diff and Minus by reducing K-sets to multiple K-relations.

4.5.1 Negation in the counting algorithm. Consider a rule with negation:

$$P(\bar{x}) \leftarrow R(\bar{x}),\ \neg S(\bar{y}). \tag{$\dagger$}$$

where $\bar{y} \subseteq \bar{x}$. This rule can be maintained by the following two delta rules:

$$\Delta P(\bar{x}) \leftarrow \Delta R(\bar{x}),\ \neg S(\bar{y}). \tag{$\ddagger$}$$

$$\Delta P(\bar{x}) \leftarrow R'(\bar{x}),\ \Delta\overline{S}(\bar{y}). \tag{$\ddagger$}$$

Here, R' denotes R applied to G', and $\Delta\overline{S}$ is the K-relation containing the following tuples:

- Tuples $t \in \Delta S$ such that $\Delta S(t) = -S(t)$. These are tuples that are entirely deleted from S. We define their annotation in $\Delta\overline{S}$ to be 1.
- Tuples $t \in \Delta S$ that are not in S. These are tuples that are inserted into S. We define their annotation in $\Delta\overline{S}$ to be -1.

Again, since each delta rule involves one delta relation, they will be efficient if deltas are very small compared to the data.

4.5.2 Delta expressions for negation in SPARQL. Let us now translate the treatment of negation from the relational setting of the counting algorithm to the proper formal setting of the SPARQL algebra. We will provide precise delta expressions and formally prove their correctness. Remarkably, no correctness proof for the treatment of negation in the counting algorithm has appeared before. Thus, our proof also serves to fill this gap in the literature.

Let us call a K-set $R : \Omega \to \mathbb{Z}$ a *K-relation* if all solution mappings in Ω have the same schema. In this case, we will also refer to the solution mappings in Ω as *tuples* and denote them by the letter t instead of μ.

When comparing two K-relations, it makes sense to ignore tuples with zero annotations. For any K-relation $R : \Omega \to \mathbb{Z}$, the set $\{t \in \Omega \mid R(t) \neq 0\}$ is called the *support* of R and denoted by $\mathrm{supp}(R)$. We will abuse notation and write $t \in R$ to mean that $t \in \mathrm{supp}(R)$.

We now say that two K-relations $R : \Omega \to \mathbb{Z}$ and $R' : \Omega' \to \mathbb{Z}$ are *equal up to zeroes*, denoted by $R \equiv R'$, if they have the same support, and agree on their support, i.e., for each $t \in \mathrm{supp}(R) = \mathrm{supp}(R')$ we have $R(t) = R'(t)$.

We are now ready to justify the treatment of negation in the counting algorith by the following theorem.

Theorem 4.6. *Let $Y \subseteq X$ be finite sets of variables. Let E be a K-relation with schema X, and let F_1 and F_2 be K-relations with schema Y. Let $F_{12} : \mathrm{supp}(F_2) \to \mathbb{Z}$ be the K-relation defined as follows:*

$$F_{12}(t) = \begin{cases} 1 & \text{if } F_2(t) = -F_1(t); \\ -1 & \text{if } t \notin F_1; \\ 0 & \text{otherwise.} \end{cases}$$

Then $E \ominus (F_1 + F_2) \equiv (E \ominus F_1) + (E \bowtie F_{12})$.

The proof of Theorem 4.6 can be found in Appendix B.4.

From Theorem 4.6 we can readily obtain delta expressions for Diff that are correct for any K-relation R with schema X and any K-relation S with schema Y, with $Y \subseteq X$. Let ΔR and ΔS hold updates for R and S respectively. We let S and ΔS play the roles of F_1 and F_2 in the theorem; denote the corresponding F_{12} by $\Delta\overline{S}$. Then

$$\begin{aligned}(R + \Delta R) \ominus (S + \Delta S) &\equiv ((R + \Delta R) \ominus S) + ((R + \Delta R) \bowtie \Delta\overline{S}) \\ &= (R \ominus S) + (\Delta R \ominus S) + ((R + \Delta R) \bowtie \Delta\overline{S}).\end{aligned}$$

Note how the sum of the second and third term in the last expression serves as the delta. The correspondence to the above delta rules (‡) is quite apparent.

4.5.3 Incremental maintenance for Diff and Minus. Now consider a SPARQL Diff pattern P of the form $P_1 \ominus P_2$. By applying our method recursively, we already have, as auxiliary information, $P_1(G)$, $P_1(G')$, and $\Delta P_1(G)$, as well as $P_2(G)$, $P_2(G')$, and $\Delta P_2(G)$. In general these are K-sets, not K-relations. For $j = 1, 2$, let $\mathcal{S}^{(j)}$ denote the set of schemas that may occur from the evaluation of P_j. These sets of schemas can easily be derived from the syntax of the patterns. In practice, we expect them to have relatively few possible schemas. In general, we will split a K-set R with set $\mathcal{S}$ of possible schemas into K-relations $R[S]$ for every $S \in \mathcal{S}$.

We now consider each $S \in \mathcal{S}^{(1)}$ separately. Let $\mathcal{S}^{(2)} = \{S_1, S_2, \ldots, S_\ell\}$. We can see that

$$P[S] = ((\ldots((P_1[S] \ominus \pi_{S \cap S_1}(P_2[S_1])) \ominus \pi_{S \cap S_2}(P_2[S_2])) \ominus \cdots) \ominus \pi_{S \cap S_\ell}(P_2[S_\ell]) \ldots).$$

Each of the Diff operators in the above expression is a relational negation as in rule (†) above. Thus, $P[S]$ can be maintained using the techniques for maintaining rules with negation, and projections. Finally, $P = \sum_{S \in \mathcal{S}^{(1)}} P[S]$ is maintained as a sum of these.

Minus and LeftJoin. For Minus, we use just a slight variation where relations $P_2[S']$ with $S \cap S' = \emptyset$ are omitted from the sequence of Diff operations. LeftJoin, being a combination of Join, Union, and Diff, is now covered as well.

4.6 Conclusion

This concludes the description of the algorithm. As part of its state, it maintains $Q(G)$ and $\Delta Q(G)$ for all subpatterns Q of the pattern P we are maintaining. The definitions for the $\Delta Q(G)$ given above are constructive, so constitute an algorithm. Yet, our method can be implemented in many different ways. We propose one approach in the next section.

5 Experiments and results

5.1 Implementation details

We implement our proposed incremental view maintenance algorithm. A key element is the maintenance of all K-sets, which we split into relations for the different schemas, and store these in tables. All such auxiliary data table manipulations are conveniently achieved using a relational database system, for which we used the fast in-process DuckDB [33]. Hence, all ancillary maintenance operations, both SPARQL operators reduced to relations, and delta rules with negation (Section 4.5), are implemented using SQL queries on main-memory tables. The implementations are described as follows.

Filter: Easily achieved in SQL by using a WHERE-condition.

Join: Assume K-relations $R(A, B, K)$ and $S(B, C, K)$; the annotation is always kept in a column K. Then $R \bowtie S$ is implemented as follows:

```
select R.A, R.B, S.C, R.K * S.K
from R join S on R.B = S.B
```

Project: Assume $R(A, B, K)$ as above. Then $\pi_A(R)$ is implemented as follows:

```
select A, sum(K) from R group by A
```

Union: Assume K-relations $R(A, K)$ and $S(A, K)$. Then $R + S$ is implemented as follows:

```
select (case when R.A not null then R.A else S.A end) as A,
       (case when R.A is null then S.K
             when S.A is null then R.K
             else R.K + S.K end) as K
from R full outer join S on R.A = S.A
```

Interestingly, we found experimentally that this implementation is significantly faster (roughly twice as fast) as the alternative where S is inserted into a copy of R, after which we group by A and sum K. We have tested this both when S is much smaller than R, i.e., $R + S$ is like updating R with S, as well as when they have the same size (tested up to 10 million).

Diff: Assume K-relations $R(A, B, K)$ and $S(B, K)$, and recall the first type of delta rule from Section 4.5:

$$\Delta P(a, b) \leftarrow \Delta R(a, b),\ \neg S(b).$$

This is easily expressed in SQL as

```
select A, B, K from DeltaR where B not in (select B from S)
```

The second type of delta rule

$$\Delta P(a, b) \leftarrow R'(a, b),\ \Delta\overline{S}(b)$$

is expressed as follows, taking into account the definition given for $\Delta\overline{S}$:

```
select Rprime.*
from Rprime join DeltaS on Rprime.B = DeltaS.B
where (DeltaS.B, -DeltaS.K) in (select * from S)
union
```

```
select Rprime.A, Rprime.B, -Rprime.K
from Rprime join DeltaS on Rprime.B = DeltaS.B
and DeltaS.B not in (select B from S)
```

Our software codes and experiments discussed below are implemented as a Python wrapper based on RDFLib [35], and are made publicly available (https://anonymous.4open.science/r/counting_sparql-C856/README.md). We used a 4-core Intel Core i5-1145G7 2.6GHz processor with 16GB DDR4 RAM and 300GB SSD to run all experiments.

5.2 Experiments and results

To our knowledge, our implementation is the first to provide incremental view maintenance of full SPARQL patterns (with the exception of property paths and aggregation). Hence, at this stage, the purpose of our experiments is not to compare with competing implementations or approaches. Rather, our goal is to demonstrate the feasibility of our algorithm and assess its merits. In this section, we describe three experiments: a simple query to illustrate incremental view maintenance where it is anticipated to work well; a range of standard queries on benchmark data from the Berlin SPARQL Benchmark (BSBM) [7]; and a dedicated illustrative query on the same BSBM dataset.

5.2.1 Synthetic data. The first experiment is based on the presumption that the incremental maintenance of a view is not necessarily always faster than recomputing the view from scratch, as incremental maintenance requires saving intermediate results, which may lead to computational overhead. When a graph G is updated by ΔG, and ΔG consists of a large number of deletions, the result $G' = G + \Delta G$ may be so small that recomputing from scratch will be faster. In the typical situation, however, is ΔG small compared to G and G'. Since the maintenance expressions always involve joins with intermediate results based on ΔG, we expect incremental maintenance to be faster in these cases. We test our implementation for this hypothesis using a simple pattern that matches pairs of nodes joined by a path of length three:

```
SELECT ?x ?y
WHERE {
    ?x :link ?z1 .
    ?z1 :link ?z2 .
    ?z2 :link ?y
}
```

In the example scenario used earlier, of patient–diagnosis–medication, paths of length three arise when we add side-effects to the chain of triples: patient–diagnosis–medication–side-effect. The query above corresponds to asking which side-effects a patient may experience given their diagnosis and medication. Since we want to assess our algorihm computationally, we simply construct a synthetic RDF graph that is composed by joining three random bipartite graphs, each between N nodes, as illustrated in Figure 3 (left). We used $N = 1000$ in the experiment and use a small ΔG which inserts and deletes 50 random edges in the graph. The results are shown in the same Figure (right). When G is relatively small ($p < 1/100$, approximately 10 000 random edges in every bipartite graph), computing the view from-scratch is faster than updating it incrementally. At $p = 1/100$ there is a break-even point. Incremental maintenance becomes increasingly faster for larger graphs ($p = 1/50$, $p = 1/35$). At around $p = 1/30$, however, the number of paths of length three no longer fits in main memory. The overhead of writing intermediate results to disk becomes heavier and the incremental advantage decreases. While $p = 1/25$ is close to the maximum capacity of the computer setup we used, we cannot test, yet expect incremental maintenance to win again more for even larger sizes. Indeed, the incremental expressions have inherently a lower complexity than the BGP from scratch.

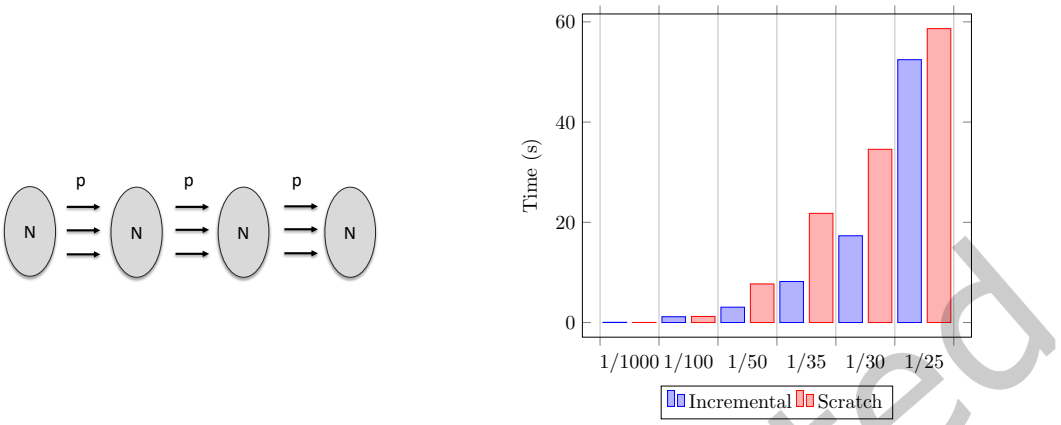


Fig. 3. Left: Synthetic graph for performance experiment. Each blob consists of N nodes. A fraction p of the N^2 possible edges from one blob to the next blob is chosen randomly. Right: results of the performance experiment.

The precise memory overheads we measured (expressed in number of tuples and rounded to thousands or millions) are as follows:

$1/p$	1000	100	50	35	30	25
BGP size	1K	1M	8M	23M	37M	64M
$\pi_{?x,?y}$ size	1K	600K	1M	1M	1M	1M

In addition, we ran a side experiment where 25 isolated edges were inserted in the $p = 1/35$ graph. These insertions are irrelevant to paths of length three, and indeed, incremental maintenance was found to run three times faster (21s) than recomputation from scratch (60s).

5.2.2 Berlin SPARQL Benchmark. Second, we conducted experiments using data and queries from the Berlin SPARQL Benchmark, BSBM [5, 7]. This benchmark simulates an e-commerce use case. BSBM includes a data generator generating RDF graphs of desired size, with nodes belonging to a number of related classes, such as Product, ProductType, and ProductFeature [6]. The BSBM "Explore" use case offers 12 SPARQL select queries. We worked with the first four queries, reproduced below, which offer a representative set of constructs. Fields between percent symbols are parameters. The remainining 8 queries contain no operators supported by our method, that are not covererd by these four queries already.

Query 1 asks for products of a specified type with two specified features and satisfying a condition on a numerical property. It is a BGP, followed by a Filter and a Project.

```
SELECT ?p ?label
WHERE {
    ?p rdfs:label ?label .
    ?p a %ProductType% .
    ?p b:Feature %Feat1% .
    ?p b:Feature %Feat2% .
    ?p b:PropNum1 ?value1 .
```

```
    FILTER (?value1 > %x%)
  }
```

The converted query and converted delta queries converted to SQL are found in appendix A.

Query 2 asks whether a specified product has some required properties, and optionally retrieves some optional properties. Its pattern involves BGPs combined with LeftJoin.

```
SELECT ?label ?comment ?p ?pFeature
?Text1 ?Text2 ?Text3
?Num1 ?Num2 ?Text4 ?Text5 ?Num4
WHERE {
    %XYZ% rdfs:label ?label .
    %XYZ% rdfs:comment ?comment .
    %XYZ% bsbm:producer ?p .
    ?p rdfs:label ?producer .
    %XYZ% dc:publisher ?p .
    %XYZ% b:Feature ?f .
    ?f rdfs:label ?pFeature .
    %XYZ% b:PropText1 ?Text1 .
    %XYZ% b:PropText2 ?Text2 .
    %XYZ% b:PropText3 ?Text3 .
    %XYZ% b:PropNum1 ?Num1 .
    %XYZ% b:PropNum2 ?Num2 .
    OPTIONAL {
        %XYZ% b:PropText4 ?Text4
      }
    OPTIONAL {
        %XYZ% b:PropText5 ?Text5
      }
    OPTIONAL {
        %XYZ% b:PropNum4 ?Num4
      }
  }
```

Query 3 is similar to Query 1, except that the products should *not* have a specified feature, so its pattern invokes the Minus operator.[5]

```
SELECT ?p ?label
WHERE {
  {
      ?p rdfs:label ?label .
      ?p a %ProductType% .
      ?p b:Feature %Feat1% .
      ?p b:PropNum1 ?p1 .
      FILTER ( ?p1 > %x% )
      ?p b:PropNum3 ?p3 .
      FILTER (?p3 < %y% )
```

[5]The BSBM formulation of Query 3 is in SPARQL 1.0 and simulates Minus using optional matching and negated bound variable checking. We formulate the query directly with Minus.

```
    }
  MINUS {
      ?p b:Feature %Feat2%
    }
}
```

Query 4 is again similar to Query 1, except that there is a disjunction between two specified features: products with one of the specified feature should satisfy a different filter condition than products with the other specified feature. Its pattern is a Union of BGPs with Filters.

```
SELECT ?p ?label ?Text
WHERE {
    ?p rdfs:label ?label .
    ?p rdf:type %ProductType% .
    ?p b:Feature %Feat1% .
    ?p b:Feature %Feat2% .
    ?p b:PropText1 ?Text .
    ?p b:PropNum1 ?p1 .
    FILTER ( ?p1 > %x% )
  } UNION {
    ?p rdfs:label ?label .
    ?p rdf:type %ProductType% .
    ?p b:Feature %Feat1% .
    ?p b:Feature %Feat3% .
    ?p b:PropText1 ?Text .
    ?p b:PropNum2 ?p2 .
    FILTER ( ?p2> %y% )
  }
```

To test the correctness of our implementation, we chose suitable parameter values for each query. Over the generated data, each instantiated query typically returns only 0–3 answers. This makes the experiment unsuitable for performance measurement, but suitable for correctness testing. We work over three RDF graphs of increasing size, generated for 1,000, 10,000, and 15,000 product entities respectively. For each query we construct update sets with four types of updates:

(1) Deletion of all triples involving products returned by the query;
(2) Insertion of all triples involving products returned by the query;
(3) Insertion of all triples involving a selected product that is irrelevant to the query;
(4) Similarly, deletion of all triples involving a selected product that is irrelevant to the query.

For each query Q, we evaluate $Q(G+\Delta G)$ both from scratch and incrementally, for each graph G and corresponding updates ΔG. The results were confirmed to be identical.

The memory overhead for all four different BSBM queries was always approximately 100% higher than the from-scratch approach; this is illustrated in Figure 4, showing the results for the smallest graph; the results for the larger ones are very similar.

The four queries were applied to the three graphs of varying sizes. The execution times are shown in Figure 5. For queries 1 and 3, the incremental updates are faster than from-scratch for the two larger graphs. For query 4, the incremental updates are roughly equivalent to from-scratch for the second largest graph and faster than from-scratch for the largest graph. The execution times for query 2 were 16,814ms, 20,762ms, and 21,166ms respectively

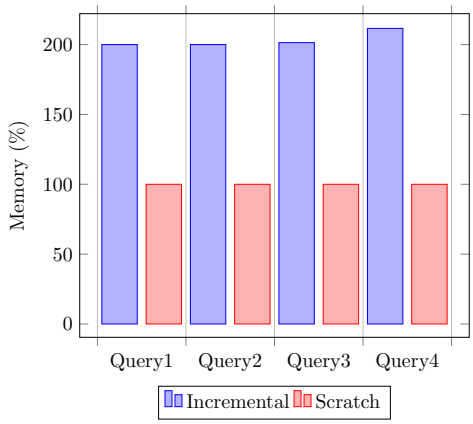


Fig. 4. Memory overhead for the four different tested BSBM queries; for each query, an increase of approximately 100% is seen in the incremental approach compared to from-scratch. For the smallest graph, shown here, the from-scratch approach consists of 13, 123, 289 and 69 triples respectively for Queries 1 to 4.

for the three graphs; for clarity, these bars in the graph are not shown completely. The poor performance of the incremental maintenance of Query 2 is likely due to the OPTIONALs it contains.

We investigated the performance of the queries in more detail, to determine where bottlenecks occur. The results for the smallest graph are shown in Figure 6. Each different operation in the queries was timed individually to assess the performance of each specific operation. The execution times of the incremental maintence approach are given relative to the from-scratch approach. Overall, a better performance of the incremental mainatence is observed for all operations except Minus, LeftJoin and BGPs. For example in Query 1, the BGP shows an increase in computation time, while the selection and projection operations both show a decrease in computation time. Note that the BGPs that do not have many triple patterns do perform better than their from-scratch counterparts. In Query 1, the BGP component refers to all triple pattens in the WHERE clause, whereas Filter refers to the time it took to compute the FILTER operation, and similarly Project for the SELECT clause. BGP_1, BGP_2 and BGP_3 refer to the first, second, and third triple patterns inside the OPTIONAL patterns in Query 2. LeftJoin_1, LeftJoin_2 and LeftJoin_3 refer to the same OPTIONAL patterns, respectively. In Query 3, the first BGP refers to the triple patterns outside of the MINUS pattern, while BGP_1 refers to the triple patterns inside the MINUS pattern. In Query 4, the first BGP and Filter refer to the triple patterns and FILTER before the UNION keyword, and the second BGP and Filter refer to the triple patterns and FILTER after the UNION. The perfomance of the BGP and LeftJoins affects Query 2 most, as was seen already in Figure 5.

5.2.3 Star-shaped versus path-shaped patterns. So far our experiments show that incremental beats from-scratch on path queries over synthetic data, but that incremental does not beat from-scratch on the BSBM queries. A possible hypothesis is that these different results are explained by the shape of the BGPs. Indeed, the BSBM queries have typically star-shaped BGPs where the join variable is in the same position in the different triple patterns that make up the BGP. Such a star shape is in contrast to the shape of a path query. Star-shaped queries are typically well supported (already from scratch) by SPARQL engines [11].

To test this hypothesis, we constructed a query over BSBM data that has more of a path shape. The following query returns all features of all products of producers that produce at least two different products:

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
```

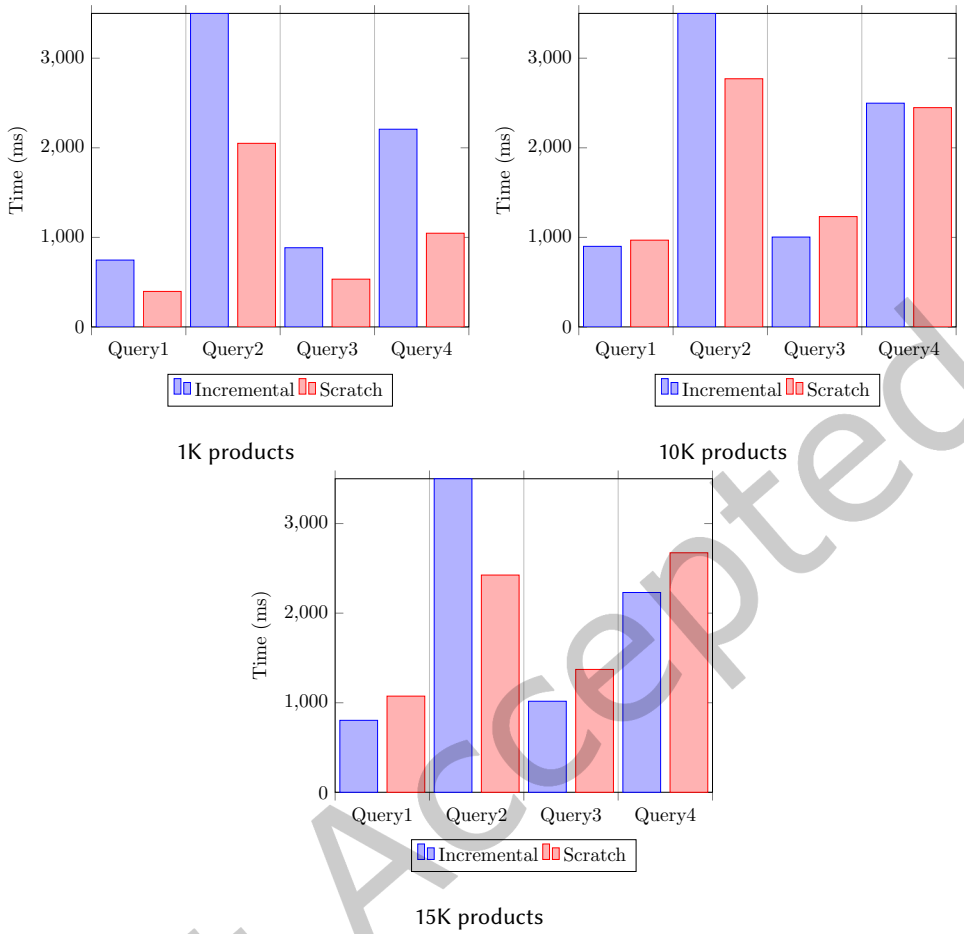
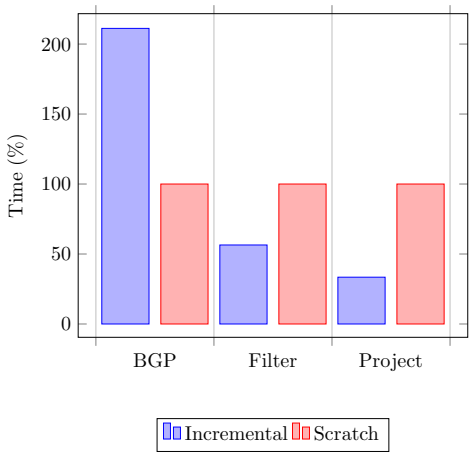


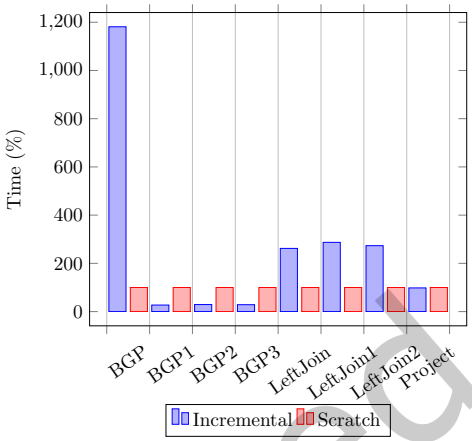
Fig. 5. BSBM queries, as applied to different sizes of datasets. Query 2, containing multiple leftjoin operators, has a value that goes beyond the scope of the figure, with the intent of increasing readability of the figure. The values for Query 2 are 16,814ms, 20,762ms, and 21,166ms respectively.


```
SELECT ?feat
WHERE {
    ?product1 bsbm:producer ?producer .
    ?product2 bsbm:producer ?producer .
    ?product2 bsbm:productFeature ?feat .
    FILTER (?product1 != ?product2)
}
```

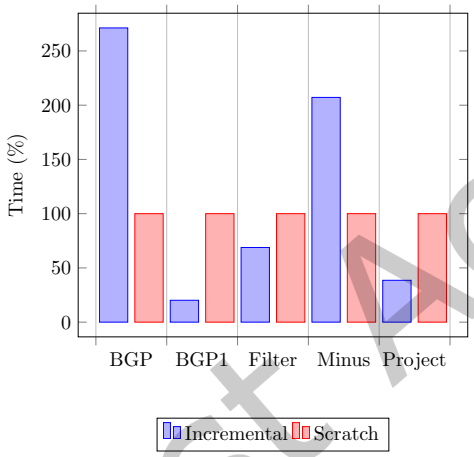
Figure 7 shows a snapshot of the BSBM graph to illustrate the path-like shape in the data (from one product to another through a common producer) that the above query matches. Figure 8 gives the execution times of the query above. Here, indeed, as opposed to the four BSBM queries in the previous experiment, incremental maintenance outperforms from-scratch for all operators, including the BGP.



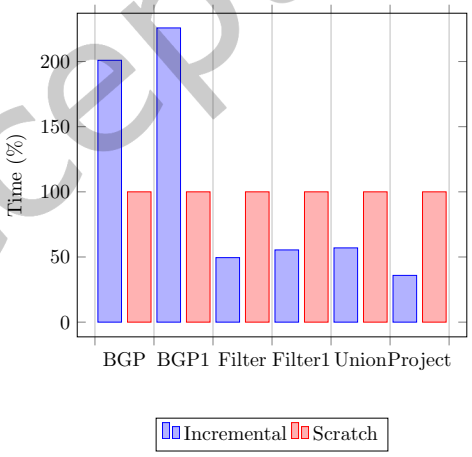
Results of the different operations for BSBM Query 1.



Results of the different operations for BSBM Query 2.



Results of the different operations for BSBM Query 3.



Results of the different operations for BSBM Query 4.

Fig. 6. BSBM Queries, decomposed into its different operations. Relative execution time is shown for each operation with the from-scratch approach set at 100%.

Conversely, we also tested the following star-shaped pattern over the synthetic data used to test the path queries:

```
SELECT ?x ?y1 ?y2 ?y3 ?y4
WHERE {
    ?x edge:link ?y1 .
    ?x edge:link ?y2 .
    ?x edge:link ?y3 .
```

Fig. 7. Snapshot of the Berlin SPARQL Benchmark and its relations between nodes, highlighting two products and their features that share a producer together with all products sharing the same producer.

```
        ?x edge:link ?y4 .
    }
```

It turns out, however, that here the incremental approach still beats the from-scratch approach, like it was for the path query. The result is shown in Figure 10.

5.3 Aggregation

Further expansion on our work includes designing and implementing incremental maintenance of aggregate queries. As already indicated by Gupta and Mumick [20], this can in principle be achieved for incrementally computable aggregate functions such as AVG, COUNT and SUM. Let us develop this further for SUM.

Consider a K-relation $R : \Omega \to \mathbb{Z}$ with schema $\{A, B\}$. Using commonly used notation [14] we can define the operation $\gamma_{A;\mathrm{SUM}(B)}$, which groups by A and sums B-values, as follows, working on K-relations. For each $t \in \Omega$, let $\bar{t}$ be the tuple with schema $\{A, C\}$ defined by setting $\bar{t}(A) := t(A)$, and

$$\bar{t}(C) := \sum_{s \in \Omega : s(A) = t(A)} s(B) \cdot R(s).$$

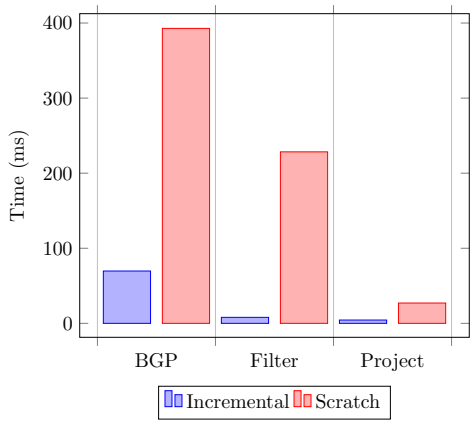


Fig. 8. Measurements of the incremental and from-scratch method on a hop-like query using the Berlin SPARQL Benchmark data.

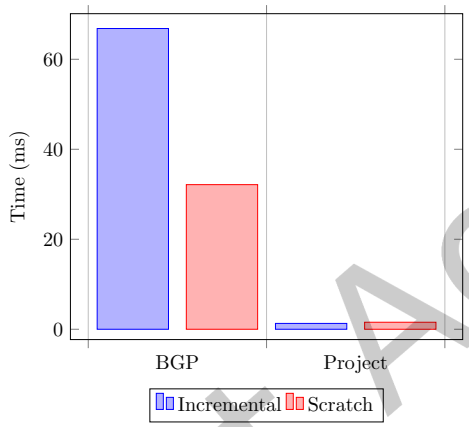


Fig. 9. Result of a star-shaped query applied to the BSBM data.

Fig. 10. Result of a star-shaped query applied to the synthethic data.

Note that the B-value of a tuple s is multiplied by its multiplicity as given by its annotation $R(s)$. Then $\gamma_{A;\mathrm{SUM}(B)}(R)$ is defined to be the K-relation

$$\{\bar{t} \mid t \in \Omega\} \longrightarrow \mathbb{Z} : \bar{t} \mapsto 1.$$

Note how, by definition, resulting solution mappings are always annotated by 1. Therefore, in what follows we will feel free to treat the results of aggregate operations simply as sets of tuples, since the annotation is understood to be 1.

Incremental maintenance of aggregation can be approached as follows. For two aggregation results S_1 and S_2 with schema $\{A, C\}$ as above, we define the following addition operation:

$$S_1 \oplus_{A;C} S_2 := \{t_1 \in S_1 \mid \neg\exists t_2 \in S_2 : t_1(A) = t_2(A)\} \cup \{t_2 \in S_2 \mid \neg\exists t_1 \in S_1 : t_1(A) = t_2(A)\} \\ \cup \{t_1 +_{A;C} t_2 \mid t_1 \in S_1 \,\&\, t_2 \in S_2 \,\&\, t_1(A) = t_2(A)\}.$$

Here, by $t_1 +_{A;C} t_2$ we mean the tuple t obtained by setting $t(A) := t_1(A) = t_2(A)$ and $t(C) := t_1(C) + t_2(C)$. We can verify that the following holds for any input K-relations R_1 and R_2 as above:

$$\gamma_{A;\mathrm{SUM}(B)\to C}(R_1 + R_2) = \gamma_{A;\mathrm{SUM}(B)\to C}(R_1) \oplus_{A;C} \gamma_{A;\mathrm{SUM}(B)\to C}(R_2).$$

Interpreting R_1 as the result of a pattern P and R_2 as ΔP, the above provides a delta rule for the aggregate query $\gamma_{A;\mathrm{SUM}(B)\to C}(P)$.

We implement $S_1 \oplus_{A;C} S_2$ in SQL as follows:

```
SELECT A, SUM(C) as C
FROM (SELECT * FROM S1 UNION ALL SELECT * FROM S2)
GROUP BY A
```

The efficiency of incrementally maintaining aggregate queries cannot be taken for granted, however. Although computing $\gamma_{A;\mathrm{SUM}(B)\to C}(\Delta P)$ will be cheap since ΔP is small, the operator $\oplus_{A;C}$ may be expensive. An exhaustive performance study is beyond the scope of the present paper, where we are focusing on formal correctness of incremental maintenance specifically for the SPARQL algebra.

However, we have performed a preliminary experiment based on the following SPARQL aggregate query:

```
SELECT ?p (SUM(?value1))
WHERE {
    ?p b:PropNum1 ?value1 .
}
GROUP BY ?p;
```

In our formalism, this is $\gamma_{?p;\mathrm{SUM}(?\mathrm{value1})}(P)$, where P is the underlying BGP. We tested incremental maintenance of this query on RDF graphs where the result of P consists of m products having n numerical properties (`b:PropNum1`) each. We let m range over 1K, 5K, 10K, 20K and 50K, and let n be 50 or 100. We generate ΔP by choosing 10 products at random; for each such product, we delete 5 of its numerical properties and insert 5 new ones. Since the size of ΔP is fixed at 100 tuples, its relative size decreases as m increases. Thus we expect to see that incremental maintenance becomes better (compared to from scratch) as m increases. This is confirmed by our results shown in Figure 11.

5.4 Discussion

Since the incremental view maintenance algorithm operates using auxiliary data tables, its memory footprint is larger than that of the recomputation of views from scratch. In our first experiment for example, the performance benefit of the incremental approach relative to the from-scratch approach decreased for larger data sets; this is likely due to increased memory requirements of the incremental method. We expect that the incremental approach would still yield better results with bigger data than the from-scratch approach, given more memory resources. As mentioned already, the BSBM queries do not tend to be good indicators for performance measurements as they have only a few results for each query. Considering the different operations within the queries, it became apparent that the BGPs with a multitude of triple patterns did not benefit from the incremental view maintenance, while the other operations did. Furthermore, the BSBM queries exhibit star-shaped patterns, with a central subject and several predicates extending from it. This type of query is typically efficient in SPARQL [11] and hence the from-scratch approach works rather well as illustrated in Figure 9. Hop or chain patterns, on the contrary, in which sequences of triples are linked, appear to benefit much more from our incremental maintenance approach; this was found in our first and third experiment. However, in general there will be a multitude of factors influencing the relative performance of incremental versus from-scratch, and a simple "star versus path shape" hypothesis cannot capture all intricacies, as was illustrated in our very last experiment.

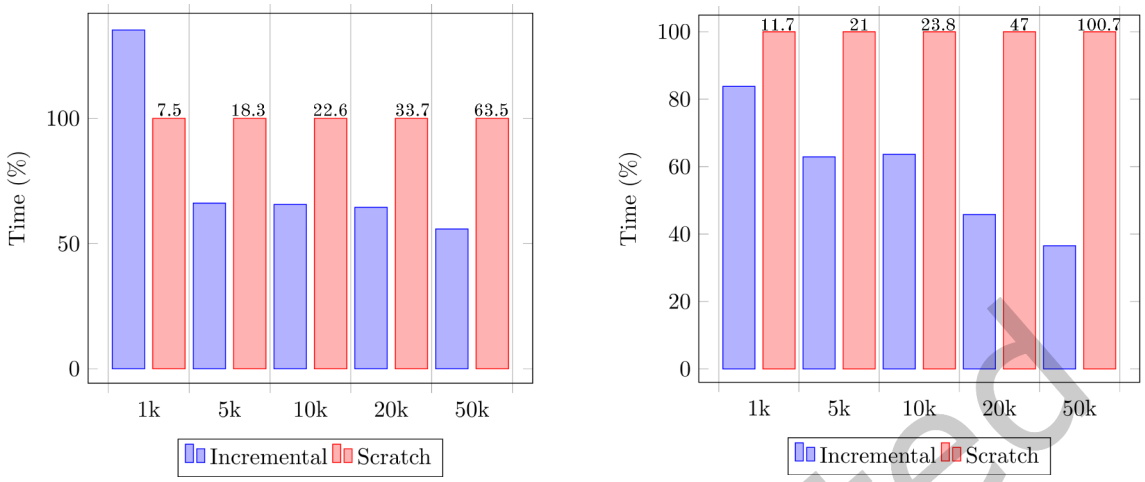


Fig. 11. Relative speed of incremental maintenance of an aggregate query compared to from-scratch. Absolute timing (in ms) of the from-scratch computation are also indicated. Left for $n = 50$ and right for $n = 100$.

We can summarize the potential limitations of our work reported in this paper as follows. First, our results cannot yet be used to reliably predict when incremental maintenance wins over from-scratch recomputation. Indeed, the community is lacking a dedicated benchmark for incremental view maintenance that would allow such predictions. Second, the method we have shown for aggregation is developed for SUM but can be extended to COUNT and AVG aggregation. Third, since the manipulation of sets of solution mappings is well supported by relational operators, we have implemented our method using a relational database system. It remains to implement our method by directly adapting an existing SPARQL engine.

6 Conclusion

In this article, we show that incremental view maintenance for SPARQL patterns can be achieved by borrowing from approaches in the relational database setting, thereby bridging the discrepancies caused by the schematic heterogeneity of sets of solution mappings, and handling the particularities of SPARQL algebra operators such as Diff and Minus. Our main goal has been to advance the incremental view maintenance approach at the conceptual level, proposing a feasible and correctly operating algorithm.

We repeat the research questions formulated in the Introduction, and summarize how they are answered in this article.

(RQ1) How can incremental view maintenance techniques from the relational setting be adapted to SPARQL, given the multiset semantics and heterogeneous solution mappings that arise from SPARQL's algebra? In Section 4.1 we recalled the counting algorithm [20], one of the standard incremental view maintenance algorithms in the relational data context. In Section 4.2 we introduced our approach to adapt the counting algorithm to the SPARQL context. We have been careful to respect the multiset semantics of SPARQL [21]. We did this by formally working in the framework of annotated sets of solution mappings, introduced in Section 3. An important feature of our approach, made explicit in Section 4.3, is that we treat BGPs as the unit of querying: each BGP is handled in its entirety rather than as a sequence of pairwise joins of triple patterns.

(RQ2) A graph pattern in SPARQL is an expression composed of BGPs using the operators of the SPARQL algebra [21]: Join, LeftJoin (known as OPTIONAL), Filter, Union, Minus, and Project (known as SELECT). How can these be handled in an incremental framework? In our approach, the delta $\Delta P(G)$ arising from an update ΔG

on G is represented by an annotated RDF graph as formalized in Section 3.4. Then in Sections 4.4 and 4.5 we systematically go through the operators of the SPARQL algebra mentioned above and formally derive ΔP in each case. Moreover, in Section 5.3 we show how aggregation through summation can be handled as well.

(RQ3) To what extent can the adapted algorithm be feasibly implemented and evaluated on realistic RDF data and SPARQL queries? In Sections 5.1 and 5.2 we provide an implementation and assess its feasibility through experiments on synthetic datasets and the Berlin SPARQL Benchmark.

Many directions remain for further work. A better understanding on the performance of incremental computation of aggregate queries remains to be extensively explored. An added complication is that, in contrast to SUM, other useful aggregate functions, including the median, are not straightforward to maintain incrementally. Another important direction is the further performance analysis of incremental view maintenance for SPARQL patterns, not only our approach, but possibly conducting further research including alternative methods, both from the relational database field [19], or from the incremental graph mining angle [2, 13]. Even in the relational setting, little performance comparison research has been conducted, except for the DBToaster work [26]. In particular, specifics of dataset and query characteristics that are important in a semantic web context should be taken into account. Encompassing empirical studies giving general guidance to the question "under which circumstances can we expect incremental maintenance to win over from-scratch recomputation of a view?" remain largely missing in the research literature.

In any case, performant querying in the decentralized web of PODs will be crucial for a real Web3.0 breakthrough. We consider views and their incremental maintenance as an instrumental element in this endeavor. The maintenance of views under changes of the base relations is important not only at the level of the PODs, but it can equally well be applied at the level of aggregators [43] which collect query results from multiple PODs.

References

[1] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[2] Ammar, K., McSherry, F., Salihoglu, S., and Joglekar, M. (2018). Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflow. *Proceedings of the VLDB Endowment*, 11(6):691–704.

[3] Arenas, M., Pérez, J., and Gutierrez, C. (2009). On the semantics of SPARQL. In De Virgilio, R., Giunchiglia, F., and Tanca, L., editors, *Semantic Web Information Management—A Model-Based Perspective*, pages 281–307. Springer.

[4] Asma, Z., Hernández, D., Galárraga, L., Flouris, G., Fundulaki, I., and Hose, K. (2024). NPCS: Native provenance computation for SPARQL. In Chua, T.-S., Ngo, C.-W., et al., editors, *Proceedings WWW*, pages 2085–2093. ACM.

[5] Bizer, C. and Schultz, A. (2009). The Berlin SPARQL benchmark. *International Journal on Semantic Web & Information Systems*, 5(2):1–24.

[6] Bizer, C. and Schultz, A. (2010). Berlin SPARQL benchmark (bsbm), dataset specification. http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/Dataset/index.html. Accessed: 2025-09-01.

[7] Bizer, C. and Schultz, A. (2013). Berlin SPARQL benchmark. http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/. Accessed: 2025-09-01.

[8] Blakeley, J., Larson, P.-Å., and Zhou, J. (1986). Efficiently updating materialized views. In *SIGMOD International Conference on Management of Data*, pages 61–71. Reprinted [19, Chapter 13].

[9] Chirkova, R. and Yang, J. (2012). Materialized Views. *Foundations and Trends in Databases*, 4(4):295–405.

[10] Dedecker, R., Slabbinck, W., Wright, J., et al. (2022). What's in a Pod? A knowledge graph interpretation for the Solid ecosystem. In Saleem, M. et al., editors, *Proceedings 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs*, volume 3279 of *CEUR Workshop Proceedings*, pages 81–96.

[11] DuCharme, B. (2013). *Learning SPARQL*. O'Reilly Media, Sebastopol, CA, 2 edition.

[12] Esposito, C., Horne, R., Robaldo, L., Buelens, B., and Goesaert, E. (2023). Assessing the solid protocol in relation to security and privacy obligations. *Information*, 14(7):411.

[13] Fan, W., Wang, X., and Wu, Y. (2013). Incremental graph pattern matching. *ACM Transactions on Database Systems*, 38(3):1–47.

[14] Garcia-Molina, H., Ullman, J., and Widom, J. (2009). *Database Systems: The Complete Book*. Prentice Hall.

[15] Geerts, F., Unger, T., Karvounarakis, G., et al. (2016). Algebraic structures for capturing the provenance of SPARQL queries. *Journal of the ACM*, 63(1):7:1–7:63.

[16] Griffin, T. and Kumar, B. (1998). Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3):22–27.

[17] Griffin, T. and Libkin, L. (1995). Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, volume 24:2 of *SIGMOD Record*, pages 328–339. ACM Press. Reprinted [19, Chapter 15].
[18] Griffin, T., Libkin, L., and Trickey, H. (1997). An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transaction on Knowledge and Data Engineering*, 9(3):508–511.
[19] Gupta, A. and Mumick, I., editors (1999). *Materialized Views: Techniques, Implementations, and Applications*. MIT Press.
[20] Gupta, A., Mumick, I., and Subrahmanian, V. (1993). Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22:2 of *SIGMOD Record*, pages 157–166. ACM Press. Reprinted [19, Chapter 14].
[21] Harris, S. and Seaborne, A. (2013). SPARQL 1.1 query language. W3C Recommendation.
[22] Hendler, J., Lassila, O., and Berners-Lee, T. (2001). The semantic web. *Scientific American*, 284(5):34–43.
[23] Hernández, D., Galárraga, L., and Hose, K. (2021). Computing how-provenance for SPARQL queries via query rewriting. *Proceedings of the VLDB Endowment*, 14(13):3389–3401.
[24] Ibragimov, D., Hose, K., Bach Pedersen, T., and Zimányi, E. (2016). Optimizing aggregate SPARQL queries using materialized RDF views. In Groth, P., Simperl, E., et al., editors, *Proceedings 15th International Semantic Web Conference*, volume 9981 of *Lecture Notes in Computer Science*, pages 341–359. Springer.
[25] Kaminski, M., Kostylev, E., and Cuenca Grau, B. (2017). Query nesting, assignment, and aggregation in SPARQL 1.1. *ACM Transactions on Database Systems*, 42(3):17:1–17:46.
[26] Koch, C. et al. (2014). DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278.
[27] Lambrechts, N., Goesaert, E., Buelens, B., Laes, E., Raeymaekers, P., Rijne, W., Neyens, K., Van Meerbeeck, A., Tambuyzer, E., and Wouters, A. (2023). We are: Giving citizens control over their data. *International Journal of Integrated Care*, 23(S1).
[28] Larson, P.-Å. and Zhou, J. (2007). Efficient maintenance of materialized outer-join views. In *Proceedings 23th International Conference on Data Engineering*, pages 56–65. IEEE Computer Society.
[29] Menendez, E., Casanova, M., Vidal, V., et al. (2016). Incremental maintenance of materialized SPARQL-based linkset views. In Hartmann, S. and Ma, H., editors, *Proceedings 27th International Conference on Database and Expert Systems Applications, Part II*, volume 9828 of *Lecture Notes in Computer Science*, pages 68–83. Springer.
[30] Pang, Y., Yang, L., Zou, L., and Özsu, M. T. (2023). gfov: A full-stack sparql query optimizer & plan visualizer. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 5081–5085. Association for Computing Machinery.
[31] Pop Stefanija, A., Buelens, B., Goesaert, E., Lenaerts, T., Pierson, J., and Van den Bussche, J. (2024). Towards a Solid acceptance of the decentralized Web of personal data: Societal and technological convergence. *Communications of the ACM*, 67(1):43–46.
[32] Qian, X. and Wiederhold, G. (1991). Incremental recomputation of active relational expressions. *IEEE Transaction on Knowledge and Data Engineering*, 3(3):337–341.
[33] Raasveld, M. and Mühleisen, H. (2019). DuckDB: An embeddable analytical database. In *Proceedings 2019 International Conference on Management of Data*, pages 1981–1984. ACM.
[34] Ragab, M., Savateev, Y., Oliver, H., Tiropanis, T., Poulovassilis, A., Chapman, A., and Roussos, G. (2024). Espresso: A framework to empower search on the decentralized web. *Data Science and Engineering*, 9(4):431–448.
[35] RDFLib (2002). RDFLib. https://rdflib.readthedocs.io/en/stable/index.html. Accessed: March 30th.
[36] Schmedding, F. (2011). Incremental SPARQL evaluation for query answering on linked data. In Hartig, O. et al., editors, *Proceedings 2nd International Workshop on Consuming Linked Data*, volume 782 of *CEUR Workshop Proceedings*.
[37] Solid (2025). The Solid Project. https://solidproject.org. Accessed: March 30th.
[38] Svingos, C., Hernich, A., et al. (2023). Foreign keys open the door for faster incremental view maintenance. *Proceedings of the ACM on Management of Data*, 1(1):40:1–40:25.
[39] Taelman, R. and Verborgh, R. (2023). Link traversal query processing over decentralized environments with structural assumptions. In Payne, T., Presutti, V., Qi, G., et al., editors, *Proceedings 22nd International Semantic Web Conference*, volume 14265 of *Lecture Notes in Computer Science*, pages 3–22. Springer.
[40] Taelman, R. and Verborgh, R. (2024). Demonstration of link traversal sparql query processing over the decentralized solid environment. In *EDBT*, pages 786–789.
[41] Vandenbrande, M. (2023). Aggregators to realize scalable querying across decentralized data sources. ISWC Doctoral Consortium.
[42] Vandenbrande, M., Jakubowski, M., Buelens, P. B. B., Ongenae, F., and Van den Bussche, J. (2024). POD-QUERY: Schema mapping and query rewriting for Solid pods. In Fundulaki, I., Kozaki, K., et al., editors, *Proceedings of the ISWC 2023 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice*, volume 3632 of *CEUR Workshop Proceedings*.
[43] Verborgh, R. (2020). Decentralizing personal data management with Solid: a hands-on workshop. https://rubenverborgh.github.io/SEMIC-Workshop-2020/. SEMIC Workshop.
[44] Verborgh, R. (2023). Re-decentralizing the Web, for good this time. In Seneviratne, O. and Hendler, J., editors, *Linking the World's Information*, pages 215–230. ACM.
[45] Vidal, V. et al. (2022). Publication and maintenance of RDB2RDF views externally materialized in enterprise knowledge graphs. *International Journal of Web Information Systems*, 18(5/6):255–285.

[46] VITO (2025). We Are. https://we-are-health.be. Accessed: March 30th.
[47] W3C Community Group (2024). Solid Protocol. https://solidproject.org/TR/protocol. Accessed: March 30th.
[48] World Wide Web Consortium (2025). Linked Web Storage Working Group. https://www.w3.org/groups/wg/lws/. Accessed: March 30th.
[49] Zhang, Q., Guo, D., Zhao, X., and Luo, L. (2022). Handling rdf streams: Harmonizing subgraph matching, adaptive incremental maintenance, and matching-free updates together. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 2580–2589.

A Generated SQL queries

A.1 Converted query of Query1 of the BSBM dataset

$BGP \leftarrow T_1(G), T_2(G), T_3(G), T_4(G), T_5(G).$:

```
SELECT G4.o AS label, G1.s AS product, G1.k_count AS k_count
FROM G G1, G G2, G G3, G G4, G G5
WHERE G1.p = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND G1.o = %ProductType% AND G1.s = G2.s AND G2.p = b:Feature AND G2.o = %Feat1%
AND G1.s = G3.s AND G3.p = b:Feature AND G3.o = %Feat2% AND G1.s = G4.s
AND G4.p = rdfs:label AND G1.s = G5.s AND G5.p = b:PropNum1
AND CAST(value1 AS INT) > 100;
```

A.2 Converted delta rules for Query1 of the BSBM dataset

(1) $\Delta\text{BGP} \leftarrow T_1(\Delta G), T_2(G), T_3(G), T_4(G), T_5(G)$:

```
SELECT G4.o AS label, G1.s AS product, G1.k_count AS k_count
FROM delta_G G1, G G2, G G3, G G4, G G5
WHERE G1.p = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND G1.o = %ProductType% AND G1.s = G2.s AND G2.p = b:Feature AND G2.o = %Feat1%
AND G1.s = G3.s AND G3.p = b:Feature AND G3.o = %Feat2% AND G1.s = G4.s
AND G4.p = rdfs:label AND G1.s = G5.s AND G5.p = b:PropNum1
AND CAST(value1 AS INT) > 100;
```

(2) $\Delta\text{BGP} \leftarrow T_1(G'), T_2(\Delta G), T_3(G), T_4(G), T_5(G)$:

```
SELECT G4.o AS label, G1.s AS product, G2.k_count AS k_count
FROM nu_G G1, delta_G G2, G G3, G G4, G G5
WHERE G1.p = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND G1.o = %ProductType% AND G1.s = G2.s AND G2.p = b:Feature AND G2.o = %Feat1%
AND G1.s = G3.s AND G3.p = b:Feature AND G3.o = %Feat2% AND G1.s = G4.s
AND G4.p = rdfs:label AND G1.s = G5.s AND G5.p = b:PropNum1
AND CAST(value1 AS INT) > 100;
```

(3) $\Delta\text{BGP} \leftarrow T_1(G'), T_2(G'), T_3(\Delta G), T_4(G), T_5(G)$:

```
SELECT G4.o AS label, G1.s AS product, G3.k_count AS k_count
FROM nu_G G1, nu_G G2, delta_G G3, G G4, G G5
WHERE G1.p = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND G1.o = %ProductType% AND G1.s = G2.s AND G2.p = b:Feature AND G2.o = %Feat1%
AND G1.s = G3.s AND G3.p = b:Feature AND G3.o = %Feat2% AND G1.s = G4.s
AND G4.p = rdfs:label AND G1.s = G5.s AND G5.p = b:PropNum1
AND CAST(value1 AS INT) > 100;
```

(4) $\Delta\text{BGP} \leftarrow T_1(G'), T_2(G'), T_3(G'), T_4(\Delta G), T_5(G)$:

```
SELECT G4.o AS label, G1.s AS product, G4.k_count AS k_count
FROM nu_G G1, nu_G G2, nu_G G3, delta_G G4, G G5
```

```
WHERE G1.p = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND G1.o = %ProductType% AND G1.s = G2.s AND G2.p = b:Feature AND G2.o = %Feat1%
AND G1.s = G3.s AND G3.p = b:Feature AND G3.o = %Feat2% AND G1.s = G4.s
AND G4.p = rdfs:label AND G1.s = G5.s AND G5.p = b:PropNum1
AND CAST(value1 AS INT) > 100;
```

(5) $\Delta\text{BGP} \leftarrow T_1(G'), T_2(G'), T_3(G'), T_4(G'), T_5(\Delta G)$:

```
SELECT G4.o AS label, G1.s AS product, G5.k_count AS k_count
FROM nu_G G1, nu_G G2, nu_G G3, nu_G G4, delta_G G5
WHERE G1.p = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND G1.o = %ProductType% AND G1.s = G2.s AND G2.p = b:Feature AND G2.o = %Feat1%
AND G1.s = G3.s AND G3.p = b:Feature AND G3.o = %Feat2% AND G1.s = G4.s
AND G4.p = rdfs:label AND G1.s = G5.s AND G5.p = b:PropNum1
AND CAST(value1 AS INT) > 100;
```

(6) The last query sums the different delta rules and adds up the k-values in case a solution mapping appears more than once.

```
SELECT r1.label, r1.product, r1.value1, SUM(r1.k_count) AS k_count
FROM temp_delta_BGP AS r1
GROUP BY r1.label, r1.product, r1.value1
HAVING SUM(r1.k_count) != 0;
```

If a solution mapping has 0 as a value for a k-value, it is removed from the delta query as it would not have any effect on the end result.

B Proofs

B.1 Proof of Theorem 3.1

We prove the equality for $(R_1 + R_2) \bowtie R_3 = (R_1 \bowtie R_3) + (R_2 \bowtie R_3)$. The other two equalities are established by similar arguments.

Let Ω_1, Ω_2 and Ω_3 be the sets of solution mappings annotated by R_1, R_2 and R_3 respectively. If we knew that $\Omega_1 = \Omega_2$, the equality would be straightforward to verify. This observation prompts us to introduce the following K-sets:

$$\bar{R}_1 : \Omega_1 \cup \Omega_2 \to \mathbb{Z} : \mu \mapsto \begin{cases} R_1(\mu) & \text{if } \mu \in \Omega_1; \\ 0 & \text{if } \mu \in \Omega_2 - \Omega_1. \end{cases} \tag{1}$$

$$\bar{R}_2 : \Omega_1 \cup \Omega_2 \to \mathbb{Z} : \mu \mapsto \begin{cases} R_2(\mu) & \text{if } \mu \in \Omega_2; \\ 0 & \text{if } \mu \in \Omega_1 - \Omega_2. \end{cases} \tag{2}$$

Since $\bar{R}_1$ and $\bar{R}_2$ have the same underlying set $\Omega_1 \cup \Omega_2$, it is easy to see, as just mentioned, that $(\bar{R}_1 + \bar{R}_2) \bowtie R_3 = (\bar{R}_1 \bowtie R_3) + (\bar{R}_2 \bowtie R_3)$. So, we are done if we can show the following two equalities:

(1) $(R_1 + R_2) \bowtie R_3 = (\bar{R}_1 + \bar{R}_2) \bowtie R_3$;
(2) $(R_1 \bowtie R_3) + (R_2 \bowtie R_3) = (\bar{R}_1 \bowtie R_3) + (\bar{R}_2 \bowtie R_3)$.

For the first equality, note that the K-sets on both sides clearly have the same underlying set Ω of solution mappings, namely, $\Omega = (\Omega_1 \cup \Omega_2) \bowtie \Omega_3$. So, take any $\mu \in \Omega$. We derive:

$$((R_1 + R_2) \bowtie R_3)(\mu) = \sum_{\substack{(\nu, \mu_3) \in (\Omega_1 \cup \Omega_2) \times \Omega_3 \\ \nu \cup \mu_3 = \mu}} (R_1 + R_2)(\nu) \cdot R_3(\mu_3)$$

$$
\begin{aligned}
&= \sum_{\substack{(\nu,\mu_3)\in(\Omega_1\cap\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} (R_1(\nu)+R_2(\nu))\cdot R_3(\mu_3)\\
&\quad + \sum_{\substack{(\nu,\mu_3)\in(\Omega_1-\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} R_1(\nu)\cdot R_3(\mu_3)\\
&\quad + \sum_{\substack{(\nu,\mu_3)\in(\Omega_2-\Omega_1)\times\Omega_3\\ \nu\cup\mu_3=\mu}} R_2(\nu)\cdot R_3(\mu_3)\\
&= \sum_{\substack{(\nu,\mu_3)\in(\Omega_1\cap\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} (\bar{R}_1(\nu)+\bar{R}_2(\nu))\cdot R_3(\mu_3)\\
&\quad + \sum_{\substack{(\nu,\mu_3)\in(\Omega_1-\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} (\bar{R}_1(\nu)+\bar{R}_2(\nu))\cdot R_3(\mu_3)\\
&\quad + \sum_{\substack{(\nu,\mu_3)\in(\Omega_2-\Omega_1)\times\Omega_3\\ \nu\cup\mu_3=\mu}} (\bar{R}_1(\nu)+\bar{R}_2(\nu))\cdot R_3(\mu_3)\\
&= \sum_{\substack{(\nu,\mu_3)\in(\Omega_1\cup\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} (\bar{R}_1(\nu)+\bar{R}_2(\nu))\cdot R_3(\mu_3)\\
&= ((\bar{R}_1+\bar{R}_2)\bowtie R_3)(\mu).
\end{aligned}
$$

In the second equality, the K-sets on both sides have Ω as underlying set of solution mappings, Take any $\mu\in\Omega$. Then $((R_1\bowtie R_3)+(R_2\bowtie R_3))(\mu)$ equals

$$
\begin{aligned}
&\sum_{\substack{(\nu,\mu_3)\in\Omega_1\times\Omega_3\\ \nu\cup\mu_3=\mu}} R_1(\nu)\cdot R_3(\mu_3) + \sum_{\substack{(\nu,\mu_3)\in\Omega_2\times\Omega_3\\ \nu\cup\mu_3=\mu}} R_2(\nu)\cdot R_3(\mu_3)\\
&\qquad = \sum_{\substack{(\nu,\mu_3)\in(\Omega_1\cup\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} \bar{R}_1(\nu)\cdot R_3(\mu_3) + \sum_{\substack{(\nu,\mu_3)\in(\Omega_1\cup\Omega_2)\times\Omega_3\\ \nu\cup\mu_3=\mu}} \bar{R}_2(\nu)\cdot R_3(\mu_3)
\end{aligned}
$$

which equals $((\bar{R}_1\bowtie R_3)+(\bar{R}_2\bowtie R_3))(\mu)$ as desired.

B.2 Proof of Theorem 4.3

The fundamental observation is that for any triple pattern T and K-graphs H_1 and H_2, we have $T(H_1+H_2)=T(H_1)+T(H_2)$. In proof, let $T=(u,v,w)$, let $X=\{u,v,w\}\cap V$, and let $H_i: G_i\to\mathbb{Z}$ for $i=1,2$. We observe:

$$
\begin{aligned}
T(H_1+H_2) &= \{\mu\colon X\to U \mid (\mu(u),\mu(v),\mu(w))\in G_1\cup G_2\}\to\mathbb{Z}\colon \mu\mapsto (H_1+H_2)(\mu)\\
&= \{\mu\colon X\to U \mid (\mu(u),\mu(v),\mu(w))\in G_1-G_2\}\to\mathbb{Z}\colon \mu\mapsto H_1(\mu)\\
&\quad \cup \{\mu\colon X\to U \mid (\mu(u),\mu(v),\mu(w))\in G_2-G_1\}\to\mathbb{Z}\colon \mu\mapsto H_2(\mu)\\
&\quad \cup \{\mu\colon X\to U \mid (\mu(u),\mu(v),\mu(w))\in G_2\cap G_1\}\to\mathbb{Z}\colon \mu\mapsto H_1(\mu)+H_2(\mu).
\end{aligned}
$$

On the other hand, using the definition of sum of two K-sets, we see that $T(H_1)+T(H_2)$ results in exactly the same K-set just described. Hence $T(H_1+H_2)=T(H_1)+T(H_2)$ as desired.

It follows in particular that $T(G') = T(G) + T(\Delta G)$. We can now prove the theorem by induction on n, using the distributivity of Join over Union (Theorem 3.1), and the commutativity and associativity of Join and of Union. A more detailed proof in a more general form, which refers to Theorem 4.5, can be found in Appendix B.3.

B.3 Proof of Theorem 4.5

By induction on n. For the base case $n = 2$ we reason as follows:

$$\begin{aligned}
P(G') &= P_1(G') \bowtie P_2(G') \\
&= (P_1(G) + \Delta P_1(G)) \bowtie (P_2(G) + \Delta P_2(G)) \\
&= P_1(G) \bowtie P_2(G) + \Delta P_1(G) \bowtie P_2(G) + P_1(G) \bowtie \Delta P_2(G) + \Delta P_1(G) \bowtie \Delta P_2(G) \\
&= P(G) + \Delta P_1(G) \bowtie P_2(G) + (P_1(G) + \Delta P_1(G)) \bowtie \Delta P_2(G) \\
&= P(G) + \Delta P_1(G) \bowtie P_2(G) + P_1(G') \bowtie \Delta P_2(G)
\end{aligned}$$

which is the desired result.

Now assume $n > 2$ and let $Q = P_2 \bowtie P_3 \bowtie \cdots \bowtie P_n$. Reasoning as above and using the induction hypothesis on Q we obtain the desired result:

$$\begin{aligned}
P(G') &= P_1(G') \bowtie Q(G') \\
&= (P_1(G) + \Delta P_1(G)) \bowtie (Q(G) + \Delta Q(G)) \\
&= P_1(G) \bowtie Q(G) + \Delta P_1(G) \bowtie Q(G) + P_1(G) \bowtie \Delta Q(G) + \Delta P_1(G) \bowtie \Delta Q(G) \\
&= P(G) + \Delta P_1(G) \bowtie Q(G) + (P_1(G) + \Delta P_1(G)) \bowtie \Delta Q(G) \\
&= P(G) + \Delta P_1(G) \bowtie P_2(G) \bowtie P_3(G) \bowtie \cdots \bowtie P_n(G) \\
&\qquad + P_1(G') \bowtie \Delta P_2(G) \bowtie P_3(G) \bowtie \cdots \bowtie P_n(G) \\
&\qquad + P_1(G') \bowtie P_2(G') \bowtie \Delta P_3(G) \bowtie \cdots \bowtie P_n(G) \\
&\qquad \vdots \\
&\qquad + P_1(G') \bowtie P_2(G') \bowtie P_3(G') \bowtie \cdots \bowtie \Delta P_n(G).
\end{aligned}$$

B.4 Proof of Theorem 4.6

Let us introduce the following abbreviations:

$$\begin{aligned}
A &= E \ominus (F_1 + F_2) \\
B_1 &= E \ominus F_1 \\
B_2 &= E \bowtie F_{12} \\
B &= B_1 + B_2
\end{aligned}$$

Let $t \in A$. Note that $A(t) = E(t)$. We want to show that $B(t) = A(t)$. We distinguish two cases.

(1) $t|_Y \in F_2$. Since $t \in A$, we know $t|_Y \notin F_1 + F_2$, so this case is only possible if $F_2(t|_Y) = -F_1(t|_Y)$. We then get $B_1(t) = 0$ and $B_2(t) = E(t)$, so $B(t) = B_2(t) = E(t) = A(t)$ as desired.
(2) $t|_Y \notin F_2$. Then also $t|_Y \notin F_1$, again because we know $t|_Y \notin F_1 + F_2$. So $B_1(t) = E(t)$. Moreover, in this case clearly $t \notin B_2$, so here $B(t) = B_1(t) = E(t) = A(t)$ as desired.

For the converse direction, let $t \in B$. In particular, $t \in E$. We distinguish two cases.

(1) $t|_Y \in F_{12}$. In principle there are two possibilities:

- $F_{12}(t|_Y) = 1$, so $B_2(t) = R(t)$. In this case $F_2(t|_Y) = -F_1(t|_Y)$, so (i) $t|_Y \notin F_1 + F_2$ and (ii) $F_1(t|_Y) \neq 0$. By (ii) we have $t \notin B_1$ so $B(t) = B_2(t) = E(t) \neq 0$. By (i) we also have $A(t) = E(t) = B(t)$ as desired.
- $F_{12}(t|_Y) = -1$, so $t|_Y \notin F_1$. Then $B_1(t) = E(t)$ and $B_2(t) = -E(t)$, so $B(t) = 0$, which contradicts the given that $t \in B$. So, this possibility cannot occur.

(2) $t|_Y \notin F_{12}$, so $t \notin B_2$, whence $B(t) = B_1(t) = E(t)$. Since $B(t) \neq 0$, we have $B_1(t) \neq 0$, so $t|_Y \notin F_1$. Also $t|_Y \notin F_2$, for otherwise we would have $F_{12}(t|_Y) = -1$, which contradicts the assumption that $t|_Y \notin F_{12}$. Hence, $t \in E - (F_1 \cup F_2) = A$, and $A(t) = E(t) = B_1(t) = B(t)$ as desired.

Received 31 March 2025; revised 13 January 2026; accepted 13 January 2026