Artificial Intelligence 269 (2019) 76–136

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Maintenance of datalog materialisations revisited

Boris Motik *, Yavor Nenov, Robert Piro, Ian Horrocks

Department of Computer Science, University of Oxford, United Kingdom

ARTICLE INFO

Article history:
Received 24 July 2017
Received in revised form 18 June 2018
Accepted 16 December 2018
Available online 4 January 2019

Keywords:
Datalog
Materialisation maintenance
Reasoning
Incremental updates

ABSTRACT

Datalog is a rule-based formalism that can axiomatise recursive properties such as reachability and transitive closure. Datalog implementations often *materialise* (i.e., pre-compute and store) all facts entailed by a datalog program and a set of explicit facts. Queries can thus be answered directly in the materialised facts, which is beneficial to the performance of query answering, but the materialised facts must be updated whenever the explicit facts change. Rematerialising all facts 'from scratch' can be very inefficient, so numerous *materialisation maintenance* algorithms have been developed that aim to efficiently identify the facts that require updating and thus reduce the overall work. Most such approaches are variants of the *counting* or *Delete/Rederive* (DRed) algorithms. Algorithms in the former group maintain additional data structures and are usually applicable only if datalog rules are not recursive, which limits their applicability in practice. Algorithms in the latter group do not require additional data structures and can handle recursive rules, but they can be inefficient when facts have multiple derivations. Finally, to the best of our knowledge, these approaches have not been compared and their practical applicability has not been investigated. Datalog is becoming increasingly important in practice, so a more comprehensive understanding of the tradeoffs between different approaches to materialisation maintenance is needed. In this paper we present three such algorithms for datalog with stratified negation: a new counting algorithm that can handle recursive rules, an optimised variant of the DRed algorithm that does not repeat derivations, and a new Forward/Backward/Forward (FBF) algorithm that extends DRed to better handle facts with multiple derivations. Furthermore, we study the worst-case performance of these algorithms and compare the algorithms' behaviour on several examples. Finally, we present the results of an extensive, first-of-a-kind empirical evaluation in which we investigate the robustness and the scaling behaviour of our algorithms. We thus provide important theoretical and practical insights into all three algorithms that will provide invaluable guidance to future implementors of datalog systems.

1. Introduction

Datalog [1] is a prominent rule-based formalism with a rich tradition in several communities. It has been studied as an expressive database query language; it is extensively used as a knowledge representation language in artificial intelligence;

* Corresponding author.
E-mail address: boris.motik@cs.ox.ac.uk (B. Motik).

https://doi.org/10.1016/j.artint.2018.12.004

and it is widely used on the Semantic Web, where the OWL 2 RL [2] profile of the OWL 2 ontology language has been designed specifically to support datalog reasoning. The prominence of datalog is primarily due to its ability to infer implicit facts using domain knowledge represented as 'if-then' rules where, if all atoms in the antecedent of a rule are true, then the consequent of the rule is true as well. The ability to form *recursive* rules lends datalog its expressivity: a fact derived by a rule can (possibly indirectly) trigger the same rule, and so the rules must be applied iteratively until a fixpoint is reached. This allows datalog to express important recursive queries such as graph reachability or transitive closure.

Due to its ability to declaratively and succinctly specify complex problems, datalog plays an increasingly important role in advanced applications that mix AI and data management techniques. For example, datalog fits naturally with regulatory compliance applications, which succinctly and declaratively describe a set of complex conditions over a domain of interest and then verify these conditions on given datasets. Along these lines, datalog was used for verifying compliance with engineering regulations in railway infrastructure [3], as well as to analyse healthcare provision records [4]. Another application area of datalog is enterprise data management, where datalog can support tasks ranging from data querying to data analysis and declarative modelling of business logic [5]. Datalog has also been used to declaratively specify network management policies [6]. Due the increasing interest in datalog, many commercial datalog systems have been implemented, such as Oracle's database[1] [7], LogicBlox,[2] GraphDB,[3] Blazegraph,[4] MarkLogic,[5] and Datomic.[6] This has lead to resurgent interest in datalog implementation techniques from both theoretical and practical points of view.

The main computational problem facing datalog systems is answering queries over the facts that logically follow from a set of explicitly given facts and a set of datalog rules. A common (but not the only) way to solve this problem is to precompute and store all implied facts in a preprocessing step; both the process and its output are commonly called *materialisation*. The *seminaïve* algorithm [1] can compute the materialisation efficiently by performing each inference only once. After such preprocessing, queries can be answered directly over the materialisation, which is usually very efficient since the rules do not need to be considered any further. Materialisations can be large, but they can usually be stored and handled on modern hardware as the available memory is continually increasing. Thus, materialisation is often the technique of choice in datalog systems.

The main drawback of such an approach to query answering is that, whenever explicit facts are added and/or deleted, the 'old' materialisation must be replaced with the 'new' materialisation that contains all facts entailed by the datalog rules and the updated explicit facts. Please note that we do not consider the problem of updating facts that are not explicit, a problem known as *belief revision* [8] and *view update* in the knowledge representation and the database communities, respectively. There is no unambiguous semantics for such updates, and computing the result of such updates is often of high computational complexity. Thus, practical applications usually allow updating only explicit facts, which has a natural semantics (i.e., an update produces the same results as recomputing the 'new' materialisation 'from scratch') and its complexity is not worse than for materialisation.

When the explicit facts do not change much, recomputing the 'new' materialisation 'from scratch' is likely to repeat most of the inferences from the 'old' materialisation and will therefore be very inefficient. Thus, several algorithms for *materialisation maintenance* have been developed, whose aim is to efficiently identify a subset of the materialisation that needs updating and thus reduce the work needed; in the database literature, these are often called *(materialised) view maintenance* algorithms. The key problem these algorithms must address is to identify facts from the materialisation that have no 'surviving' derivations and should thus be deleted. Numerous approaches to materialisation maintenance have been devised. As one can see from our comprehensive survey in Section 2, these can broadly be categorised into two groups. The first group consists of *counting* approaches that, roughly speaking, associate with each fact a counter keeping the number of the fact's derivations; thus, a fact can be deleted when its counter reaches zero. As we discuss in Section 5, all such approaches apart from one are applicable only to nonrecursive rules, and so they cannot be used in general datalog implementations. The approach by Dewan et al. [9] handles recursive rules by maintaining an array of counters per fact; however, as we discuss in Section 5, this algorithm is based on the inefficient *naïve* datalog evaluation strategy and is thus unlikely to be suitable for practice. All approaches in this group require additional memory to store the counters, which can be a considerable source of overhead. The second group of existing approaches consists of variants of the Delete/Rederive (DRed) algorithm. Roughly speaking, such approaches first eagerly overdelete all facts that depend (possibly indirectly) on a deleted fact, and then they rederive all facts that still hold after an update. These approaches do not require any additional memory and can handle recursive rules; however, eager overdeletion and the subsequent rederivation of facts can be very inefficient, particularly when facts participate in long inference chains. Despite an extensive body of existing work on materialisation maintenance, several important questions still remain unanswered.

First, counting approaches seem very natural and popular, but their inability to handle recursion is a significant practical drawback. An efficient and practical counting approach capable of handling recursive rules would complete the portfolio of algorithms available to datalog implementors. We present the first such algorithm in Section 5. Although inspired by Dewan

[1] http://docs.oracle.com/database/122/RDFRM/.
[2] http://www.logicblox.com.
[3] http://ontotext.com.
[4] http://www.blazegraph.com.
[5] http://www.marklogic.com.
[6] http://www.datomic.com.

et al. [9], our algorithm is much more sophisticated; for example, it is based on the seminaïve instead of the naïve algorithm in order to make the approach applicable to large amounts of data.

Second, as we discuss in detail in Section 6, various formalisations of the DRed algorithm found in the literature can repeat inferences unnecessarily: if k atoms matching a rule are deleted, the same rule instance will fire k times, each time deriving the same fact for overdeletion. This can considerably affect the performance of DRed and can become important in applications where updates should be handled very rapidly. Moreover, some variants of the DRed algorithm are also inefficient because they copy the 'old' materialisation in the process of updating it, rather than updating the materialisation in situ. This can be very inefficient, particularly on small updates. To address these problems, in Section 6 we present a new variant of the DRed algorithm that does not repeat derivations or copy the materialisation and is thus more suitable for practice.

Third, as we discuss in detail in Section 7, the 'eagerly overdelete, then rederive if needed' approach used in DRed can be a source of considerable overhead: many facts can be overdeleted when inference chains are long, but many of these facts are often rederived when facts have multiple derivations since derivations are then more likely to 'survive' an update. In such cases, it would be beneficial to use a more precise deletion strategy to prevent overdeletion and subsequent rederivation of many facts. To this end, in Section 7 we present a new Forward/Backward/Forward (FBF) algorithm that uses a combination of backward and forward chaining to check whether a fact still holds after an update and thus limit the effects of overdeletion. The algorithm is based on the idea of backward chaining from our Backward/Forward algorithm [10]. However, unlike the B/F algorithm, the FBF algorithm combines backward chaining with overdeletion, and it uses a strategy to determine the balance between the two. In this way, both DRed and B/F correspond to instances of FBF parameterised with suitable strategies.

Fourth, the theoretical worst-case behaviour of the different approaches has not been studied and remains unknown. All approaches lie in the same complexity class as materialisation: PTIME for data complexity and EXPTIME for combined complexity [11]. Nevertheless, it is interesting to compare the inferences of a materialisation maintenance algorithm with that of computing the 'old' and/or the 'new' materialisations and thus obtain hard bounds on an algorithm's performance. We prove that our versions of counting and DRed perform, in the worst case, all inferences from the 'old' and the 'new' materialisation combined; however, FBF can additionally consider all inferences from the 'old' involving recursive rules during backward chaining. These bounds are hard in the sense that there exist inputs on which our algorithms actually perform all of these inferences.

Fifth, there is currently no understanding of how individual algorithms perform on different kinds of inputs. To clarify this, in Section 8 we compare the performance of our three algorithms. In particular, we show that, if no rule instance stops being applicable due to an update (e.g., the program does not contain negation and an update only adds facts), then DRed and FBF are optimal, whereas the counting algorithm may not be optimal if the program contains recursive rules; however, if the rules are nonrecursive, the counting algorithm always exhibits ideal behaviour by considering only inferences that change due to the update, whereas DRed and FBF can consider more inferences. Furthermore, if the rules are recursive, for each algorithm there exist inputs on which the algorithm exhibits the worst-case behaviour, and inputs on which it considers just the changed inferences; moreover, the same input can simultaneously be 'good' for one algorithm but 'bad' for another. Thus, no algorithm can be deemed universally better, but our examples illustrate situations in which one algorithm might be preferable to another.

Sixth, in Section 9 we discuss various important implementation issues that, to the best of our knowledge, have not yet been considered systematically. For example, we discuss how to efficiently maintain various sets of facts, and how to match the rules to the facts in a way that guarantees nonrepetition of inferences.

Finally, apart from a comparison of DRed with nonrecursive counting by Urbani et al. [12], we are unaware of any attempts to compare different materialisation maintenance algorithms empirically. Thus, the designers of datalog systems currently have no way of making an informed decision about which algorithm to use. In Section 10 we present the results of an extensive evaluation where we subjected all three algorithms to two kinds of tests. First, our *robustness* experiments investigate whether an algorithm exhibits consistent performance on small updates regardless of which facts are changed. In many applications updates are usually small, so a robust algorithm with a consistent performance is more likely to be useful in practice. Second, our *scalability* experiments investigate how an algorithm's performance depends on the size of an update, and in particular at which point incremental update becomes less efficient than rematerialisation 'from scratch'. Our results paint a complex picture of the behaviour of our algorithms: the FBF algorithm seems effective in most, but not all cases on small updates, whereas the counting algorithm and sometimes DRed seem to be better at handling large updates. Based on these results, we draw many important conclusions that provide guidance for future developers of datalog-based systems.

This is a substantial extension of our earlier work published at the AAAI 2015 conference [10]. Additional material includes a completely new counting algorithm that can handle recursive rules, an extension of the B/F algorithm that also generalises DRed, an extensive theoretical and empirical comparison of all algorithms, and revised and extended correctness proofs.

2. Related work

Materialisation maintenance is a special case of *view maintenance*, where a database contains *base relations* and *views* defined by (possibly recursive) queries over the base relations. Views are *materialised* for efficiency—that is, their defining queries are evaluated and the results stored. When base relations are updated, view maintenance algorithms reflect these updates in the materialised views more efficiently than by reevaluating the defining queries. Gupta and Mumick [13] classified many approaches to view maintenance according to the view language, the kinds of update, and the amount of information available to the maintenance algorithm. Similarly, Vista [14] compared some of the existing approaches from the point of their applicability and ease of implementation. In the rest of this section, we summarise these results and survey solutions to related problems.

2.1. Maintaining views in relational databases

Views defined by nonrecursive queries can be maintained via queries that refer to the old database state and the updates. For example, let $V = \pi_X(R_1 \bowtie R_2)$ be a view defined as a natural join of relations R_1 and R_2 projected to attributes X, and assume that, for each $i \in \{1, 2\}$, relation R_i^+ contains facts to be added to R_i. Then, the changes to the materialisation of V are given by $V^+ = \pi_X[(R_1 \cup R_1^+) \bowtie (R_2 \cup R_2^+)] \setminus V$, which can be simplified to $V^+ = \pi_X(R_1^+ \bowtie R_2) \cup \pi_X(R_1 \bowtie R_2^+) \cup \pi_X(R_1^+ \bowtie R_2^+)$ using the equivalences of the relational algebra. If V stores duplicate tuples, then deletion from R_1 and R_2 can be computed analogously. For views without duplicates, a common solution is to count how many times a tuple has been derived; then, algebraic expressions analogous to V^+ are used to maintain these counts so that a tuple can be deleted when its count reaches zero.

These ideas have been used extensively in the literature. Blakeley et al. [15] proposed such an approach with derivation counting for SPJ views. Hanson [16] described a very similar approach and compared analytically the cost of view maintenance to the cost of view rematerialisation. Ceri and Widom [17] analysed key dependencies on the base relations to determine whether a view can contain duplicates, and they presented an approach for maintaining views using production rules. Roussopoulos [18] presented a data structure that facilitates easier maintenance of views with duplicates. Qian and Wiederhold [19] presented an approach for all of relational algebra with set semantics and without any additional bookkeeping (such as derivation counts), and Griffin et al. [20] corrected this algorithm so that all computed updates are minimal in a well-defined sense. Griffin and Libkin [21] extended this idea to relational algebra with multiset (aka bag) semantics and a duplicate elimination operator, and they proved an $O(n)$ upper complexity bound for projection- and product-free views and the asymptotic optimality of their approach. Since a view can often be maintained equivalently using several different algebraic expressions, extensions of database query optimisation techniques have been developed that are likely to identify more efficient such expressions [22,23].

2.2. Maintaining views in deductive databases

Nicolas and Yazdanian [24] and Gupta et al. [25] presented closely related counting-based approaches applicable to nonrecursive datalog programs that we discuss in detail in Section 5. Gupta et al. [26] extend these approaches to handle recursive rules, but only if the structure of the dataset guarantees that rules will not be applied recursively during materialisation. Wolfson et al. [27] presented the only counting-based approach we are aware of that can handle both recursive rules and arbitrary datasets; we discuss this approach and present a number of optimisations in Section 5.

Gupta et al. [25] presented the Delete/Rederive (DRed) algorithm that uses no extra bookkeeping, but handles fact deletion by first deleting all facts that depend on the deleted fact and then rederiving the facts that still hold after deletion; we discuss this approach in detail in Section 6. Ceri and Widom [17] follow the same general idea from DRed, but their solution is applicable only to nonrecursive datalog programs. The algorithm by Staudt and Jarke [28] is almost exactly the same as DRed, but it uses *maintenance* rules that copy the materialisation while computing the update; we discuss the details extensively in Section 6. Kotowski et al. [29] presented another variant of DRed optimised to use seminaïve evaluation in various stages of the algorithm. Finally, Lu et al. [30] presented an approach to view maintenance in constrained databases based on DRed.

Apt and Pugin [31] presented an algorithm that dynamically derives dependencies between relations to overdelete facts that potentially do not hold after the update, after which the standard materialisation algorithm rederives any missing facts. This approach is reminiscent of DRed, but overdeletion is done coarsely at the predicate level, and there is no discussion of optimisations such as not repeating derivations during update.

Barbieri et al. [32] presented an incremental approach to stream reasoning. At the time of insertion, each fact is associated with an expiration time based on the aperture of the sliding window, so deletion involves simply dropping all expired facts and insertion is handled by seminaïve evaluation.

Urbani et al. [12] compared the performance of DRed and counting. They did not explicitly state which version of counting they used in their work, but their (informal) discussion suggests that they used only one count per fact. Since they also used recursive datalog rules, their approach is either incomplete or it does not terminate in general.

Küchenhoff [33] and Urpí and Olivé [34] discussed how to declaratively maintain materialisations by evaluating maintenance programs. While one would intuitively expect the maintenance rules to use the update and the old state of the

database to compute the new state, in both cases the rules seem to use the new and the old state of the database to compute the update; hence, it is unclear to us how to use these rules for materialisation update. Harrison and Dietrich [35] presented an algorithm that seems to integrate backward and forward chaining, but the technical details of their approach are not precisely specified.

2.3. First-order incremental evaluation systems

The materialisation of certain classes of recursively defined relations can be maintained by evaluating first-order queries over the base and materialised relations, and auxiliary relations describing the update. Consider the following rules that define relation T as the transitive closure of relation E:

$$T(x, z) \leftarrow E(x, z) \qquad T(x, z) \leftarrow E(x, y) \wedge T(y, z)$$

Assume that T has been materialised for some E and that we wish to update E by a single tuple stored in E^+. Then, Dong and Topor [36] showed that required additions to T are given by the following (nonrecursive) first-order query:

$$Q(x, z) = E^+(x, z) \vee [\exists y.E^+(x, y) \wedge T(y, z)] \vee [\exists y.T(x, y) \wedge E^+(y, z)] \vee [\exists y_1, y_2.T(x, y_1) \wedge E^+(y_1, y_2) \wedge T(y_2, z)]$$

In fact, this approach can be extended to correctly handle incremental additions in weakly regular chain datalog programs [37].

First-Order Incremental Evaluation Systems (FOIESs) generalise these ideas. A FOIES for a query specifies how to (i) materialise the query result and zero or more additional *auxiliary* relations given a set of base relations, and (ii) update all materialised relations by evaluating first-order queries over the changes in the base relations. Dong and Su [38] present FOIESs for the transitive closure of acyclic graphs, directed graphs where deleted edges do not belong to the same strongly connected component, and 0–1 graphs; Dong and Pang [39] generalised these results w.r.t. deletion; Dong and Ramamohanarao [40] considered constrained transitive closure; and Pang et al. [41] considered all-pairs shortest paths in undirected graphs. Dong and Su studied the relationship of FOIES with increment boundedness and structural recursion in datalog [42], their space requirements [43], and how nondeterminism increases their power [44].

The existence of a deletion FOIES for transitive closure over arbitrary relations has long been an open problem. Dong et al. [45] proved that no FOIES exists that uses unary auxiliary relations only, and that, without extra space, the same result holds even if we allow aggregates in the maintenance queries. In contrast, Libkin and Wong [46] solved the problem using aggregation in maintenance queries; however, the extra space required is exponential so this approach is mainly of theoretical interest. Dong et al. [47] translated all these results into an SQL-based implementation.

Patnaik and Immerman [48] proposed the Dyn-FO class of problems where, for a constant-sized change in the input, the change to the output can be computed by evaluating first-order *maintenance queries* over the input, the changes, and auxiliary data. Transitive closure of directed acyclic or undirected graphs, minimum spanning forest, bipartiteness, k-edge connectivity, maximal matching in undirected graphs, lowest common ancestor in directed forests, and binary number multiplication are all in Dyn-FO. More recently, Datta et al. [49] solved a long-standing open problem by showing that directed reachability in arbitrary graphs is also in Dyn-FO. In a very interesting recent work, Zeume and Schwentick [50] studied fragments of Dyn-FO obtained by reducing the expressivity of the maintenance queries (e.g., to conjunctive, quantifier-free, or negation-free queries), they introduced and studied the Δ-Dyn-FO class its subclasses where even the auxiliary relations can be maintained incrementally, and they determined which of these subclasses are capable of maintaining first-order definable queries.

While these results can be useful in specialised settings that use restricted types of rules, the techniques we present in this paper are applicable to arbitrary datalog rules with stratified negation.

2.4. Related problems

A question closely related to view maintenance is to determine whether updating the base relations affects the view. Blakeley et al. [51] solved this problem for SPJ views and explicitly specified updates; Elkan [52] considered updates that are specified as queries; and Levy and Sagiv [53] considered updates that are specified as datalog programs.

When materialised views are stored separately from the base relations, it is desirable to maintain the views without accessing the base relations. Gupta et al. [54] determined when an SPJ view can be maintained without referring to the materialised view or the base relations; and Tompa and Blakeley [55] studied updates that can access the materialised view.

Deferred view maintenance generalises the view maintenance problem to cases when updates to the base relations and the views cannot be performed within a single transaction. For example, in a distributed of a data warehousing setting, base relations can evolve asynchronously, so standard view maintenance algorithms can be incorrect. Segev and Park [56] solve this problem by a judicious use of timestamps. Zhuge et al. [57] solve this problem without any bookkeeping information, by using *compensating queries* that take into account possible interactions due to data distribution. Colby et al. [58] use auxiliary tables to record information since the last view update with the goal of reducing the per-transaction overhead and view refresh times. Griffin and Hull [59] present a closely related technique for *hypothetical query answering*, where the goal is to answer a query over a database modified in a specific way, but without actually modifying the base relations at any point.

Related maintenance approaches have been proposed for *database snapshots*—replicated fragments of database tables that can be understood as views defined using selection over a single relation. In this context, Lindsay et al. [60] and Kähler and Risnes [61] showed how to identify updates to base relations from database logs.

2.5. Other related techniques in AI

In artificial intelligence, it is often necessary to maintain the set of conclusions of a logical theory w.r.t. updates. Such a setting typically consists of a *problem solver* that supports theorem proving for a logic of interest (usually first-order logic), and a *truth maintenance system* (TMS) that uses the problem solver as a black box. The TMS associates with each conclusion F of the problem solver one or more *justifications*—data structures recording one possible way of deriving F. Justifications can be propositional implications between the antecedents and the consequent [62], but they can also have other forms [63]. The TMS implements a search strategy that ensures consistency of justifications. TMSs can maintain datalog materialisations: whenever a rule r derives a fact H from facts $B_1, \ldots, B_n$, we add justification $B_1 \wedge \ldots \wedge B_n$ to H. Facts with no justification are not derivable from the explicit facts and should be deleted, so we can maintain a materialisation efficiently if we index its justifications. However, such an approach is unlikely to be efficient: as our experiments in Section 10 show, the total number of derivations (which is equal to the total number of justifications) is often orders of magnitude larger than the number of derived facts, and so storing all justifications of all derived facts is infeasible in practice.

A closely related problem arises in production systems, which need to efficiently identify rules that become applicable or stop being applicable as facts are derived and retracted from the working memory; one can view this as a 'continuous view maintenance' problem. This problem is commonly solved by compiling the production rules into a Rete network [64]—a graph-like data structure that encodes how changes in the working memory propagate between atoms in the rule bodies. Roughly speaking, this data structure cached partial instantiations of the rule bodies, thus allowing the changes in the applicable rule instances to be efficiently computed as the contents of the working memory changes. The number of partial rule instantiations, however, can be exponential in the size of the rules. Hence, the memory requirements of Rete networks can be very high, which generally prevents their application to the problem considered in this paper.

3. Preliminaries

Throughout this paper, we fix countably infinite, pairwise disjoint sets of *variables* (written x, y, z, etc.), *constants* (written a, b, c, etc.), and predicates (written P, Q, R, etc.). Each predicate is associated with a nonnegative integer called *arity*. A *term* is a constant or a variable. An *atom* is an expression the form $P(t_1, \ldots, t_k)$, where P is a k-ary predicate with $k \geq 0$ and each t_i, $1 \leq i \leq k$, is a term. A *(ground) fact* is a variable-free atom; a *dataset* is a finite set of facts; and $\mathcal{F}$ is the countably infinite set of all facts constructed using all constants and predicates. A *(datalog) rule* r is an expression of the form (1), where $0 \leq m \leq n$ and H and all B_i are atoms. Each rule r must be *safe*—that is, each variable occurring anywhere in r must also occur in r in at least one atom B_i with $1 \leq i \leq m$. A *(datalog) program* is a finite set of rules.

$$B_1 \wedge \cdots \wedge B_m \wedge \text{not}\, B_{m+1} \wedge \cdots \wedge \text{not}\, B_n \rightarrow H \tag{1}$$

For r a rule, $\mathsf{h}(r) := H$ is the *head atom* of the rule, $\mathsf{b}^+(r) := \{B_1, \ldots, B_m\}$ is the set of the *positive body atoms* of the rule, and $\mathsf{b}^-(r) := \{B_{m+1}, \ldots, B_n\}$ is the set of the *negative body atoms* of the rule. Note that $\mathsf{b}^+(r)$ and/or $\mathsf{b}^-(r)$ are allowed to be empty. A rule is *positive* if it has no negative body atoms; moreover, a program is *positive* if all of its rules are positive.

Semantic Web applications commonly represent data using the Resource Description Framework (RDF) [65], in which all facts and rules use a distinct ternary predicate Tr; atoms of the form $\mathsf{Tr}(t_1, t_2, t_3)$ are then commonly written as $\langle t_1, t_2, t_3 \rangle$ and are called *triple patterns*; facts of that form are called *triples*; and datasets of such facts are called *(RDF) graphs*.

In the database literature [1], it is common to distinguish *extensional* from *intensional* predicates, where only the former can occur in the data and only the latter can occur in rule heads. We, however, use the logic programming perspective, where such a distinction is not made. This perspective is particularly important in RDF, which uses a single predicate and thus cannot distinguish extensional from intensional predicates. Moreover, the logic programming perspective is slightly more general: if required, one can always adopt a convention where predicates used in the data are not allowed to occur in the rule heads.

A *substitution* σ is a partial mapping of variables to terms such that the domain of the mapping is finite. For α a term, an atom, a rule, or a set thereof, $\alpha\sigma$ is the result of replacing each occurrence of a variable x in α with $\sigma(x)$, provided that the latter is defined. If α is an atom or a rule and σ is a substitution defined on all variables of α, then $\alpha\sigma$ is a *(ground) instance* of α.

Let Π be a datalog program. A *stratification* of Π is a function λ mapping each atom occurring in Π to a positive integer that satisfies the following conditions:

- for each rule $r \in \Pi$, we have $\lambda(A) \leq \lambda(\mathsf{h}(r))$ for each atom $A \in \mathsf{b}^+(r)$ and $\lambda(A) < \lambda(\mathsf{h}(r))$ for each atom $A \in \mathsf{b}^-(r)$;
- for all atoms A_1 and A_2 occurring in Π that share a ground instance,[7] we have $\lambda(A_1) = \lambda(A_2)$.

Given such λ, each $\Pi^s := \{r \in \Pi \mid \lambda(\mathsf{h}(r)) = s\}$ is called a *stratum* of Π w.r.t. λ, and the *recursive* subset Π^s_{r} and the *nonrecursive* subset Π^s_{nr} of Π^s w.r.t. λ are defined as follows:

$$\Pi^s_{\mathsf{r}} := \{r \in \Pi^s \mid \text{atom } A \in \mathsf{b}^+(r) \text{ exists such that } \lambda(\mathsf{h}(r)) = \lambda(A)\} \tag{2}$$

$$\Pi^s_{\mathsf{nr}} := \Pi^s \setminus \Pi^s_{\mathsf{r}} \tag{3}$$

Let $\Pi_{\mathsf{r}} := \bigcup_s \Pi^s_{\mathsf{r}}$ and $\Pi_{\mathsf{nr}} := \bigcup_s \Pi^s_{\mathsf{nr}}$. Program Π is *nonrecursive* w.r.t. λ if $\Pi_{\mathsf{r}} = \emptyset$; otherwise, Π is *recursive* w.r.t. λ. For each $s \geq 2$, let Out^s be the subset of $\mathcal{F}$ containing all instances of each atom A with $\lambda(A) = s$; let $\mathsf{Out}^1 := \mathcal{F} \setminus \bigcup_{i \geq 2} \mathsf{Out}^i$; and let $\mathsf{Out}^{<s} := \bigcup_{1 \leq s' < s} \mathsf{Out}^{s'}$. The program Π is *stratifiable* if a stratification of Π exists. Each positive program is stratifiable: a function mapping all atoms in Π to the same stratum is a valid stratification. Please note Π can admit many stratifications, which are naturally ordered partially as follows: a stratification λ of Π is *more granular* than a stratification λ' of Π if, for each stratum Π^s of Π w.r.t. λ, there exists a stratum $\Pi^{s'}$ of Π w.r.t. λ' such that $\Pi^s \subseteq \Pi^{s'}$, and at least one of these inclusions is proper.

Please note that our definition of stratification generalises the common definition: usually, λ maps predicates, rather than atoms, to strata Abiteboul et al. [1], so two atoms having the same predicate are always in the same stratum. In contrast, our definition allows atoms over the same predicate to be in different strata as long as they have no common instances. This generalisation allows us to introduce stratification in RDF (where all atoms necessarily refer to a single predicate Tr), but it does not affect any of the standard results about stratification. In particular, if a program is stratified according to our definition, we can obtain a program stratified according to the usual notion: for all atoms A_1 and A_2 that have the same predicate but do not share a ground instance, we replace the predicate in either A_1 or A_2. It is straightforward to see that the resulting program is stratified according to the usual notion (i.e., where λ maps predicates to strata), and that each materialisation of the original program corresponds one-to-one to each materialisation of the transformed program. Moreover, it is straightforward to see that our condition is a special case of *modular stratification* [66].

We now define the materialisation of a stratifiable datalog program w.r.t. a dataset. Given a program Π, a rule r, and a dataset I, the set $\mathsf{inst}_r\big[I\big]$ captures the *instances* of r obtained by applying r to I, the set $r\big[I\big]$ captures the head atoms derived by these instances, and the set $\Pi\big[I\big]$ captures the head atoms derived by all the rules:

$$\mathsf{inst}_r\big[I\big] = \{r\sigma \mid \sigma \text{ is a substitution such that } \mathsf{b}^+(r)\sigma \subseteq I \text{ and } \mathsf{b}^-(r)\sigma \cap I = \emptyset\} \tag{4}$$

$$r\big[I\big] = \{\mathsf{h}(r') \mid r' \in \mathsf{inst}_r\big[I\big]\} \tag{5}$$

$$\Pi\big[I\big] = \bigcup_{r \in \Pi} r\big[I\big] \tag{6}$$

Now, let Π be a stratifiable datalog program, let λ be a stratification of Π with a maximum stratum index S, and let E be a dataset of *explicit facts*. We inductively define a sequence of datasets as follows:

- let $I^0_\infty := E$;
- for each stratum index s with $1 \leq s \leq S$,
  - let $I^s_0 := I^{s-1}_\infty$,
  - for each $i \geq 1$, let $I^s_i := I^s_{i-1} \cup \Pi^s\big[I^s_{i-1}\big]$, and
  - let $I^s_\infty := \bigcup_{i \geq 0} I^s_i$.

It is well known that I^S_∞ is independent of the choice of λ [1]; this set is called the *materialisation* of Π w.r.t. E and we denote it by $\mathsf{mat}(\Pi, E)$. Additionally, the datasets I^s_∞, for each s with $0 \leq s \leq S$ (and hence the dataset $\mathsf{mat}(\Pi, E)$ as well) are all finite [1]. The process of computing $\mathsf{mat}(\Pi, E)$ is often called *forward chaining* or (somewhat ambiguously) *materialisation*.

The definition from the previous paragraph provides us with a naïve way of computing $\mathsf{mat}(\Pi, E)$: we compute an arbitrary stratification λ and initialise $I := E$; then, for each stratum s we repeatedly compute $\Pi^s\big[I\big]$ and add the result to I as long as I changes. Such computation, however, is very inefficient because, at each iteration step, the computation $\Pi^s\big[I\big]$ repeats the derivation of all facts derived in all previous iteration steps. We can avoid this major source of inefficiency by ensuring that, whenever we evaluate a rule against the dataset I, we require that at least one body atom is matched in the dataset Δ containing the facts that were newly derived in the previous iteration. In other words, in each iteration, instead of $\Pi^s\big[I\big]$, we use $\Pi^s\big[I \mathbin{\vdots} \Delta\big]$ that is defined as in (5) and (6), but by considering $\mathsf{inst}_r\big[I \mathbin{\vdots} \Delta\big]$ defined as follows:

$$\mathsf{inst}_r\big[I \mathbin{\vdots} \Delta\big] := \{r\sigma \mid \sigma \text{ is a substitution such that } \mathsf{b}^+(r)\sigma \subseteq I, \quad \mathsf{b}^-(r)\sigma \cap I = \emptyset, \quad \text{and} \quad \mathsf{b}^+(r)\sigma \cap \Delta \neq \emptyset\}.$$

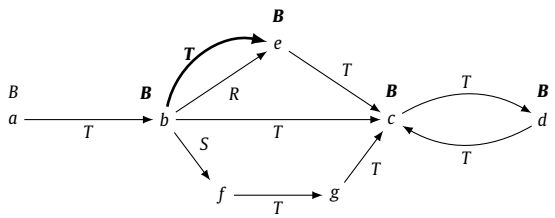
[7] This condition can be checked by letting A'_2 be a 'copy' of A_2 obtained by renaming all variables in A_2 with fresh variables and then checking whether a unifier of A_1 and A'_2 exists.

Algorithm 1 SEMINAIVE(E, Π, λ).

1: $I := \emptyset$
2: **for** each stratum index s with $1 \leq s \leq S$ **do**
3: $\quad N := (E \cap \mathsf{Out}^s) \cup \Pi^s_{\mathsf{nr}}[I]$
4: $\quad$ **loop**
5: $\quad\quad \Delta := N \setminus I$
6: $\quad\quad$ **if** $\Delta = \emptyset$ **then break**
7: $\quad\quad I := I \cup \Delta$
8: $\quad\quad N := \Pi^s_{\mathsf{r}}[I \,\vdots\, \Delta]$

Algorithm 1 uses this idea to compute the materialisation more efficiently via *seminaïve* computation. Dataset N stores the facts derived in each iteration by applying the rules, and dataset Δ stores the newly derived such facts. Since we do not distinguish extensional from intensional predicates, the set of explicit facts E can contain facts derivable in any stratum. We take this into account in line 3 by initialising N to contain $E \cap \mathsf{Out}^s$—that is, the part of E relevant to stratum s. Moreover, the rules in Π^s always derive facts from Out^s, whereas the body atoms of the nonrecursive rules in Π^s_{nr} can match only to atoms in $\mathsf{Out}^{<s}$; thus, we can evaluate $\Pi^s_{\mathsf{nr}}[I]$ only once and use the result to compute set N in line 3. If program Π^s is nonrecursive, then set N is empty in line 8, so the algorithm exits the inner loop in the next iteration; otherwise, using $\mathsf{inst}_r[I \,\vdots\, \Delta]$ in line 8 ensures that each instance of each rule in Π is considered at most once [1]. Consequently, the algorithm never considers a rule instance more than once, and it is said to have the *nonrepetition property*. Note that a fact can still be derived many times by *different* rule instances; while such derivations can be easily prevented when all rules in a program are ground (i.e., variable-free), we do not know of an evaluation technique in the literature that can prevent such derivations for general datalog programs.

We illustrate all these definitions by means of the following example.



$$E = \{\, R(b,e),\ S(b,f),\ B(a),\ T(a,b),\ T(b,c),\ T(c,d),\ T(d,c),\ T(e,c),\ T(f,g),\ T(g,c) \,\}$$

$$\mathsf{mat}(\Pi, E) = E \cup \{\, T(b,e),\ B(b),\ B(c),\ B(d),\ B(e) \,\}$$

Fig. 1. The dataset and the materialisation from Example 1. Constants correspond to vertices; explicit facts are shown in normal font, and facts introduced by materialisation are shown in bold font; unary facts are shown by labelling constants with predicates; and binary facts are shown as arcs labelled with predicates.

Example 1. Let Π be the program consisting of rules (7)–(9), and let E be the set of explicit facts shown in Fig. 1. The facts obtained by materialising Π w.r.t. E are also shown in the figure.

$$R(x,y) \wedge \text{not}\, A(x) \rightarrow T(x,y) \tag{7}$$

$$S(x,y) \wedge A(x) \rightarrow T(x,y) \tag{8}$$

$$T(x,y) \wedge B(x) \rightarrow B(y) \tag{9}$$

Predicate T occurs in both the set of explicit facts and the heads of rules (7) and (8), and so there is no separation between extensional and intensional predicates. However, we can distinguish the explicit from the derived facts; for example, $T(a,b)$ is explicit as it is given as part of the input, whereas $T(b,e)$ is derived during materialisation. If separation between extensional and intensional facts were desired, we could simply introduce a fresh predicate T_E, change each explicit fact $T(s,t)$ into $T_E(s,t)$, and add a rule $T_E(x,y) \rightarrow T(x,y)$. The program Π is stratifiable: function λ defined in (10) satisfies the conditions of stratification. Thus, $\Pi^1 = \emptyset$, $\Pi^2 = \Pi^2_{\mathsf{nr}}$ and it contains rules (7) and (8), and $\Pi^3 = \Pi^3_{\mathsf{r}}$ and it contains rule (9).

$$\lambda = \{\, A(x) \mapsto 1,\ R(x,y) \mapsto 1,\ S(x,y) \mapsto 1,\ T(x,y) \mapsto 2,\ B(x) \mapsto 3,\ B(y) \mapsto 3 \,\} \tag{10}$$

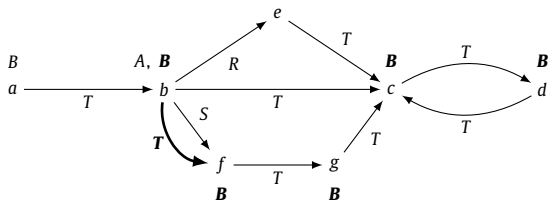
4. Problem statement

In this paper, we study the problem of efficiently updating the materialisation of a datalog program in response to changes in the explicit facts. More specifically, let Π be a datalog program and let E be a set of explicit facts, and let us assume that $I = \mathsf{mat}(\Pi, E)$ has already been computed. Furthermore, let E^- and E^+ be sets of facts to be removed from E and added to E, respectively, and let $E' = (E \setminus E^-) \cup E^+$. The problem of *materialisation maintenance* is to compute $I' = \mathsf{mat}(\Pi, E')$. We often call I and I' the *'old'* and *'new'* materialisation, respectively. A naïve solution is to recompute the 'new' materialisation 'from scratch' using Algorithm 1; however, if the size of E^- and E^+ is small compared to the

size of E, this is likely to require roughly the same work as the computation of I. Hence, in this paper we discuss various *(materialisation) maintenance algorithms* that can reuse (parts of) $\mathsf{mat}(\Pi, E)$ in order to avoid repeating most of the work and thus improve performance.

We say that a fact F is *derived* w.r.t. E by an instance $r\sigma$ of a rule $r \in \Pi$ if $r\sigma \in \mathsf{inst}_r\big[\mathsf{mat}(\Pi, E)\big]$ and $F = \mathsf{h}(r\sigma)$; we also say that $r\sigma$ *fires* w.r.t. E to derive F. Intuitively, the seminaïve algorithm computes all rule instances that fire w.r.t. E and adds to I all derived facts. The number of rule instances that fire w.r.t. E often provides us with a good measure of the performance of a reasoning algorithm: identifying such rule instances is costly as it requires join evaluation; moreover, each such rule instance derives a fact that may require duplicate elimination; and finally, the number of firing rule instances can be much larger than the number of facts in $\mathsf{mat}(\Pi, E)$. Thus, to promote efficiency, materialisation maintenance algorithms typically aim to reduce the number of rule instances that fire during the course of the algorithm's execution. As our evaluation shows, the number of rule instances considered can be a good indicator of an algorithm's performance, but other aspects (e.g., query planning or various implementation issues) can also considerably influence an algorithm's performance.

Updating E to E' changes the set of rule instances that fire—that is, some rule instances that fire w.r.t. E may not fire w.r.t. E', and vice versa. Dealing with a rule instance $r\sigma$ that fires w.r.t. E' but not w.r.t. E is comparatively easy: we simply add $\mathsf{h}(r\sigma)$ to the materialisation. In contrast, dealing with a rule instance $r\sigma$ that fires w.r.t. E but not w.r.t. E' is much more involved: $\mathsf{h}(r\sigma)$ should be deleted from I *only if* no rule instance $r'\sigma'$ exists that derives $\mathsf{h}(r\sigma)$ w.r.t. E'. In other words, to handle $r\sigma$, we may need to look at an unbounded number of other rule instances, as the following example shows.



$$E \cup E^+ = \{\, R(b,e),\ S(b,f),\ B(a),\ A(b),\ T(a,b),\ T(b,c),\ T(c,d),\ T(d,c),\ T(e,c),\ T(f,g),\ T(g,c) \,\}$$

$$\mathsf{mat}(\Pi, E \cup E^+) = E \cup E^+ \cup \{\, T(b,f),\ B(b),\ B(c),\ B(d),\ B(f),\ B(g) \,\}$$

Fig. 2. Update for Example 1.

Example 2. Let Π and E be the program and the set of explicit facts, respectively, as specified in Example 1, and let $E^- := \emptyset$ and let $E^+ := \{A(b)\}$. Fig. 2 shows the explicit facts and the materialisation after the update. We next discuss different situations that can occur when updating the materialisation.

Due to the addition of $A(b)$, the rule instances (11)–(14) now fire; however, fact $B(c)$ is already present in the materialisation, and so only $T(b, f)$, $B(f)$, and $B(g)$ should be added. As one can see, dealing with rule instances that fire due to the update is easy and it requires just adding the derived facts to the materialisation.

$$S(b, f) \wedge A(b) \rightarrow T(b, f) \tag{11}$$

$$T(b, f) \wedge B(b) \rightarrow B(f) \tag{12}$$

$$T(f, g) \wedge B(f) \rightarrow B(g) \tag{13}$$

$$T(g, c) \wedge B(g) \rightarrow B(c) \tag{14}$$

In contrast, dealing with rule instances that no longer fire is much more involved. In particular, due to the addition of $A(b)$, rule instances (15)–(17) no longer fire, suggesting that facts $T(b, e)$, $B(e)$, and $B(c)$ may not hold after the update. However, fact $B(c)$ is still derived by rule instances (18) and (19) after the update, whereas facts $T(b, e)$ and $B(e)$ have no such alternative derivations; thus, $B(c)$ should be kept in the materialisation, while $T(b, e)$ and $B(e)$ should be deleted.

$$R(b, e) \wedge \text{not}\, A(b) \rightarrow T(b, e) \tag{15}$$

$$T(b, e) \wedge B(b) \rightarrow B(e) \tag{16}$$

$$T(e, c) \wedge B(e) \rightarrow B(c) \tag{17}$$

$$T(b, c) \wedge B(b) \rightarrow B(c) \tag{18}$$

$$T(d, c) \wedge B(d) \rightarrow B(c) \tag{19}$$

This example demonstrates two important aspects of the materialisation maintenance problem. First, the main issue is to efficiently determine the changes in rule instances that fire, as well as determining when a fact derived by a rule instance that no longer fires has an alternative derivation. Second, due to negation in rule bodies, additions are not necessarily monotonic—that is, adding explicit facts may lead to deleting derived facts. As a consequence, maintenance cannot be split into a deletion step and an insertion step that process E^- and E^+ independently; rather, the two must be interleaved.

In Sections 5, 6, and 7, we present three materialisation maintenance algorithms: counting, DRed, and FBF. We shall demonstrate how each algorithm handles Example 2. As we shall see, each algorithm uses a different strategy to efficiently identify the changes in the rule instances that fire and identify alternative derivations for a given fact. In Section 8 we compare these strategies and identify patterns of input data that might be particularly suited for different algorithms, and in Section 9 we discuss certain implementation-related issues. As the size of the update increases, the cost of identifying changes will necessarily outgrow the cost of reconsidering all rule instances: in the extreme case where $E^- = E$, a maintenance algorithm will have to undo all rule instances from the initial materialisation, whereas rematerialisation 'from scratch' requires no work at all. Thus, as the size of the update increases, all three algorithms will eventually become less efficient than rematerialisation. In Section 10 we empirically investigate this phenomenon, and we also compare the performance of our algorithms on small updates.

5. The counting algorithm

In this section we discuss the counting algorithm. First, in Section 5.1 we discuss the existing approaches and show that they either do not handle recursive rules or do it inefficiently. Next, in Section 5.2 we discuss the intuition behind our extension of the counting algorithm to recursive rules, and then in Section 5.3 we present our algorithm formally and state its correctness.

5.1. Existing approaches

Gupta et al. [25] presented a materialisation maintenance algorithm for nonrecursive datalog that uses the idea of counting derivations that we outlined in Section 2. In Example 3 we discuss in detail various aspects of the algorithm, and then in Example 4 we point out an inefficiency inherent in the approach.

Example 3. Let Π be the program containing only the nonrecursive rule (20).

$$R(x, y) \wedge R(y, z) \rightarrow S(x, z) \tag{20}$$

As we have already explained, the algorithm by Gupta et al. [25] counts how many times a fact has been derived; these counts must be maintained continuously, so this algorithm does not distinguish between the 'initial materialisation' and 'materialisation maintenance'—that is, the algorithm assumes that the set of explicit facts E is initially empty, and that all changes to E are made incrementally. Hence, we start our example by initialising E and I as empty.

We next demonstrate how the algorithm handles the insertion of $R(a, b)$, $R(b, d)$, $R(a, c)$, and $R(c, d)$ into E. To formalise their algorithm, Gupta et al. [25] distinguish the state of all predicates before and after the update; in this example, R (without any annotation) and R^ν refer to the state before and after the update, respectively, and similarly for S and S^ν. Moreover, Gupta et al. [25] represent the desired changes using special Δ-predicates, in which facts can be annotated with an integer specifying the number of derivations of a fact to be added or removed, respectively. In our example, the desired changes to the predicate R are represented as the following facts over predicate ΔR, where we represent the annotations as superscripts:

$$\Delta R(a, b)^{+1} \qquad \Delta R(b, d)^{+1} \qquad \Delta R(a, c)^{+1} \qquad \Delta R(c, d)^{+1} \tag{21}$$

The algorithm processes the program by strata. Our example program Π consists of two strata: $\Pi^1 = \emptyset$ and $\Pi^2 = \Pi$. Since Π^1 is empty, the algorithm handles Π^1 by computing R^ν from R and ΔR in the obvious way, producing the following facts:

$$R^\nu(a, b)^1 \qquad R^\nu(b, d)^1 \qquad R^\nu(a, c)^1 \qquad R^\nu(c, d)^1 \tag{22}$$

In stratum Π^2, the algorithm needs to compute ΔS from R, ΔR, and R^ν. To achieve this, Gupta et al. [25] observe that S^ν is defined by rule (23), which can be rewritten as formula (24):

$$R^\nu(x, y) \wedge R^\nu(y, z) \rightarrow S^\nu(x, z) \tag{23}$$

$$R^\nu(x, y) \wedge [R(y, z) \vee \Delta R(y, z)] \rightarrow S^\nu(x, z) \tag{24}$$

The disjunction in formula (24) can be distributed over the conjunction to produce rules (25) and (26).

$$R^\nu(x, y) \wedge R(y, z) \rightarrow S^\nu(x, z) \tag{25}$$

$$R^\nu(x, y) \wedge \Delta R(y, z) \rightarrow S^\nu(x, z) \tag{26}$$

Furthermore, atom $R^\nu(x, y)$ in formula (25) can be rewritten as in formula (27), and the disjunction can again be distributed over the conjunction to produce rules (28) and (29).

$$[R(x, y) \vee \Delta R(x, y)] \wedge R(y, z) \rightarrow S^{\nu}(x, z) \tag{27}$$

$$R(x, y) \wedge R(y, z) \rightarrow S^{\nu}(x, z) \tag{28}$$

$$\Delta R(x, y) \wedge R(y, z) \rightarrow S^{\nu}(x, z) \tag{29}$$

Now rule (28) defines the 'old' state of S, so rules (26) and (29) define ΔS, which can be captured by rules (30) and (31).

$$\Delta R(x, y) \wedge R(y, z) \rightarrow \Delta S(x, z) \tag{30}$$

$$R^{\nu}(x, y) \wedge \Delta R(y, z) \rightarrow \Delta S(x, z) \tag{31}$$

Thus, the algorithm evaluates rules (30) and (31) on facts (21) and (22), but rule evaluation is modified so that, when a rule body is matched to facts $B_1^{k_1}, \ldots, B_n^{k_n}$, the instantiated rule head is derived with count $k_1 \cdots k_n$, and the results of all such derivations are added. In our example, rule (31) matches as $R^{\nu}(a, b)^1 \wedge \Delta R(b, d)^{+1} \rightarrow \Delta S(a, d)^{+1}$ and $R^{\nu}(a, c)^1 \wedge \Delta R(c, d)^{+1} \rightarrow \Delta S(a, d)^{+1}$, and adding the results of these two derivations produces (32). Finally, the algorithm computes S^{ν} as (33) by applying ΔS to S.

$$\Delta S(a, d)^{+2} \tag{32}$$

$$S^{\nu}(a, d)^2 \tag{33}$$

We next demonstrate how the algorithm uses these counts to correctly update the materialisation when $R(a, b)$ is deleted; note that this removes only one derivation of $S(a, d)$, so the fact should be kept in the materialisation. To this end, R and S are initialised to reflect the 'old' state (which were computed as R^{ν} and S^{ν} in the previous paragraph), ΔR is initialised to reflect the desired changes, and R^{ν} and S^{ν} are initialised as empty, as shown in (34).

$$R(a, b)^1 \qquad R(b, d)^1 \qquad R(a, c)^1 \qquad R(c, d)^1 \qquad \Delta R(a, b)^{-1} \qquad S(a, d)^2 \tag{34}$$

In stratum Π^1, the count of $R^{\nu}(a, b)$ is updated to zero so the fact is deleted, which leaves R^{ν} as shown in (35).

$$R^{\nu}(b, d)^1 \qquad R^{\nu}(a, c)^1 \qquad R^{\nu}(c, d)^1 \tag{35}$$

In stratum Π^2, the algorithm again evaluates rules (30) and (31) to compute ΔS and then compute S^{ν}. Specifically, the algorithm considers the rule instance $\Delta R(a, b)^{-1} \wedge R(b, d)^1 \rightarrow \Delta S(a, d)^{-1}$, which produces (36).

$$\Delta S(a, d)^{-1} \qquad S^{\nu}(a, d)^1 \tag{36}$$

The count of $S^{\nu}(a, d)$ is nonzero, so the fact is kept in the materialisation, as required.

Example 4 shows that the algorithm by Gupta et al. [25] can sometimes derive facts that are contained neither in the 'old' nor in the 'new' materialisation. In contrast, our new counting algorithm (cf. Section 5.3) does not exhibit such inefficiencies.

Example 4. Consider again the program Π from Example 3, containing the rule (20). Furthermore, let us assume that we initially insert only $R(b, c)$; then, the rule does not fire and so the materialisation after the initial insertion is as shown in (37).

$$R(b, c)^1 \tag{37}$$

Now let us assume that, in a single request, we insert $R(a, b)$ and delete $R(b, c)$. This clearly does not affect the S predicate (i.e., neither the 'old' nor the 'new' materialisation contains a fact with the S predicate), but the algorithm by Gupta et al. [25] still considers two instances of rule (20) that derive such facts. Let ΔR and the corresponding R^{ν} be as shown in (38).

$$\Delta R(a, b)^{+1} \qquad \Delta R(b, c)^{-1} \qquad R(a, b)^1 \tag{38}$$

The evaluation of rules (30) and (31) on (38) thus gives rule to rule instances (39) and (40).

$$\Delta R(a, b)^{+1} \wedge R(b, c)^1 \rightarrow \Delta S(a, c)^{+1} \tag{39}$$

$$R^{\nu}(a, b)^1 \wedge \Delta R(b, c)^{-1} \rightarrow \Delta S(a, c)^{-1} \tag{40}$$

The effects of these two derivations cancel out so the materialisation is updated correctly. However, both rule instances combine a fact occurring only in the 'old' materialisation with a fact occurring only in the 'new' materialisation, which is inefficient.

It is natural to wonder whether the algorithm by Gupta et al. [25] can be applied to recursive rules. Nicolas and Yazdanian [24] presented a closely related algorithm and suggested that it can be used with arbitrary (i.e., possibly recursive) rules. Example 5 shows that such a simple algorithm does not correctly solve the materialisation maintenance problem. In order to not introduce more notation, we present the algorithm by Nicolas and Yazdanian [24] using the notation by Gupta et al. [25].

Example 5. Let Π be the program containing rule (41).

$$R(x, y) \rightarrow R(y, x) \tag{41}$$

The main idea of Nicolas and Yazdanian [24] is to evaluate the program using a variant of the seminaïve algorithm in which counts are maintained as in the algorithm by Gupta et al. [25]. Since rule (41) is recursive, we need to distinguish ΔR before and after the rule application. To this end, we refer to the latter using predicate ΔR^{ν}, defined using rule (42).

$$\Delta R(x, y) \rightarrow \Delta R^{\nu}(y, x) \tag{42}$$

Thus, to insert a fact $R(a, b)$, we initialise ΔR, update R, and compute ΔR^{ν} using rule (42) as shown in (43).

$$\Delta R(a, b)^{+1} \qquad R(a, b)^{1} \qquad \Delta R^{\nu}(b, a)^{+1} \tag{43}$$

As in the seminaïve algorithm, we next set ΔR to be the fresh facts derived in the previous rule application (i.e., not the facts whose count was changed, but only the facts that were added or removed). The latter is needed to ensure termination: if instead we set ΔR to be the set of facts whose count changed, on our example the algorithm would keep increasing the counts indefinitely. Furthermore, we update R by extending it with ΔR^{ν}, and we clear ΔR^{ν}, producing the state shown in (44).

$$\Delta R(b, a)^{+1} \qquad R(a, b)^{1} \qquad R(b, a)^{1} \tag{44}$$

Finally, we apply the rule (42) to (44) to derive (45).

$$R(a, b)^{1} \qquad R(b, a)^{1} \qquad \Delta R^{\nu}(a, b)^{+1} \tag{45}$$

We update R using ΔR^{ν}; however, the fact $R(a, b)$ is not freshly derived (i.e., the update only changes its count), and so we set ΔR to be empty. Consequently, the algorithm terminates in the state shown in (46).

$$R(a, b)^{2} \qquad R(b, a)^{1} \tag{46}$$

Now let us assume that we wish to delete $R(a, b)$. To this end, we initialise ΔR as shown in (47). Applying rule (42) to (47) produces (48), which after updating R produces (49).

$$\Delta R(a, b)^{-1} \qquad R(a, b)^{2} \qquad R(b, a)^{1} \tag{47}$$

$$\Delta R(a, b)^{-1} \qquad R(a, b)^{2} \qquad R(b, a)^{1} \qquad \Delta R^{\nu}(a, b)^{-1} \tag{48}$$

$$R(a, b)^{1} \qquad R(b, a)^{1} \tag{49}$$

The count of fact $R(a, b)$ is decreased to one, but the fact is still present in the materialisation, so ΔR is empty; as we mentioned earlier, this is needed for termination. Hence, the algorithm terminates, but without correctly deleting $R(a, b)$ and $R(b, a)$.

Nicolas and Yazdanian [24] already noticed this problem, but without proposing a concrete solution; hence, their algorithm correctly handles only nonrecursive programs. Intuitively, this problem arises because a fact can depend on itself through arbitrarily long dependency cycles (i.e., the lengths of such cycles is determined by the rules and the data), and it is analogous to why reference counting does not provide a general garbage collection strategy for languages such as Java.

As a possible solution, Dewan et al. [9] proposed an algorithm that maintains multiple counts per fact. As we discuss in Example 6, this algorithm is based on the naïve, rather than the seminaïve algorithm. It is well known that naïve materialisation is very inefficient on large datasets, and the algorithm by Dewan et al. [9] is likely to suffer from similar issues in practice.

Example 6. Let Π be the recursive program from Example 5 consisting only of the rule (41). To establish clear links with the work by Dewan et al. [9], we use their notation. The algorithm distinguishes the initial materialisation from subsequent maintenance, and so we first show how the algorithm handles the former.

To this end, the algorithm starts with the initial set of facts $\mathcal{D}_0^0$ that, in our example, is as shown in (50). Next, the algorithm computes the *update set* US_0^0 by applying the rules of Π to $\mathcal{D}_0^0$; however, each fact in US_0^0 is annotated with a

natural number reflecting the number of derivations. In our example, the algorithm computes US_0^0 as shown in (51). Next, the algorithm propagates the changes in US_0^0 to obtain $\mathcal{D}_1^0$—that is, to compute the next set of facts in the naïve computation of the materialisation; in our example, this produces $\mathcal{D}_1^0$ as shown in (52). Next, the algorithm again applies all rules of Π to $\mathcal{D}_1^0$ to compute the update set US_1^0 shown in (53); note that this repeats all derivations done in all previous steps, which is known to be very inefficient. Next, the algorithm propagates US_1^0 to $\mathcal{D}_2^0$, as shown in (54). At this point, the algorithm notices that $\mathcal{D}_1^0$ and $\mathcal{D}_2^0$ are the same, so the algorithm terminates. Consequently, the only difference to the standard naïve materialisation algorithm is the maintenance of the counts on the facts in update sets.

$$\mathcal{D}_0^0 = \{R(a,b)\} \tag{50}$$

$$US_0^0 = \{R(b,a)^{+1}\} \tag{51}$$

$$\mathcal{D}_1^0 = \{R(a,b), R(b,a)\} \tag{52}$$

$$US_1^0 = \{R(b,a)^{+1}, R(a,b)^{+1}\} \tag{53}$$

$$\mathcal{D}_2^0 = \{R(a,b), R(b,a)\} \tag{54}$$

To maintain the materialisation, the algorithm assumes that the changes to the explicit facts are represented in a set Δ. Thus, to delete $R(a,b)$, this set is initialised as shown in (55). The algorithm next computes sets of facts $\mathcal{D}_0^1, \mathcal{D}_1^1, \ldots$ and update sets $US_0^1, US_1^1, \ldots$; intuitively, each $\mathcal{D}_i^1$ will represent the set of facts in the i-th iteration of the naïve algorithm, where the superscript 1 means 'after the first change'. On our example, this is done as follows. First, the algorithm computes $\mathcal{D}_0^1$ from $\mathcal{D}_0^0$ and Δ as shown in (56). Next, the algorithm computes US_0^1; however, instead of applying all the rules in Π 'from scratch', the algorithm computes the so-called *fix set* that can be added to US_0^0 to obtain US_0^1. This is done in two steps.

- The algorithm evaluates Π by matching all body atoms in $\mathcal{D}_0^0$ and by requiring at least one atom to be matched to $\mathcal{D}_0^0 \setminus \mathcal{D}_0^1$; this produces facts that must be *undone* in US_0^0 to obtain US_0^1.
- The algorithm evaluates Π by matching all body atoms in $\mathcal{D}_0^1$ and by requiring at least one atom to be matched to $\mathcal{D}_0^1 \setminus \mathcal{D}_0^0$; this produces facts that must be *added* to US_0^0 to obtain US_0^1.

In our example, this produces the fix set as shown in (57); this produces the update set US_0^1 as shown in (58) that, when applied to $\mathcal{D}_0^1$, produces $\mathcal{D}_1^1$ as shown in (59)

$$\Delta = \{R(a,b)^{-1}\} \tag{55}$$

$$\mathcal{D}_0^1 = \emptyset \tag{56}$$

$$\text{fix\_set}(\mathcal{D}_0^0, \mathcal{D}_0^1) = \{R(a,b)^{-1}\} \tag{57}$$

$$US_0^1 = \emptyset \tag{58}$$

$$\mathcal{D}_1^1 = \emptyset \tag{59}$$

The algorithm now repeats the process and computes the fix set from $\mathcal{D}_1^0$ and $\mathcal{D}_1^1$ as shown in (60); due to the naïve rule application strategy, this repeats all derivations made in the previous step. This produces the update set US_1^1 shown in (61), which in turn produces $\mathcal{D}_2^1$ shown in (62).

$$\text{fix\_set}(\mathcal{D}_1^0, \mathcal{D}_1^1) = \{R(a,b)^{-1}, \quad R(b,a)^{-1}\} \tag{60}$$

$$US_1^1 = \emptyset \tag{61}$$

$$\mathcal{D}_2^1 = \emptyset \tag{62}$$

At this point, the algorithm notices that no further changes can be made to either $\mathcal{D}_2^1$ or US_1^1 so the algorithm terminates, thus correctly solving the materialisation maintenance problem. Note that keeping a separate count for each update set is crucial for the algorithm's correctness: whereas the algorithm in Example 5 records that fact $R(a,b)$ has two derivations in total, the algorithm by Dewan et al. [9] splits this information over update sets US_0^0 and US_1^0.

Thus, although the algorithm by Dewan et al. [9] can be very inefficient, it demonstrates how the problems of cyclic dependencies between facts outlined in Example 5 can be solved by maintaining one count for each round of rule application.

Initial Trace							Facts	Updated Trace						
$\boldsymbol{N}_1^1$	$\boldsymbol{N}_1^2$	$\boldsymbol{N}_1^3$	$\boldsymbol{N}_2^3$	$\boldsymbol{N}_3^3$	$\boldsymbol{N}_4^3$	$\boldsymbol{N}_5^3$		$\boldsymbol{N}_1^1$	$\boldsymbol{N}_1^2$	$\boldsymbol{N}_1^3$	$\boldsymbol{N}_2^3$	$\boldsymbol{N}_3^3$	$\boldsymbol{N}_4^3$	$\boldsymbol{N}_5^3$
							$A(b)$ ▷	1*						
1							◁ $R(b,e)$ ▷	1						
1							◁ $S(b,f)$ ▷	1						
	1						◁ $T(a,b)$ ▷		1					
	1						◁ $T(b,c)$ ▷		1					
	1						◁ $T(c,d)$ ▷		1					
	1						◁ $T(e,c)$ ▷		1					
	1						◁ $T(f,g)$ ▷		1					
	1						◁ $T(g,c)$ ▷		1					
	1*						◁ $T(b,e)$							
							$T(b,f)$ ▷		1*					
		1					◁ $B(a)$ ▷			1				
			1				◁ $B(b)$ ▷				1			
				1	1*	1*	◁ $B(c)$ ▷					1		2*
				1*			◁ $B(e)$							
							$B(f)$ ▷					1*		
					1		◁ $B(d)$ ▷						1	
							$B(g)$ ▷						1*	

Fig. 3. The traces for Examples 7 and 8. Each $\boldsymbol{N}_i^s$ is represented as a column of multiplicities of the corresponding facts. For example, fact $B(c)$ occurs with multiplicity one in $\boldsymbol{N}_3^3$, $\boldsymbol{N}_4^3$, and $\boldsymbol{N}_5^3$ of the initial trace, and with multiplicity one in $\boldsymbol{N}_3^3$ and multiplicity two in $\boldsymbol{N}_5^3$ of the updated trace. For easier reference, all facts belonging to the initial (resp. updated) trace are marked with ◁ (resp. ▷). Multiplicities that differ between the two traces are marked by an asterisk.

5.2. Intuition behind our recursive counting algorithm

We now present the intuition behind of our counting-based algorithm for materialisation maintenance, which overcomes the shortcomings of the algorithms outlined in Section 5.1. In particular, unlike the algorithms by Gupta et al. [25] and Nicolas and Yazdanian [24], we correctly handle recursive datalog with stratified negation; unlike the algorithm by Gupta et al. [25], we only consider rule instances that fire w.r.t. either the old or the new set of explicit facts; and, unlike the algorithm in Dewan et al. [9], our approach is based on the seminaïve evaluation.

To use our algorithm, the initial materialisation must be computed using a variant of the standard seminaïve algorithm: the modified algorithm must remember all sets N computed in lines 3 and 8 of Algorithm 1, and, to count the derivations, these must be *multisets*, rather than sets. Thus, for each stratum index s, multiset $\boldsymbol{N}_1^s$ contains the facts computed in line 3, and, for each iteration $i \geq 2$ of the loop in lines 4–8, multiset $\boldsymbol{N}_i^s$ contains the facts derived in line 8. We call the collection of all of these sets a *trace* of the seminaïve algorithm. The following example demonstrates how the trace is computed.

Example 7. Let Π, E, and λ be the program, the dataset, and the stratification, respectively, from Example 1. Fig. 1 shows the materialisation of Π w.r.t. E, and the left-hand side of Fig. 3 shows the initial trace obtained by the modified seminaïve algorithm.[8] The trace is computed by iteratively considering all strata. For $s = 1$, multiset $\boldsymbol{N}_1^1$ is initialised to $E \cap \mathsf{Out}^1$ and, since Π^1 is empty, no further derivations are made. For $s = 2$, multiset $\boldsymbol{N}_1^2$ is initialised to the union of $E \cap \mathsf{Out}^1$ and the result of applying Π_{nr}^2 to the facts derived in the previous stratum. Finally, for $s = 3$, since Π_{nr}^3 is empty, the multiset $\boldsymbol{N}_1^3$ consists only of the facts in $E \cap \mathsf{Out}^3$, and multisets $\boldsymbol{N}_2^3$–$\boldsymbol{N}_5^3$ are obtained by iteratively applying Π_{r}^2. Note that the fact $B(c)$ is derived in three different iterations by the rule instances (17)–(19).

Now let E_o be the 'old' set of explicit facts, and let $E_n := (E_o \setminus E^-) \cup E^+$ be the 'new' set. Our counting-based materialisation maintenance algorithm updates the 'old' trace for E_o to the 'new' trace for E_n—that is, it updates each multiset $\boldsymbol{N}_i^s$ by undoing the rule instances that no longer hold and applying the newly applicable rule instances. To understand how this is achieved, imagine a modification of Algorithm 1 that computes in parallel the 'old' and the 'new' materialisation; thus, line 3 is modified to

$$N_o := (E_o \cap \mathsf{Out}^s) \cup \Pi_{\mathsf{nr}}^s \llbracket I_o \rrbracket, \qquad N_n := (E_n \cap \mathsf{Out}^s) \cup \Pi_{\mathsf{nr}}^s \llbracket I_n \rrbracket,$$

line 5 is modified to

$$\Delta_o := N_o \setminus I_o, \qquad \Delta_n := N_n \setminus I_n,$$

and so on. Our maintenance algorithm computes sets I_o and I_n and thus reconstructs the 'old' and the 'new' materialisation in the same way as the modified Algorithm 1. However, instead of computing $\mathsf{inst}_r[I_o]$ and $\mathsf{inst}_r[I_n]$, and $\mathsf{inst}_r[I_o \vdots \Delta_o]$ and

[8] The right-hand side of Fig. 3 also shows the trace after the update in Example 8, and arrows ◁ and ▷ next to each fact indicate whether the fact occurs in the 'old'/'new' materialisation.

$\mathsf{inst}_r\big[I_n \vdots \Delta_n\big]$ in lines 3 and 8 'from scratch', our algorithm uses I_o, I_n, Δ_o, and Δ_n to efficiently compute the *symmetric difference* between the corresponding sets. This difference contains rule instances that no longer fire, whose head atoms are removed from $\boldsymbol{N}_i^s$ to obtain the 'new' trace, and rule instances that fire only after the update, whose head atoms are added to $\boldsymbol{N}_i^s$.

Example 8. Let Π, λ, and E be as in Example 1, and let E^- and E^+ be as in Example 2. The materialisation of Π w.r.t. $(E \setminus E^-) \cap E^+$ is shown in Fig. 2, and the 'old' trace is shown in the left of Fig. 3; the 'new' trace is shown in the right of Fig. 3 and it is computed analogously to Example 7. We next discuss how our counting-based materialisation maintenance algorithm computes I_o and I_n 'in parallel', but without considering all rule instances from scratch.

As in line 1 of Algorithm 1, our algorithm initialises I_o and I_n as the empty sets. Next, it considers each stratum in turn. For $s = 1$, the algorithm adds $A(b)$ to $\boldsymbol{N}_1^1$ and, since $\Pi^1 = \emptyset$, no rule instances are considered in this stratum.

Now consider $s = 2$. Set N_o in line 3 of Algorithm 1 is given by $\boldsymbol{N}_1^2$—that is, we do not need to apply Π_{nr}^2 to I_o to reconstruct N_o. The algorithm next updates $\boldsymbol{N}_1^2$: it removes $E^- \cap \mathsf{Out}^1$ and the facts from $\Pi_{\mathsf{nr}}^2 [\![I_o]\!]$ that no longer hold, and it adds $E^+ \cap \mathsf{Out}^1$ and the facts from $\Pi_{\mathsf{nr}}^2 [\![I_n]\!]$ that previously did not hold. Instead of recomputing $\Pi_{\mathsf{nr}}^2 [\![I_o]\!]$ and $\Pi_{\mathsf{nr}}^2 [\![I_n]\!]$ from scratch, our algorithm efficiently computes just the symmetric difference between these sets. Consider an arbitrary rule $r \in \Pi_{\mathsf{nr}}^2$. The instances of r that fire in the 'old' but not the 'new' trace are the rule instances in $\mathsf{inst}_r\big[I_o\big]$ that either have a positive body atom outside I_n, or a negative one inside I_n. Conversely, instances of r that fire in the 'new', but not in the 'old', trace are the rule instances in $\mathsf{inst}_r\big[I_n\big]$ that either have a positive body atom outside I_o, or a negative one inside I_o. In our example, we have $A(b) \in I_n \setminus I_o$; thus, rule instance (11) did not fire previously so the multiplicity of $T(b, f)$ is increased to 1; and rule instance (15) no longer fires and so the multiplicity of $T(b, e)$ is decreased to 0. Since $\Pi_{\mathsf{r}}^2 = \emptyset$, the algorithm moves on to the next stratum.

Now consider $s = 3$ and iteration $i = 1$. The algorithm does not change $\boldsymbol{N}_1^3$ since there are no relevant nonrecursive rules or explicit facts. The algorithm next computes Δ_o and Δ_n as in line 5 of Algorithm 1. In line 6 both Δ_o and Δ_n contain $B(a)$ and are thus not empty, so the computation continues.

For $s = 3$ and $i = 2$, the algorithm updates I_o and I_n as in line 7 and then computes sets N_o and N_n as in line 8; again, N_o is given by $\boldsymbol{N}_2^3$. Now let r be the only rule (9) from program Π_{r}^3. The instances of r that fire in the 'old' and the 'new' evaluation are given by $\mathsf{inst}_r\big[I_o \vdots \Delta_o\big]$ and $\mathsf{inst}_r\big[I_n \vdots \Delta_n\big]$, respectively. But then, each rule instance r' of r that fires in the 'old', but not the 'new' trace satisfies $r' \in \mathsf{inst}_r\big[I_o \vdots \Delta_o\big]$ and, additionally, either

(a) r' contains a positive body atom in $I_o \setminus I_n$ or a negative one in $I_n \setminus I_o$ (and so $r' \notin \mathsf{inst}_r\big[I_n \vdots \Delta_n\big]$), or
(b) all body atoms of r' are contained in $I_n \setminus \Delta_n$ (and so again $r' \notin \mathsf{inst}_r\big[I_n \vdots \Delta_n\big]$.

The rule instances of r that fire in the 'new', but not the 'old' trace can be determined analogously. In our example, before line 8, set $I_o \setminus I_n$ contains $T(b, e)$, and set $I_n \setminus I_o$ contains $T(b, f)$ and $A(b)$; hence, the only rule instance (63) that fires in the 'old' trace also fires in the 'new' trace, and no additional rule instances fire in the 'new' trace; consequently, the algorithm leaves $\boldsymbol{N}_2^3$ unchanged and updates Δ_o and Δ_n to contain $B(b)$.

For $s = 3$ and $i = 3$, rule instance (15) satisfies condition (a) since $T(b, f) \in I_o \setminus I_n$, so the algorithm reduces the multiplicity of $B(e)$ to 0. Analogously, rule instance (64) fires only in the new trace, so the algorithm increases the multiplicity of $B(f)$ to 1.

$$T(a, b) \wedge B(a) \rightarrow B(b) \tag{63}$$

$$T(b, f) \wedge B(b) \rightarrow B(f) \tag{64}$$

For $s = 3$ and $i = 4$, the algorithm computes Δ_o and Δ_n as $\Delta_o = \{B(c), B(e)\}$ and $\Delta_n = \{B(c), B(f)\}$; it updates I_o and I_n as in line 7; and it computes N_o and N_n from line 8. As in the previous paragraph, the algorithm determines that rule instance (17) fires only in the 'old' trace, and that rule instance (13) fires only in the 'new' trace; hence, it updates $\boldsymbol{N}_4^3$ by decreasing the multiplicity of $B(c)$ to 0 and increasing the multiplicity of $B(g)$ to 1.

For $s = 3$ and $i = 5$, the algorithm determines in a similar way that the rule instance (14) fires only in the 'new' trace, and hence increases in $\boldsymbol{N}_5^3$ the multiplicity of $B(c)$ to 2.

Finally, for $s = 3$ and $i = 6$, the algorithm updates both Δ_o and Δ_n to the empty set; this ensures that the trace has been correctly updated to the 'new' trace, so the algorithm terminates.

We next summarise several important aspects of our algorithm. First, the 'old' sets I_o, Δ_o, and N_o can be determined directly using the 'old' trace and they do not require rule evaluation. Furthermore, the 'new' I_n, Δ_n, and N_n can be determined by considering only the rule instances that fire in the 'old' but not the 'new' trace and vice versa. A rule instance can be undone at most once and applied at most once, and so our algorithm enjoys a form of the *nonrepetition property*.

Second, assume that program Π satisfies $\mathsf{mat}(\Pi, \emptyset) = \emptyset$; for example, this is the case if each rule in Π contains at least one positive body atom. Then, we can compute the initial materialisation by applying the maintenance algorithm where E^+

contains all explicit facts and $E^- = \emptyset$. In other words, for such Π, we do not need a separate 'initial materialisation' algorithm; moreover, due to the nonrepetition property, the maintenance algorithm is as efficient as the 'initial materialisation' algorithm.

Third, if program Π is not recursive, Algorithm 1 never derives any facts in line 8, and so the trace contains just one set $\boldsymbol{N}_1^s$ per stratum index s; in other words, our algorithm maintains just one counter per fact. Our algorithm thus becomes an optimised version of the single-count algorithm by Gupta et al. [25] that, due to the nonrepetition property, does not suffer from the drawbacks we outlined in Example 4.

We finish this section by demonstrating how our counting-based algorithm handles the problems arising from recursive dependencies between facts that we outlined in Example 5.

	$\boldsymbol{N}_1^1$	$\boldsymbol{N}_2^1$	$\boldsymbol{N}_3^1$
$R(a,b)$	1		1
$R(b,a)$		1	

Fig. 4. The trace for Example 9.

Example 9. Analogously to Example 5, let Π contain (41), and let $E = \{R(a,b)\}$. The trace of the seminaïve algorithm is shown in Fig. 4. The two counters for $R(a,b)$ show that this fact is derived twice: once from E, and once from $R(b,a)$. This allows our algorithm to decrement all counters to zero when $R(a,b)$ is deleted from E and thus remove $R(a,b)$ from the materialisation.

5.3. Formalisation

We now formalise our counting-based approach to materialisation maintenance. Before presenting the algorithm in detail, we first fix some notation.

First, we use the standard generalisation of sets to *multisets*, where each element is associated with a positive integer called the *multiplicity* specifying the number of the element's occurrences. We typeset multiset names in bold font. For $\boldsymbol{N}$ a multiset, N is the set containing the same elements as $\boldsymbol{N}$, and $\boldsymbol{N}(x)$ is the multiplicity of each element $x \in N$. Furthermore, $\oplus$ and $\ominus$ are the standard multiset addition and subtraction operators, respectively: for $\boldsymbol{N_1}$ and $\boldsymbol{N_2}$ multisets, multiset $\boldsymbol{N} = \boldsymbol{N_1} \oplus \boldsymbol{N_2}$ is defined so that $N = N_1 \cup N_2$ and $\boldsymbol{N}(x) = \boldsymbol{N_1}(x) + \boldsymbol{N_2}(x)$ for each $x \in N$, and multiset $\boldsymbol{N'} = \boldsymbol{N_1} \ominus \boldsymbol{N_2}$ is defined so that $N' = N'_1 \cup N'_2$ where $N'_1 = N_1 \setminus N_2$ and $N'_2 = \{x \in N_1 \cap N_2 \mid \boldsymbol{N_1}(x) > \boldsymbol{N_2}(x)\}$, and

$$\boldsymbol{N'}(x) = \begin{cases} \boldsymbol{N_1}(x) & \text{if } x \in N'_1, \text{ and} \\ \boldsymbol{N_1}(x) - \boldsymbol{N_2}(x) & \text{if } x \in N'_2. \end{cases}$$

When used in combination with $\oplus$ and $\ominus$, sets are treated as multisets in which all elements occur once. For example, $N_1 \oplus N_2$ is equivalent to $\boldsymbol{N_1} \oplus \boldsymbol{N_2}$ where $\boldsymbol{N_1}$ and $\boldsymbol{N_2}$ are multisets containing each element of N_1 and N_2, respectively, with multiplicity one. We write multisets by listing their elements inside $\{\!\{\ldots\}\!\}$. Note that multiplicities are always positive; thus, for $\boldsymbol{N_1} = \{\!\{\alpha, \alpha, \alpha, \beta\}\!\}$, $\boldsymbol{N_2} = \{\!\{\alpha, \beta, \gamma\}\!\}$ and $\boldsymbol{N} = \boldsymbol{N_1} \ominus \boldsymbol{N_2}$, we have $\boldsymbol{N} = \{\!\{\alpha, \alpha\}\!\}$ and $N = \{\alpha\}$—that is, $\boldsymbol{N}$ and N do not contain β or γ.

Second, we extend notation (4) from Section 3. In particular, let I^+ and I^- be datasets, let $k \geq 0$ be a nonnegative integer, and let $P_1, \ldots, P_k$ and $N_1, \ldots, N_k$ be datasets such that, for each $1 \leq i \leq k$, we have $P_i \subseteq I^+$ and $N_i \cap I^- = \emptyset$. Then, we define

$$\begin{aligned} \mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big] = \{r\sigma \mid \text{substitution } \sigma \text{ satisfies } \mathsf{b}^+(r)\sigma \subseteq I^+ \text{ and } \mathsf{b}^-(r)\sigma \cap I^- = \emptyset, \text{ and} \\ \mathsf{b}^+(r)\sigma \cap P_i \neq \emptyset \text{ or } \mathsf{b}^-(r)\sigma \cap N_i \neq \emptyset \text{ holds for each } 1 \leq i \leq k\}. \end{aligned} \tag{65}$$

Moreover, given a program Π and a rule r, we define $r\big[\!\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big]\!\big]$ and $\Pi\big[\!\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big]\!\big]$ as follows, where multiset (66) contains each distinct occurrence of $\mathsf{h}(r')$.

$$r\big[\!\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big]\!\big] = \{\!\{\mathsf{h}(r') \mid r' \in \mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big]\}\!\} \tag{66}$$

$$\Pi\big[\!\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big]\!\big] = \bigoplus_{r \in \Pi} r\big[\!\big[I^+, I^- \vdots P_1, N_1 \vdots \ldots \vdots P_k, N_k\big]\!\big] \tag{67}$$

We shall use $\mathsf{inst}_r\big[I^+, I^-\big]$, $\mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1\big]$, and $\mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1 \vdots P_2, N_2\big]$ only—that is, k will be at most two. In all of these, for readability we omit I^- if it is equal to I^+, and we omit N_i if $N_i = \emptyset$ for $1 \leq i \leq k$. We discuss in Section 9 how one can efficiently compute these sets in practice.

Next, we present the counting version of the seminaïve algorithm that computes the initial trace. As we have already mentioned in Section 5.2, this algorithm is needed only if $\mathsf{mat}(\Pi, \emptyset) \neq \emptyset$. The pseudo-code is given in Algorithm 2. The

algorithm takes as input a program Π, a stratification λ of Π where S is the largest stratum index, and a dataset E. The only difference to Algorithm 2 is that lines 12 and 18 compute multisets rather than sets, and that these multisets are retained; the sequence $\overline{\boldsymbol{N}} = ((\boldsymbol{N}_i^s)_{i \geq 1})_{s=1}^S$ of these multisets computed by the algorithm is the *trace* of Π w.r.t. E and λ. Unlike the materialisation of Π w.r.t. E, the trace depends on the stratification λ. Moreover, each trace satisfies the following theorem.

Theorem 10. *Let Π be a datalog program, let λ be a stratification of Π with maximum stratum index S, let E be a dataset, and let $\overline{\boldsymbol{N}}$ be the trace of Π w.r.t. E and λ as computed by Algorithm 2. Then,* $\mathsf{mat}(\Pi, E) = \bigcup_{s=1}^S \bigcup_{i \geq 1} N_i^s$.

Algorithm 2 SEMINAIVECOUNTING(E, Π, λ).

```
 9: I := ∅
10: for each stratum index s with 1 ≤ s ≤ S do
11:     i := 1
12:     N_i^s := (E ∩ Out^s) ⊕ Π_nr^s [[I]]
13:     loop
14:         Δ := N_i^s \ I
15:         if Δ = ∅ then break
16:         I := I ∪ Δ
17:         i := i + 1
18:         N_i^s := Π_r^s [[I ⋮ Δ]]
```

As we discussed in Section 5.3, our maintenance algorithm emulates running Algorithm 2 in parallel on the 'old' and the 'new' explicit facts. To this end, the maintenance algorithm maintains sets I_o and Δ_o that correspond to sets I and Δ from Algorithm 2 in the 'old' run, and sets I_n and Δ_n that reflect the 'new' run. As we have explained earlier, our algorithm uses these sets to efficiently compute the symmetric difference between the rule instances that fire in the 'old' and the 'new' trace. Lemma 11 shows how to achieve this efficiently. Equation (68) states that a rule instance from $\mathsf{inst}_r\big[I_1\big]$ is not in $\mathsf{inst}_r\big[I_2\big]$ if it has a positive atom outside I_2 or a negative one inside I_2. By substituting $I_1 = I_o$ and $I_2 = I_n$ we obtain the rule instances that fire in the 'old', but not in the 'new' materialisation, and for $I_1 = I_n$ and $I_2 = I_o$ we obtain the converse; hence, we can use the righthand side of (68) to update $\boldsymbol{N}_1^s$ computed in line 12 of Algorithm 2. Furthermore, equation (69) states that a rule instance from $\mathsf{inst}_r\big[I_1 \vdots \Delta_1\big]$ is not in $\mathsf{inst}_r\big[I_2 \vdots \Delta_2\big]$ if either it has a positive atom outside I_2 or a negative one inside I_2, or it has no positive atom in Δ_2. In addition, equation (70) states that the two alternatives in (69) are mutually exclusive, and so we can use the expressions from the righthand side of (69) independently to update the multisets $\boldsymbol{N}_i^s$, $i \geq 2$, computed in line 18 of Algorithm 2.

Lemma 11. *For each rule r and all pairs of datasets (I_1, Δ_1) and (I_2, Δ_2), the following equalities hold.*

$$\mathsf{inst}_r\big[I_1\big] \setminus \mathsf{inst}_r\big[I_2\big] = \mathsf{inst}_r\big[I_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \tag{68}$$

$$\mathsf{inst}_r\big[I_1 \vdots \Delta_1\big] \setminus \mathsf{inst}_r\big[I_2 \vdots \Delta_2\big] = \mathsf{inst}_r\big[I_1 \vdots \Delta_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \cup \mathsf{inst}_r\big[(I_1 \cap I_2) \setminus \Delta_2, I_1 \cup I_2 \vdots (\Delta_1 \cap I_2) \setminus \Delta_2\big] \tag{69}$$

$$\mathsf{inst}_r\big[I_1 \vdots \Delta_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \cap \mathsf{inst}_r\big[(I_1 \cap I_2) \setminus \Delta_2, I_1 \cup I_2 \vdots (\Delta_1 \cap I_2) \setminus \Delta_2\big] = \emptyset \tag{70}$$

We are now ready to discuss our counting-based materialisation maintenance algorithm, which is shown in Algorithm 3. The algorithm takes as input an explicit dataset E, a program Π, a stratification λ of Π with maximum stratum index S, the trace of Π w.r.t. E and λ, and datasets E^- and E^+ to remove from and add to E, respectively. In addition to sets I_o, Δ_o, I_n, and Δ_n, our algorithm also maintains I_{on} as a shorthand for $I_o \setminus I_n$, and I_{no} as a shorthand for $I_n \setminus I_o$: sets I_o and I_n are likely to be large, but $I_o \setminus I_n$ and $I_n \setminus I_o$ are likely to be small on small updates; thus, instead of computing $I_o \setminus I_n$ and $I_n \setminus I_o$ each time 'from scratch', our algorithm maintains these difference sets during the course of its execution.

Algorithm 3 follows the general structure of Algorithm 2. The algorithm first initialises sets I_o, I_n, I_{on}, and I_{no} as empty in line 19. The algorithm also ensures that E^- is a subset of E (so we do not try to delete facts that are not in E) and that E^+ is disjoint with $E \setminus E^-$ (so we do not try to insert facts that are already in E); this is important because each addition or removal of a fact from E is counted in $\boldsymbol{N}_1^s$. Finally, the algorithm updates E. Next, in lines 20–38 the algorithm examines each stratum index s and reconstructs the 'old' and the 'new' derivations. Lines 22–25 correspond to line 12 in Algorithm 2: multiset $\boldsymbol{N}_1^s$ contains the 'old' state in line 22; moreover, line 23 updates $\boldsymbol{N}_1^s$ by removing the 'explicit' derivations $E^- \cap \mathsf{Out}^s$ and by removing the implicit derivations $\Pi_{\mathsf{nr}}^s \big[\!\big[I_o \vdots I_{on} \big]\!\big]$ where the latter are computed using (68); finally, line 24 updates $\boldsymbol{N}_1^s$ with the 'new' derivations in a similar way, and so multiset $\boldsymbol{N}_1^s$ contains the 'new' state in line 25—that is, $\boldsymbol{N}_1^s$ is equal to $\Pi_{\mathsf{nr}}^s \big[\!\big[I_n \big]\!\big]$ at this point. Lines 26–38 simulate lines 13–18 of Algorithm 2. Sets Δ_o and Δ_n are computed in line 27 analogously to line 14 of Algorithm 2. If both Δ_o and Δ_n are empty, then both the 'old' and the 'new' materialisation terminate, so the stratum finishes in line 28. Otherwise, the algorithm computes I_o and I_n in line 29 analogously to line 16 of Algorithm 2. Next, the algorithm computes I_{on} and I_{no} in line 30, but in a way that does not require considering the entire sets I_o and I_n. If the condition in line 31 is satisfied at this point, then both the 'old' and the

Algorithm 3 MAINTENANCECOUNTING$(E, \Pi, \lambda, \overline{\boldsymbol{N}}, E^-, E^+)$.

19: $I_o := I_n := I_{on} := I_{no} := \emptyset, \quad E^- := (E^- \cap E) \setminus E^+, \quad E^+ := E^+ \setminus E, \quad E := (E \setminus E^-) \cup E^+$
20: **for** each stratum index s with $1 \leq s \leq n$ **do**
21: $\quad i := 1$
22: $\quad N_o := N_i^s$
23: $\quad \boldsymbol{N}_i^s := \boldsymbol{N}_i^s \ominus \big((E^- \cap \mathsf{Out}^s) \oplus \Pi_{\mathsf{nr}}^s \llbracket I_o \mathrel{:} I_{on}, I_{no} \rrbracket\big)$
24: $\quad \boldsymbol{N}_i^s := \boldsymbol{N}_i^s \oplus \big((E^+ \cap \mathsf{Out}^s) \oplus \Pi_{\mathsf{nr}}^s \llbracket I_n \mathrel{:} I_{no}, I_{on} \rrbracket\big)$
25: $\quad N_n := N_i^s$
26: $\quad$ **loop**
27: $\quad\quad \Delta_o := N_o \setminus I_o, \qquad \Delta_n := N_n \setminus I_n$
28: $\quad\quad$ **if** $\Delta_o = \Delta_n = \emptyset$ **then break**
29: $\quad\quad I_o := I_o \cup \Delta_o, \qquad I_n := I_n \cup \Delta_n$
30: $\quad\quad I_{on} := I_{on} \setminus \Delta_n \cup \Delta_o \setminus I_n, \qquad I_{no} := I_{no} \setminus \Delta_o \cup \Delta_n \setminus I_o$
31: $\quad\quad$ **if** $\Delta_o = \Delta_n$ and $I_{on} = I_{no} = \emptyset$ **then**
32: $\quad\quad\quad I_o := I_n := I_o \cup \bigcup_{j>i} N_j^s$
33: $\quad\quad\quad$ **break**
34: $\quad\quad i := i + 1$
35: $\quad\quad N_o := N_i^s$
36: $\quad\quad \boldsymbol{N}_i^s := \boldsymbol{N}_i^s \ominus \big(\Pi_{\mathsf{r}}^s \llbracket I_o \mathrel{:} \Delta_o \mathrel{:} I_{on}, I_{no} \rrbracket \oplus \Pi_{\mathsf{r}}^s \llbracket (I_o \cap I_n) \setminus \Delta_n, I_o \cup I_n \mathrel{:} (\Delta_o \cap I_n) \setminus \Delta_n \rrbracket\big)$
37: $\quad\quad \boldsymbol{N}_i^s := \boldsymbol{N}_i^s \oplus \big(\Pi_{\mathsf{r}}^s \llbracket I_n \mathrel{:} \Delta_n \mathrel{:} I_{no}, I_{on} \rrbracket \oplus \Pi_{\mathsf{r}}^s \llbracket (I_n \cap I_o) \setminus \Delta_o, I_n \cup I_o \mathrel{:} (\Delta_n \cap I_o) \setminus \Delta_o \rrbracket\big)$
38: $\quad\quad N_n := N_i^s$

'new' materialisation coincide on the remaining iterations in stratum index s; therefore, the algorithm reconstructs I_o and in I_n from the trace in line 32 and proceeds to the next stratum. Finally, lines 35–38 correspond to line 18 of Algorithm 2; the symmetric difference between the rule instances in lines 36 and 37 is computed using equation (69).

The following theorem, proved in Appendix A, captures the formal properties of Algorithm 3. In particular, property (a) says that the algorithm correctly updates the 'old' to the 'new' trace, and properties (b) and (c) say that, in the worst case, the algorithm considers all rule instances from the 'old' and the 'new' materialisation once. Note that, if $E^- = E$ and $E^+ \cap E^- = \emptyset$ and the rules do not contain any constants, then the algorithm considers all of these rule instances: the instances from (b) are used to delete the 'old' materialisation, and the instances from (c) are used to compute the 'new' materialisation.

Theorem 12. *Let Π be a program, let λ be a stratification of Π with maximum stratum index S, let E be a dataset, let $\overline{\boldsymbol{N}}$ be the trace of Π w.r.t. E and λ, and let E^- and E^+ be datasets. When applied to this input, Algorithm 3 terminates and*

(a) *$\overline{\boldsymbol{N}}$ contains upon termination the trace of Π w.r.t. $(E \setminus E^-) \cup E^+$ and λ;*
(b) *lines 23 and 36 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[\mathsf{mat}(\Pi, E)\big]$ without repetition; and*
(c) *lines 24 and 37 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[\mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)\big]$ without repetition.*

6. The delete/rederive algorithm

An important drawback of the counting algorithm from Section 5 is increased memory use: facts are associated with counters whose count is, in the worst case, determined by the total number of rule instances fired by the seminaïve algorithm. Since this can be a considerable source of overhead in practice, approaches to materialisation maintenance have been developed that do not require any additional bookkeeping and that typically proceed in three steps: in the *overdeletion* step they delete each fact derivable in zero or more steps from a fact being deleted; in the *rederivation* step they rederive each fact that remains derivable after overdeletion; and in the *insertion* step they compute the consequences of the facts being inserted. Based on these ideas, Ceri and Widom [17] presented an approach to view maintenance where view definitions essentially correspond to nonrecursive datalog. Gupta et al. [25] developed the Delete/Rederive (DRed) algorithm that applies this idea to general (i.e., recursive) datalog. Although the authors claim that their approach can be extended to programs with negation, their formalisation handles only positive datalog programs and, as we shall see later in this section, the extension to programs with negation is not straightforward. The algorithm was formalised using a mixture of declarative and procedural steps, where the latter were not always specified precisely. Staudt and Jarke [28] presented another, fully declarative formalisation of DRed, where the updates to the materialisation are computed by evaluating a *maintenance* datalog program. Such a presentation style allows us to discuss various properties of the algorithm precisely, so in Section 6.1 we present an overview of the DRed algorithm using the formalisation by Staudt and Jarke [28]. Furthermore, we point out several sources of inefficiency whereby the algorithm repeatedly makes the same (redundant) inferences. Then, in Section 6.2 we present an optimised version of DRed that does not suffer from such redundancy and that extends to programs with negation. As we shall see, our optimisations interact in ways that make ensuring correctness nontrivial.

6.1. Overview

We next apply DRed to our running example introduced in Sections 3 and 4. The formalisations of DRed by Gupta et al. [25] and Staudt and Jarke [28] do not handle rules with negation, so we slightly modify our example to remove negation.

Example 13. Let Π be the program containing rules (71)–(75). Rules (71)–(73) are as in Example 1, but not $A(x)$ in rule (7) is replaced in rule (71) with $A_n(x)$. Most DRed variants require explicit and implicit facts to use distinct predicates; since predicates T and B occur in Example 1 both in rule heads and in the explicit facts, we replace predicates T and B in the explicit facts with fresh predicates T_e and B_e, respectively, and we add rules (74) and (75) that 'copy' the explicit into the implicit facts. Let E be as in Fig. 1 (with predicates T and B renamed as explained) extended with fact $A_n(b)$, and let $E^- = \{A_n(b)\}$ and $E^+ = \{A(b)\}$. By reading A_n as 'not A', one can see that the effects of the update are as in Example 2; thus, Figs. 1 and 2 show the state of the materialisation before and after the update (apart from the facts involving A_n, B_e, and T_e).

$$R(x, y) \wedge A_n(x) \rightarrow T(x, y) \tag{71}$$

$$S(x, y) \wedge A(x) \rightarrow T(x, y) \tag{72}$$

$$T(x, y) \wedge B(x) \rightarrow B(y) \tag{73}$$

$$T_e(x, y) \rightarrow T(x, y) \tag{74}$$

$$B_e(x) \rightarrow B(x) \tag{75}$$

We next discuss the intuition behind all variants of the DRed algorithm [25,28]. The algorithm starts with the overdeletion step. Fact $A_n(b)$ is used in rule instance (76) to derive $T(b, e)$ so $T(b, e)$ is deleted. Overdeletion is propagated iteratively as long as possible: $T(b, e)$ is used in rule instance (77) and so $B(e)$ is deleted; $B(e)$ is used in rule instance (78) and so $B(c)$ is deleted; $B(c)$ is used in rule instance (79) and so $B(d)$ is deleted. Finally, $B(d)$ is used in rule instance (80) to derive $B(c)$, but $B(c)$ has already been deleted, and so overdeletion stops.

$$R(b, e) \wedge A_n(b) \rightarrow T(b, e) \tag{76}$$

$$T(b, e) \wedge B(b) \rightarrow B(e) \tag{77}$$

$$T(e, c) \wedge B(e) \rightarrow B(c) \tag{78}$$

$$T(c, d) \wedge B(c) \rightarrow B(d) \tag{79}$$

$$T(d, c) \wedge B(d) \rightarrow B(c) \tag{80}$$

The algorithm next proceeds to the rederivation step, in which it tries to rederive the overdeleted facts using the 'surviving' facts. Specifically, rule instance (81) still fires after overdeletion, and so $B(c)$ is rederived—that is, it is put back into the materialisation. This, in turn, ensures that rule instance (79) fires as well, so $B(d)$ is rederived too. Finally, although this ensures that rule instance (80) fires as well, fact $B(c)$ has already been rederived, so the rederivation stops.

$$T(b, c) \wedge B(b) \rightarrow B(c) \tag{81}$$

The algorithm next proceeds to the final insertion step, in which it computes the consequences of the added facts. Specifically, adding $A(b)$ fires rule instance (82) and so $T(b, f)$ is derived. Insertion is propagated iteratively as long as possible: adding $T(b, f)$ fires rule instance (83) and so $B(f)$ is derived; and adding $B(f)$ fires rule instance (84) and so $B(g)$ is derived. Finally, adding $B(g)$ fires rule instance (85) but $B(c)$ is already part of the materialisation, and so insertion stops.

$$S(b, f) \wedge A(b) \rightarrow T(b, f) \tag{82}$$

$$T(b, f) \wedge B(b) \rightarrow B(f) \tag{83}$$

$$T(f, g) \wedge B(f) \rightarrow B(g) \tag{84}$$

$$T(g, c) \wedge B(g) \rightarrow B(c) \tag{85}$$

As we have already pointed out, the overdeletion step deletes only facts that depend on E^-: in our example, $B(b)$ is not deleted as it does not depend on $A_n(b)$. In this way, DRed can be efficient if not many facts depend on E^-. Nevertheless, overdeletion can lead to redundant work: in our example, facts $B(c)$ and $B(d)$ are overdeleted only to be rederived.

To implement this idea, we need to efficiently identify the overdeleted, rederived, and inserted facts: for the maintenance to be more efficient than recomputing materialisation from scratch, the three steps should consider only a relatively small subset of the rule instances considered while computing the 'old' materialisation. In the rest of this section, we argue that the existing variants of DRed can be inefficient as they sometimes consider certain redundant rule instances. To this end, we first discuss precisely how the derivations from Example 13 are realised. We base our discussion on the formalisation by Staudt and Jarke [28] since their version has been specified more precisely than the one by Gupta et al. [25].

Example 13 *(continued).* Staudt and Jarke [28] formalise the DRed algorithm declaratively: the updated materialisation is computed from the 'old' materialisation by evaluating a *maintenance* datalog program that, for each predicate P, uses several fresh predicates: P^{del}, P^{new}, P, and P^{ins}. The maintenance program consists of three parts, Π^{del}, Π^{red}, and Π^{ins}, which encode the actions in the overdeletion, rederivation, and the insertion steps, respectively. Let I be an 'old' materialisation, and let $I' = \mathsf{mat}(\Pi^{del} \cup \Pi^{red} \cup \Pi^{ins}, I)$ be the 'new' materialisation; the latter can be computed using the seminaïve algorithm. Then, the updated materialisation consists of the P^{new}-facts in I'. We next present programs Π^{del}, Π^{red}, and Π^{ins} for our example program Π. Note that, even though program Π is positive, the overdeletion program Π^{del} uses negation in rule bodies.

To encode the overdeletion step, the example program Π and the dataset E^- are transformed into the *overdeletion program* Π^{del} consisting of rules (86)–(102).

$$R^{del}(x, y) \wedge A_n(x) \rightarrow T^{del}(x, y) \tag{86}$$
$$R(x, y) \wedge A_n^{del}(x) \rightarrow T^{del}(x, y) \tag{87}$$
$$S^{del}(x, y) \wedge A(x) \rightarrow T^{del}(x, y) \tag{88}$$
$$S(x, y) \wedge A^{del}(x) \rightarrow T^{del}(x, y) \tag{89}$$
$$T^{del}(x, y) \wedge B(x) \rightarrow B^{del}(y) \tag{90}$$
$$T(x, y) \wedge B^{del}(x) \rightarrow B^{del}(y) \tag{91}$$
$$T_e^{del}(x, y) \rightarrow T^{del}(x, y) \tag{92}$$
$$B_e^{del}(x, y) \rightarrow B^{del}(x, y) \tag{93}$$
$$\rightarrow A_n^{del}(b) \tag{94}$$
$$A(x) \wedge \text{not}\, A^{del}(x) \rightarrow A^{new}(x) \tag{95}$$
$$A_n(x) \wedge \text{not}\, A_n^{del}(x) \rightarrow A_n^{new}(x) \tag{96}$$
$$B(x) \wedge \text{not}\, B^{del}(x) \rightarrow B^{new}(x) \tag{97}$$
$$R(x, y) \wedge \text{not}\, R^{del}(x, y) \rightarrow R^{new}(x, y) \tag{98}$$
$$S(x, y) \wedge \text{not}\, S^{del}(x, y) \rightarrow S^{new}(x, y) \tag{99}$$
$$T(x, y) \wedge \text{not}\, T^{del}(x, y) \rightarrow T^{new}(x, y) \tag{100}$$
$$T_e(x, y) \wedge \text{not}\, T_e^{del}(x, y) \rightarrow T_e^{new}(x, y) \tag{101}$$
$$B_e(x) \wedge \text{not}\, B_e^{del}(x) \rightarrow B_e^{new}(x) \tag{102}$$

Rules (86)–(93) essentially encode the overdeletion process described earlier. For example, rule (86) intuitively says 'delete $T(x, y)$ for all values x and y such that $R(x, y)$ is deleted and $A_n(y)$ is in the materialisation', and rule (87) says analogously 'delete $T(x, y)$ for all values x and y such that $R(x, y)$ is in the materialisation and $A_n(y)$ is deleted'; thus, (86) and (87) together identify all instances of (71) in which at least one body atom is deleted. Rules (88)–(89) and (90)–(91) are obtained analogously from (72) and (73). Rule (94) is obtained from E^- and it says that $A_n(b)$ is deleted. Finally, rules (95)–(102) simulate deletion by copying each fact that is not deleted into the 'new' materialisation.

To encode the rederivation step, the example program Π is transformed into the *rederivation program* Π^{red} consisting of rules (103)–(111).

$$T^{del}(x, y) \wedge R^{new}(x, y) \wedge A_n^{new}(x) \rightarrow T^{red}(x, y) \tag{103}$$
$$T^{del}(x, y) \wedge S^{new}(x, y) \wedge A^{new}(x) \rightarrow T^{red}(x, y) \tag{104}$$
$$B^{del}(y) \wedge T^{new}(x, y) \wedge B^{new}(x) \rightarrow B^{red}(y) \tag{105}$$
$$T_e^{del}(x, y) \wedge T_e^{new}(x, y) \rightarrow T^{red}(x, y) \tag{106}$$
$$B_e^{del}(x) \wedge B_e^{new}(x) \rightarrow B^{red}(x) \tag{107}$$
$$T^{red}(x, y) \rightarrow T^{new}(x, y) \tag{108}$$
$$B^{red}(x) \rightarrow B^{new}(x) \tag{109}$$
$$T_e^{red}(x, y) \rightarrow T_e^{new}(x, y) \tag{110}$$
$$B_e^{red}(x) \rightarrow B_e^{new}(x) \tag{111}$$

Rules (103)–(107) are the same as (71)–(75), but their bodies are 'restricted' to the overdeleted facts. For example, rule (103) intuitively says 'derive $T(x, y)$ for all values x and y where $R(x, y)$ and $A_n(x)$ hold, but only if $T(x, y)$ has been overdeleted'. In other words, atom $T^{del}(x, y)$ restricts the applicability of the rule so that only the relevant instances of rule (71) are considered. Moreover, rules (108)–(111) copy the rederived facts into the 'new' materialisation. Please note that rules (103)–(107) could put their consequences into the 'new' materialisation directly, but this is how Staudt and Jarke formalised their algorithm.

Finally, to encode the insertion step, the example program Π is transformed into the *insertion program* Π^{ins} consisting of rules (112)–(128).

$$R^{ins}(x, y) \wedge A_n^{new}(x) \rightarrow T^{ins}(x, y) \quad (112)$$
$$R^{new}(x, y) \wedge A_n^{ins}(x) \rightarrow T^{ins}(x, y) \quad (113)$$
$$S^{ins}(x, y) \wedge A^{new}(x) \rightarrow T^{ins}(x, y) \quad (114)$$
$$S^{new}(x, y) \wedge A^{ins}(x) \rightarrow T^{ins}(x, y) \quad (115)$$
$$T^{ins}(x, y) \wedge B^{new}(x) \rightarrow B^{ins}(y) \quad (116)$$
$$T^{new}(x, y) \wedge B^{ins}(x) \rightarrow B^{ins}(y) \quad (117)$$
$$T_e^{ins}(x, y) \rightarrow T^{ins}(y) \quad (118)$$
$$B_e^{ins}(x) \rightarrow B^{ins}(y) \quad (119)$$
$$\rightarrow A^{ins}(b) \quad (120)$$
$$A^{ins}(x) \rightarrow A^{new}(x) \quad (121)$$
$$A_n^{ins}(x) \rightarrow A_n^{new}(x) \quad (122)$$
$$B^{ins}(x) \rightarrow B^{new}(x) \quad (123)$$
$$R^{ins}(x, y) \rightarrow R^{new}(x, y) \quad (124)$$
$$S^{ins}(x, y) \rightarrow S^{new}(x, y) \quad (125)$$
$$T^{ins}(x, y) \rightarrow T^{new}(x, y) \quad (126)$$
$$T_e^{ins}(x, y) \rightarrow T_e^{new}(x, y) \quad (127)$$
$$B_e^{ins}(x) \rightarrow B_e^{new}(x) \quad (128)$$

Rules (112)–(119) are analogous to rules (86)–(93) from the overdeletion program; for example, rule (112) intuitively says 'derive $T(x, y)$ for all values x and y where $R(x, y)$ has been inserted and $A_n(x)$ holds after the update'. Rule (120) is obtained from E^+ and it says that $A(b)$ is inserted. Finally, rules (121)–(128) copy all inserted facts into the 'new' materialisation.

The algorithm by Gupta et al. [25] is closely related and the main differences can be summarised as follows. First, programs Π^{del}, Π^{red}, and Π^{ins} are evaluated in three separate stages; also, the structure of these programs has not been specified precisely, which makes discussing the sources of redundant derivations difficult. Second, rules (103)–(107) contain B^{new} and T^{new} in their heads instead of B^{red} and T^{red}, respectively, which eliminates the need for rules (108)–(111). Third, instead of using rules (95)–(102) to copy the 'surviving' part of the materialisation, the algorithm evaluates Π^{del} and then deletes $P(c_1, \ldots, c_n)$ for each fact $P^{del}(c_1, \ldots, c_n)$; consequently, rather than creating a copy of the materialisation, the DRed version by Gupta et al. [25] updates the materialisation in situ, which eliminates a considerable source of overhead.

Even if programs Π^{del}, Π^{red}, and Π^{ins} are evaluated using an efficient technique such as the seminaïve evaluation, the DRed algorithm can still be inefficient: the overdeletion and the insertion rules can easily derive the same fact several times, and the rederivation rules may contain more joins than necessary. This is demonstrated by the following examples.

Example 14. Let E, Π, and I be as in Example 13, but let $E^- = \{R(b, e), A_n(b)\}$; then, the overdeletion program Π^{del} consists of rules (86)–(102) extended with rule (129).

$$\rightarrow R^{del}(b, e) \quad (129)$$

The seminaïve evaluation of Π^{del} derives $T^{del}(b, e)$ twice: once form rules (129) and (86), and once from rules (94) and (87). Note, however, that rule instance (76) stops firing as soon as one of the body atoms is deleted, and so one of these two derivations is redundant. More generally, if k body atoms of a rule instance have been overdeleted, the approach by Staudt and Jarke derives the head k times, but $k - 1$ of these derivations are unnecessary. Please note that the two derivations are produced by *distinct* rules (86) and (87), and so the seminaïve algorithm provides no guarantee about nonrepetition of rule instances.

Example 15. Problems similar to Example 14 arise in the insertion step: rules (112)–(117) aim to emulate the seminaïve evaluation, but they can derive the same fact several times unnecessarily. For example, let $E^+ = \{A(b), S(b, h)\}$; then, the overdeletion program consists of rules (112)–(128) extended with rule (130).

$$\rightarrow S^{ins}(b, h) \quad (130)$$

After deriving $A^{ins}(b)$, $A^{new}(b)$, $S^{ins}(b, h)$, and $S^{new}(b, h)$ using the rules (120), (121), (130), and (125), respectively, the seminaïve evaluation of Π^{ins} derives $T^{ins}(b, h)$ twice: once from rules (130), (121) and (114), and once from rules (120), (125) and (115). The problem is analogous to the overdeletion one and it arises because rules (114) and (115) are *distinct*.

Example 16. There are two sources of inefficiency in the rederivation step. Let $E = \{T(a_i, b), B_e(a_i) \mid 1 \leq i \leq k\} \cup \{T(b, c)\}$, and so $I = E \cup \{B(a_i) \mid 1 \leq i \leq k\} \cup \{B(b), B(c)\}$. Moreover, let $E^- = \{B_e(a_1)\}$, and so facts $B(b)$ and $B(c)$ are overdeleted. The first problem is that $B(b)$ is rederived $k - 1$ times using rule instances (131), whereas just one rule instance would suffice.

$$B^{del}(b) \wedge T^{new}(a_i, b) \wedge B^{new}(a_i) \rightarrow B^{red}(b) \quad \text{for } 2 \leq i \leq k \quad (131)$$

The second problem is in the rederivation of $B(c)$ using rule instance (132). Atom $B^{new}(b)$ in the body of (132) has been rederived, so $B(b)$ has been overdeleted; but then, $B(c)$ has been overdeleted as well, and so atom $B^{del}(c)$ is redundant in (132).

$$B^{del}(c) \wedge T^{new}(b, c) \wedge B^{new}(b) \rightarrow B^{red}(c) \quad (132)$$

More generally, atom $B^{del}(y)$ in the body of rule (105) is relevant in an instance of the rule only if $T^{new}(x, y)$ or $B^{new}(x)$ are matched to an overdeleted fact; otherwise, atom $B^{del}(y)$ can be safely omitted.

Example 17. The final inefficiency is due to the fact that this approach does not take any advantage of stratification. Let $E = \{T_e(a_i, a_{i+1}) \mid 1 \leq i < k\} \cup \{B_e(a_1), R(a_1, a_2), A_n(a_1)\}$, and so $I = E \cup \{T(a_i, a_{i+1}) \mid 1 \leq i < k\} \cup \{B(a_i) \mid 1 \leq i \leq k\}$. Moreover, let $E^- = \{A_n(a_1)\}$. Since $A_n(a_1)$ is deleted, rule (87) overdeletes $T(a_1, a_2)$, so rule (90) overdeletes $B(a_2)$, and finally rule (91) overdeletes each $B(a_i)$ with $2 \leq i \leq k$. Note, however, that we can stratify the program by assigning rules (71), (72), and (74) to stratum 1 and rules (73) and (75) to stratum 2. If we then update the materialisation by strata, we can rederive $T(a_1, a_2)$ *before* propagating deletion using rules (73) and (75), which in turn prevents the overdeletion of any facts with predicate B.

To address the problems outlined in Examples 14–17, we present in Section 6.2 a new, optimised version of DRed that avoids repeated derivations in overdeletion and rederivation, reduces overdeletion by exploiting stratification, and handles negations. Specifically, we address the following four problems.

First, we ensure that the overdeletion and the insertion steps never repeat derivations as in the seminaïve algorithm—that is, we ensure that no rule instance from the seminaïve evaluation is used more than once to overdelete or insert a fact. To this end, we abandon the purely declarative formalisation and use a more procedural style that mimics the seminaïve algorithm. For example, we implement overdeletion iteratively where, in each iteration, we consider rule instances from the original materialisation that not only have a newly overdeleted body atom, but, crucially, also have no other body atom that has been overdeleted in a previous iteration; we achieve the latter by removing facts immediately after each round of rule application.

Second, we optimise the rederivation step. In particular, we use Π^{red} to identify only atoms that are rederivable in just one step, and we ensure that each fact is rederived just once. We push the rederivation of all other atoms into the insertion step, which ensures that rederivation is done without repeating derivations.

Third, we update the materialisation by strata, which ensures that only facts that truly disappear from a stratum are propagated to the subsequent strata. This can significantly limit the effects of overdeletion in practice.

Fourth, we incorporate negation into our algorithm. Since the rules are stratified, from a conceptual point of view adding an atom can be seen as removing the atom's negation and vice versa. Thus, a simple solution would be to explicitly introduce, for each predicate P, a predicate P_n that contains the complement of the facts of P. Note, however, that storing all of P_n is not practicable since, in typical cases, it would contain orders of magnitude more facts than P. As a possible solution, Gupta et al. [25] suggest storing only the subset of P_n that is used by some rule instance considered in the original evaluation. This, however, still increases the storage requirements and introduces the need to maintain P_n, both of which are undesirable. In contrast, our solution does not require any extra storage, but it relies on evaluating the rules in the three steps in a special way as demonstrated by the following example. As a consequence, we obtain a solution that handles negation in a lightweight and transparent way.

Example 18. Let Π be as in Example 1, and consider the update from Example 2. Due to the addition of $A(b)$, our algorithm must identify that rule instance (15) no longer fires. Conceptually, we can achieve this by evaluating rule (133).

$$R(x, y) \wedge A^{ins}(x) \rightarrow T^{del}(x, y) \tag{133}$$

Intuitively, this rule says 'overdelete $T(x, y)$ for all values x and y for which $R(x, y)$ holds in the "old" materialisation but $A(x)$ has been inserted'—that is, the last condition takes into account that insertion of $A(x)$ is equivalent to the deletion of not $A(x)$. Our solution does not use such rules explicitly, but it is based on this idea. Note that this requires us to process insertion after deletion in each stratum; that is, we cannot first apply deletion to all strata, and then apply insertion to all strata.

6.2. Formalisation

We now formalise our optimised variant of the DRed algorithm. To this end, we extend the rule matching notation from Section 3 to sets of facts. In particular, let I^+, I^-, P, and N be datasets such that $P \subseteq I^+$ and $N \cap I^- = \emptyset$, let Π be a program, and let r be a rule; then, we define $r\big[I^+, I^- \vdots P, N\big]$ and $\Pi\big[I^+, I^- \vdots P, N\big]$ as the following sets, where $\mathsf{inst}_r\big[I^+, I^- \vdots P, N\big]$ is defined in (65). Please note that (134) and (135) are the set versions of (66) and (67), respectively.

$$r\big[I^+, I^- \vdots P, N\big] = \{\mathsf{h}(r') \mid r' \in \mathsf{inst}_r\big[I^+, I^- \vdots P, N\big]\} \tag{134}$$

$$\Pi\big[I^+, I^- \vdots P, N\big] = \bigcup_{r \in \Pi} r\big[I^+, I^- \vdots P, N\big] \tag{135}$$

We define $r\big[I^+, I^-\big]$ and $\Pi\big[I^+, I^-\big]$ analogously. Note that these are analogous to $r\big[\![I^+, I^-]\!\big]$ and $\Pi\big[\![I^+, I^-]\!\big]$, and $r\big[\![I^+, I^- \vdots P, N]\!\big]$ and $\Pi\big[\![I^+, I^- \vdots P, N]\!\big]$ from (66) and (67), respectively, in Section 5.3, but (5) and (6) are sets rather

Algorithm 4 DRED$(E, \Pi, \lambda, I, E^-, E^+)$.

39: $D := A := \emptyset, \quad E^- = (E^- \cap E) \setminus E^+, \quad E^+ = E^+ \setminus E$
40: **for** each stratum index s with $1 \leq s \leq S$ **do**
41: OVERDELETE
42: $R := \{F \in D \cap \mathsf{Out}^s \mid F \in E \setminus E^-$ or there exist $r \in \Pi^s$ and $r' \in \mathsf{inst}_r\big[I \setminus (D \setminus A), I \cup A\big]$ with $\mathsf{h}(r') = F\}$
43: INSERT
44: $E := (E \setminus E^-) \cup E^+, \quad I := (I \setminus D) \cup A$

45: **procedure** OVERDELETE
46: $N_D := (E^- \cap \mathsf{Out}^s) \cup \Pi^s\big[I \,\vdots\, D \setminus A, A \setminus D\big]$
47: **loop**
48: $\Delta_D := N_D \setminus D$
49: **if** $\Delta_D = \emptyset$ **then break**
50: $N_D := \Pi^s_r\big[I \setminus (D \setminus A), I \cup A \,\vdots\, \Delta_D\big]$
51: $D := D \cup \Delta_D$

52: **procedure** INSERT
53: $N_A := R \cup (E^+ \cap \mathsf{Out}^s) \cup \Pi^s\big[(I \setminus D) \cup A \,\vdots\, A \setminus D, D \setminus A\big]$
54: **loop**
55: $\Delta_A := N_A \setminus ((I \setminus D) \cup A)$
56: **if** $\Delta_A = \emptyset$ **then break**
57: $A := A \cup \Delta_A$
58: $N_A := \Pi^s_r\big[(I \setminus D) \cup A \,\vdots\, \Delta_A\big]$

than multisets. In all of these, for readability we omit I^- if it is equal to I^+, and we omit N if $N = \emptyset$. We discuss ways of computing these sets in Section 9.

Our variant of the DRed approach is shown in Algorithm 4. Given a set of explicit facts E, a program Π, a stratification λ, a materialisation I, a set of facts to delete E^-, and a set of facts to add E^+, the algorithm updates I to $\mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$. The algorithm iteratively processes the strata of Π by applying overdeletion (line 41), rederivation (line 42), and insertion (line 43). Gupta et al. [25] suggested in their work that one should update each stratum immediately after the three steps have been applied, but this leads to problems: as we shall see, computing the overdeleted facts for some stratum relies on the state of the materialisation before any updates. Therefore, our algorithm accumulates the overdeleted and the added facts in global datasets D and A, respectively. Our algorithm will ensure that $D \subseteq I$, but not $I \cap A = \emptyset$. Thus, at each point during the algorithm's execution, I is the original materialisation; $I \setminus (D \setminus A)$ is the result of the applied deletions; and $(I \setminus D) \cup A$ is the result of the applied deletions and additions. After processing all strata, the algorithm updates the materialisation in line 44. As the results of our experiments in Section 10.3 suggest, on small updates sets D and A are often (but not always) much smaller than the set I; but then, our algorithm maintains only small sets, unlike the variant by Staudt and Jarke [28]. We next discuss how overdeletion, rederivation, and insertion steps are applied to each stratum.

Overdeletion is realised using a form of 'inverse seminaïve' strategy, which ensures that each rule instance is considered at most once. The structure is similar to Algorithm 1: we identify the initial set N_D of atoms to be deleted (line 46), and then we enter a loop (lines 47–51) in which we identify the subset of Δ_D of N_D that has not been processed yet (line 48), we compute the facts that depend on Δ_D (line 50), and we update D (line 51). The differences to Algorithm 1 can be summarised as follows.

- Algorithm 4 fires *all* (i.e., both nonrecursive and recursive) rules in line 46, whereas Algorithm 1 fires just the nonrecursive rules in line 3. Intuitively, both recursive and nonrecursive rule instances can stop firing due to deletion in previous strata. Thus, line 46 computes all instances of the rules in Π^s where at least one body atom no longer holds—that is, with either a positive atom in $D \setminus A$ or a negative atom in $A \setminus D$. By constraining all body atoms in a single condition, we overcome the problems highlighted in Example 14. Furthermore, to obtain only rule instances that fired in the 'old' materialisation, we restrict all atoms to I—that is, we need the 'old' materialisation and thus cannot update I immediately after each stratum.
- In line 50 it suffices to fire just the recursive rules: facts computed in line 46 are all from stratum s, and such facts occur only in the bodies of recursive rules. The key difference to line 8 is in how we ensure that no rule instance fires more than once. First, at least one positive atom must be in Δ_D so that a rule instance fires because of a fact that was derived in the most recent iteration; moreover, no positive body atom should be in $D \setminus A$ so that a rule instance never fires twice. Second, all rule instances that no longer fire due to a negative atom being added to A are considered in line 46, and so in line 50 we exclude such rule instances by restricting the negative atoms to $I \cup A$.
- Set D identifies a growing subset of I that must be excluded from I. Therefore, Δ_D must be added to D only *after* the consequences of Δ_D are computed in line 50, reflecting the 'inverse seminaïve' nature of the computation. This is in contrast to Algorithm 1, where Δ is added to I before computing the consequences of Δ in line 8.

As we have already explained in Example 16, we can safely rederive all facts derivable in more than one step during insertion; therefore, in line 42 we compute the set R of facts that can be rederived using just one step. To this end, we consider each fact F that was added to D in stratum s, and we add F to R if F is explicit or if we find a rule instance that rederives F—that is, whose positive body atoms are among the 'surviving' facts $I \setminus (D \setminus A)$ and whose negative atoms are in $I \cup A$. Note that, for each F, it suffices to find one such rule instance; thus, if we solve this step by identifying each rule $r \in \Pi^s$ whose head unifies with F, apply the unifier to the body of r, and evaluate the body as a query that returns just one answer (if one exists), we address the concerns from Example 16. Please note that line 42 uses $\mathsf{inst}_r\big[I \setminus (D \setminus A), I \cup A\big]$ to ensure that all rule instances considered here are from the 'old' materialisation; the algorithm would be correct if we used $\mathsf{inst}_r\big[(I \setminus D) \cup A\big]$ instead, but then one-step rederivation could also consider rule instances that exist solely in the 'new' materialisation, and these could be performed again later in line 53 in the insertion phase; consequently, the algorithm could repeat inferences needlessly.

The insertion step uses set R to complete the rederivation and derive the new facts by a variant of the seminaïve strategy. The structure is analogous to Algorithm 1, but with the difference that the rules are evaluated w.r.t. the 'new' materialisation, which is given by $(I \setminus D) \cup A$, and that the derived facts are added to A. Moreover, to identify in line 53 the rule instances that fire after, but not before the update due to changes in the previous strata, we must, as in overdeletion, consider all, and not just nonrecursive rules. We therefore consider rule instances for which one positive body atom was added (i.e., that is in $A \setminus D$), or a negative body atom was removed (i.e., that is in $D \setminus A$).

Theorem 19 summarises the properties of the DRed algorithm: property (a) ensures correctness of our algorithm; property (b) says that, during overdeletion, the algorithm considers only rule instances from the 'old' materialisation without any repetition; and property (c) says that, during rederivation and insertion, the algorithm considers only rule instances from the 'new' materialisation without any repetition. Note that, if $E^- = E$ and $E^+ \cap E^- = \emptyset$ and the rules do not contain any constants, then the algorithm considers all rule instances from the 'old' materialisation in Overdelete, and all rule instances from the 'new' materialisation in Insert. In Section 7 we present the FBF algorithm and show that the DRed algorithm is a special case of FBF; thus, Theorem 19 is a special case of Theorem 25, and we prove the latter in Appendix B.

Theorem 19. *Let Π be a program, let λ be a stratification of Π, let E be a dataset, let $I = \mathsf{mat}(\Pi, E)$, and let E^- and E^+ be two datasets. Then,*

(a) *Algorithm 4 correctly updates I to $I' = \mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$;*
(b) *lines 46 and 50 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[I\big]$ without repetition; and*
(c) *lines 42, 53, and 58 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[I'\big]$ without repetition.*

We finish this section with a note that sets I, A, and D can overlap in our formalisation; however, we could have equivalently formalised the algorithm so that set A is disjoint with sets D and I: instead of line 57, we add two lines that first remove $D \cap \Delta_A$ from D and then add $\Delta_A \setminus I$ to A; then, in the rest of the algorithm we can replace all occurrences of $A \setminus D$, $D \setminus A$, and $I \setminus (D \setminus A)$ with just A, D, and $I \setminus D$, respectively. We choose, however, a style of formalisation where all sets grow monotonically.

7. The forward/backward/forward algorithm

When a fact is derived by several rule instances, even if one rule instance no longer fires after an update, other rule instances often still 'survive' the update. In such cases the DRed algorithm can be inefficient, as illustrated by the following example.

Example 20. Consider again applying the DRed algorithm to our running example (which was introduced in Example 1 and Fig. 1) as described in Example 13. Before the update, rule instances (78) and (81) both fire and derive $B(c)$. After the update, rule instance (78) no longer fires, and so the DRed algorithm overdeletes $B(c)$; moreover, the algorithm also overdeletes $B(d)$ because it is derived by rule instance (79) that uses $B(c)$. However, rule instance (81) still fires and so $B(c)$ still holds after the update; hence, deleting $B(c)$ and $B(d)$ only to rederive them later is unnecessary.

In this section we present a new materialisation maintenance algorithm, called Forward/Backward/Forward (FBF), that addresses these drawbacks. The algorithm extends our Backward/Forward (B/F) algorithm [10]. It uses forward chaining similarly to DRed (procedure Overdelete in Algorithm 4) to identify facts that can be affected by an update. However, for each such fact, the algorithm can use a form of backward chaining to determine whether the fact still holds after the update, and it does not delete the fact if a proof is found. This can limit unnecessary propagation of deletion as shown in the following example.

Example 21. Consider again the update from Example 20. In a way similar to DRed, the FBF algorithm determines that rule instance (78) does not fire after the update and so fact $B(c)$ may not hold after the update. At this point, however, the algorithm looks for other rule instances that can derive $B(c)$ from the 'surviving' facts. Thus, FBF determines that $B(c)$ is

derived by rule instance (81), which contains $T(b, c)$ and $B(b)$ in the body. Now $T(b, c)$ is explicitly present in the input dataset, but $B(b)$ is not, and so FBF recursively considers $B(b)$ and determines that it is derived by rule instance (136).

$$T(a, b) \wedge B(a) \rightarrow B(b) \tag{136}$$

Facts $T(a, b)$ and $B(a)$ are both explicitly present in the input dataset after the update, and so $B(b)$ holds after the update; this implies that $B(c)$ hold as well and so $B(c)$ is not overdeleted; but then, $B(d)$ is not overdeleted either.

As Example 21 demonstrates, FBF prevents unnecessary overdeletion/rederivation at the expense of additional backward chaining, and the relative cost of these operations depends on the specific example; we discuss in Section 8 the situations in which either approach is likely to be more efficient. Because neither approach is universally better, we parameterise our algorithm with a strategy that can adapt the algorithm's behaviour; for example, the strategy could decide to use backward chaining only up to a certain number of steps and fall back on overdeletion if this number is exceeded. Thus, DRed is an instances of FBF parameterised with a strategy that never uses backward chaining. At the other end of the spectrum, B/F is obtained by a strategy that fully explores all proofs and thus makes the deletion step exact. Between these two extremes, a strategy can, for example, decide to explore the proofs up to some predetermined depth, or use any other heuristic to try to minimise the overall work.

In the rest of this section we discuss the details behind this idea. Specifically, in Section 7.1 we discuss backward chaining and some relevant issues informally, and then in Section 7.2 we present the algorithm formally and state its correctness.

7.1. Overview

As Example 21 suggests, the FBF algorithm is similar to DRed (cf. Algorithm 4), but with the following difference in line 48: for each fact $F \in N_D \setminus D$, add F to Δ_D only if F is not proved from the 'surviving' facts via backward chaining. In the rest of this section we discuss how to realise backward chaining in a way that guarantees both efficiency and termination.

Example 22. Let Π be the subset of our running example program containing only the rule (137), and let E be as shown in (138); hence, the materialisation I of Π w.r.t. E is as shown in (139).

$$T(x, y) \wedge B(x) \rightarrow B(y) \tag{137}$$

$$E = \{B(a), B(b), T(a, b), T(b, c), T(c, b), T(c, d), T(d, e)\} \tag{138}$$

$$I = E \cup \{B(c), B(d), B(e)\} \tag{139}$$

We assume that the atoms unifiable with $T(x, y)$ belong to a lower stratum than the atoms unifiable with $B(x)$. Finally, let us delete $B(b)$—that is, $E^- = \{B(b)\}$. Fact $B(b)$ is still derived after the update, and the materialisation remains unchanged.

When applied to this input, the DRed algorithm overdeletes $B(b)$, which in turn leads to the overdeletion of $B(c)$, $B(d)$, and $B(e)$; however, all of these facts hold after the update and so they are rederived later.

In contrast, the FBF algorithm eagerly determines that $B(b)$ holds after the update and thus stops overdeletion. To determine that $B(b)$ holds, the algorithm attempts to find a proof using backward chaining by examining all rule instances from the current materialisation that derive $B(b)$. To this end, the algorithm attempts to unify $B(b)$ with the head of each rule where unification is possible; in our case (137) is the only such rule, and unification produces the partially instantiated rule (140). The algorithm next evaluates the body of (140) on the materialisation to identify rule instances that fire before the update and derive $B(b)$; this produces substitutions $\sigma_1 = \{x \mapsto a\}$ and $\sigma_2 = \{x \mapsto c\}$ corresponding to rule instances (141) and (142).

$$T(x, b) \wedge B(x) \rightarrow B(b) \tag{140}$$

$$T(a, b) \wedge B(a) \rightarrow B(b) \tag{141}$$

$$T(c, b) \wedge B(c) \rightarrow B(b) \tag{142}$$

The facts occurring in the body of (141) are both explicitly given in the input dataset after the update, so the rule instance still fires and $B(b)$ holds after the update. Hence, $B(b)$ is proved so the algorithm does not need to consider (142).

Note, however, that backward chaining can consider (142) before (141). Since $B(c)$ does not occur in the input dataset, the algorithm recursively tries to prove it using backward chaining: it unifies $B(c)$ with the head of rule (137) obtaining the partially instantiated rule (143), and evaluates its body over the materialisation to obtain rule instance (144).

$$T(x, c) \wedge B(x) \rightarrow B(c) \tag{143}$$

$$T(b, c) \wedge B(b) \rightarrow B(c) \tag{144}$$

Fact $B(b)$ is not explicitly present in the input, and attempting to recursively prove it would lead to an infinite loop; hence, this basic idea of backward chaining must be refined to ensure termination. The obvious solution is based on the observation that no proof should contain a fact that is used to prove itself. In our example, $B(b)$ should not be used to prove $B(b)$ and so, when examining the body of (144), no recursive attempt to prove $B(b)$ should be made because there is already an active recursive call for $B(b)$. By simply returning from the nested recursive attempted to prove $B(b)$, the algorithm will backtrack, consider rule instance (141), and find a proof for $B(b)$.

The simple backward chaining strategy outlined in Example 22 is correct, but, as the following example demonstrates, it can be very inefficient.

Example 23. Let Π contain rule (137), and let E contain facts $B(a_1)$ and $T(a_i, a_j)$, for $1 \leq i, j \leq n$ for some $n \geq 1$. The materialisation of Π and E extends the explicit facts with $B(a_i)$ for each i with $1 < i \leq n$. Rule instance (145) fires for each pair of i and j with $1 \leq i, j \leq n$, and so a total of n^2 rule instances fire during the materialisation.

$$T(a_i, a_j) \wedge B(a_i) \rightarrow B(a_j) \tag{145}$$

Now assume that we delete $B(a_1)$; clearly, none of the facts $B(a_i)$, $1 \leq i \leq n$, survives the update. An attempt to prove $B(a_1)$ using a variant of backward chaining outlined in Example 22 considers all instances of (145) with $j = 1$ and tries to recursively prove each $B(a_i)$ with $i > 1$. The termination condition prevents considering $B(a_1)$ again, but the algorithm recursively tries to prove $B(a_i)$ using any $B(a_{i'})$ different from $B(a_1)$ and $B(a_i)$. In other words, the algorithm tries to prove $B(a_1)$ by trying (and failing) to prove facts $B(a_2), B(a_3), \ldots, B(a_n)$ in any possible order, and so backward chaining incurs $(n-1)!$ recursive calls, which is exponentially worse than computing the 'old' materialisation. Hence, the simple backward chaining approach from Example 22 is not suitable for practice.

The FBF algorithm addresses the problem outlined in Example 23 by considering each fact during backward chaining at most once, and by using a separate forward chaining step that ensures that the order in which rule instances are considered does not matter. The following example illustrates this idea.

Example 24. Let Π be the subset of our running example program containing only the rule (137), and let E be as shown in (146); hence, the materialisation I of Π w.r.t. E is as shown in (147).

$$E = \{B(a), B(b), B(c), T(a,b), T(b,c), T(c,b), T(c,d), T(d,e)\} \tag{146}$$

$$I = E \cup \{B(d), B(e)\} \tag{147}$$

Let us delete $B(b)$ and $B(c)$—that is, $E^- = \{B(b), B(c)\}$. The FBF algorithm must determine that, although facts $B(b)$ and $B(c)$ are no longer explicit, they 'survive' the update.

The FBF algorithm maintains a set C of *checked* facts; moreover, it perform backwards chaining for a fact F only if $F \notin C$ holds and, if so, it adds F to C. Thus, each fact F and the rule instances deriving it are examined during backward chaining at most once, which prevents problems from Example 23. This, however, leads to another problem: after backward chaining for F finishes, the algorithm cannot be certain whether F holds or not. To see why this is the case, assume that C is initially empty, and that the algorithm first attempts to prove $B(b)$. Moreover, assume that the algorithm first considers rule instance (142), and so it recursively tries to prove $B(c)$. The algorithm adds $B(c)$ to C and considers the only rule instance (144) that derives $B(c)$; thus, the algorithm recursively tries to prove $B(b)$, which fails because $B(b) \in C$ holds at this point—that is, $B(b)$ has already been checked. Thus, the algorithm must abandon its proof attempt for $B(c)$, but it still does not know whether $B(c)$ holds or not; moreover, since $B(c)$ has already been checked (i.e., $B(c) \in C$ holds), any future attempt to prove $B(c)$ will fail.

The FBF algorithm solves this problem by maintaining another set P of *proved* facts. Each proved fact is added to P and forward chaining is used to compute all consequences of P and Π. In our example, after abandoning a proof attempt for $B(c)$, the algorithm next considers rule instance (141) and recursively tries to prove $B(a)$. Since $B(a)$ is explicitly given in the input dataset, it is added to P; moreover, forward chaining is next applied to P and it derives $B(b)$ and $B(c)$. Fact $B(c)$ is thus derived eventually, but without having to check it more than once, which ensures correctness of the deletion step.

Finally, note that facts $B(b)$ and $B(c)$ are proved and so the algorithm does not need to consider facts $B(d)$ and $B(e)$, which was our goal in the first place. However, exhaustively applying forward chaining to P and Π derives $B(d)$ and $B(e)$, so there is no saving in terms of rule instances considered or facts derived compared to rematerialisation from scratch. Because of that, forward chaining of P and Π is modified so that facts are proved only if they have been previously checked—that is, if they are in C. In our example, set C contains $B(a)$, $B(b)$, and $B(c)$, but not $B(d)$ and $B(e)$; thus, the modified forward chaining proves only $B(b)$ and $B(c)$, but not $B(d)$ or $B(e)$, as desired.

Algorithm 5 FBF($E, \Pi, \lambda, I, E^-, E^+$).

59: $D := A := \emptyset, \quad E^- = (E^- \cap E) \setminus E^+, \quad E^+ = E^+ \setminus E$
60: **for** each stratum index s with $1 \leq s \leq S$ **do**
61: $\quad B := C := P := Y := \emptyset$
62: $\quad$ DeleteUnproved
63: $\quad A := A \cup (B \cap D \cap P)$
64: $\quad R := \{F \in B \cap (D \setminus P) \mid F \in (E \setminus E^-) \cup Y$ or there exist $r \in \Pi^s$ and $r' \in \mathsf{inst}_r\big[I \setminus (D \setminus A), I \cup A\big]$ with $\mathsf{h}(r') = F\}$
65: $\quad$ Insert from Algorithm 4
66: $E := (E \setminus E^-) \cup E^+, \quad I := (I \setminus D) \cup A$

67: **procedure** DeleteUnproved
68: $\quad N_D := (E^- \cap \mathsf{Out}^s) \cup \Pi^s\big[I \,\vdots\, D \setminus A, A \setminus D\big]$
69: $\quad$ **loop**
70: $\quad\quad \Delta_D := \emptyset$
71: $\quad\quad$ **for** $F \in N_D \setminus D$ **do**
72: $\quad\quad\quad$ Check(F)
73: $\quad\quad\quad$ **if** $F \notin P$ **then** $\Delta_D := \Delta_D \cup \{F\}$
74: $\quad\quad$ **if** $\Delta_D = \emptyset$ **then break**
75: $\quad\quad N_D := \Pi^s_{\mathsf{r}}\big[I \setminus (D \setminus A), I \cup A \,\vdots\, \Delta_D\big]$
76: $\quad\quad D := D \cup \Delta_D$

77: **procedure** Check(F)
78: $\quad$ **if** $F \notin C$ **then**
79: $\quad\quad$ **if** AbortBranch **then** $B := B \cup \{F\}$
80: $\quad\quad$ **else if** Saturate(F) $= \mathsf{f}$ **then**
81: $\quad\quad\quad$ **for** each $r \in \Pi^s_{\mathsf{r}}$ and each $r' \in \mathsf{inst}_r\big[I \setminus ((D \cup \Delta_D) \setminus (A \cup B)), I \cup A\big]$ such that $\mathsf{h}(r') = F$ **do**
82: $\quad\quad\quad\quad$ **for** $G \in \mathsf{b}^+(r') \cap \mathsf{Out}^s$ **do**
83: $\quad\quad\quad\quad\quad$ Check(G)
84: $\quad\quad\quad\quad\quad$ **if** $G \in B \setminus P$ **then** $B := B \cup \{F\}$
85: $\quad\quad\quad\quad\quad$ **if** $F \in P$ **then return**

86: **function** Saturate(F)
87: $\quad C := C \cup \{F\}$
88: $\quad$ **if** $F \in Y \cup (E \setminus E^-)$ or there exist $r \in \Pi^s_{\mathsf{nr}}$ and $r' \in \mathsf{inst}_r\big[I \setminus (D \setminus A), I \cup A\big]$ such that $\mathsf{h}(r') = F$ **then**
89: $\quad\quad N_P := \{F\}$
90: $\quad\quad$ **loop**
91: $\quad\quad\quad \Delta_P := (N_P \cap C) \setminus P, \quad Y := Y \cup N_P \setminus C$
92: $\quad\quad\quad$ **if** $\Delta_P = \emptyset$ **then return** t
93: $\quad\quad\quad P := P \cup \Delta_P$
94: $\quad\quad\quad N_P := \Pi^s_{\mathsf{r}}\big[P \cup (\mathsf{Out}^{<s} \cap (I \setminus (D \setminus A))), I \cup A \,\vdots\, \Delta_P\big]$
95: $\quad$ **else return** f

7.2. Formalisation

Algorithm 5 formalises the ideas from Section 7.1. Procedure DeleteUnproved is analogous to procedure Overdelete in the DRed algorithm: the main difference is that, instead of adding each fact $F \in N_D \setminus D$ to Δ_D straight away, the FBF algorithm tries first to prove F (line 72) and adds it to Δ_D only if a proof cannot be found (line 73).

Procedure Saturate applies forward chaining to the set P of proved facts as explained in Example 24: in line 88, the procedure tries to prove F from the explicit facts or using nonrecursive rules; and in lines 90–94 it computes the consequences of the recursive rules. The positive atoms of the recursive rules are evaluated in line 94 in $P \cup (\mathsf{Out}^{<s} \cap (I \setminus (D \setminus A)))$: the atoms from the previous strata are matched against the updated materialisation, but the atoms from the current stratum are matched in P. Moreover, as we explained in Example 24, Saturate should not compute the consequences of P and Π^s that have not been checked. Therefore, in line 91, only the subset of N_P that is contained in C is added to Δ_P, whereas all other facts are added to the set Y of *delayed* facts. Intuitively, the set Y contains all facts that follow from P and Π^s_{r}, but that have not been checked yet. If any such fact is checked at a later point, it will be proved in line 88 without having to apply the rules again; hence, the set of delayed facts ensures that Saturate considers each rule instance at most once.

Procedure Check attempts to prove F. Line 78 ensures that F is checked at most once, as we discussed in Example 24. Now to ensure that FBF subsumes DRed, line 79 is parameterised by a function AbortBranch that determines whether the current branch should be aborted. If a branch is aborted, F is added to the set B of *blocked* facts, which means that not all recursive rule instances that can prove F have been explored. The status of blocked facts is not known after DeleteUnproved finishes. Moreover, facts can be checked in an arbitrary order and they are never reexamined; thus, if we later encounter a fact F' that depends on a blocked and unproved fact G, the rule instances of F' have not been fully explored and so F' is also blocked (line 84). Note that a blocked fact can subsequently be checked and possibly also proved. However, each fact $F \in D \setminus B$ has been fully explored, and $F \notin P$ holds since F is added to D in line 73 only if F is not contained in P.

When DeleteUnproved finishes, the set $B \cap D$ contains facts that have been deleted, but not completely explored during backward chaining. Of these, the facts in $B \cap D \cap P$ have been proved, so they are simply added back to A in line 63; the

algorithm would be correct had we added these facts to R, but then the insertion phase could consider in line 53 or 58 a rule instance that has already been considered in line 88 or 94. In contrast, set $B \cap (D \setminus P)$ contains facts whose status is truly unknown and so these must be proved using one-step rederivation (line 64). This is done as in DRed, but with the difference that facts in Y have already been proved and are thus excluded from rederivation; this ensures that one-step rederivation does not consider rule instances from Saturate. Note that, if all branches are explored in full, then $B = \emptyset$ holds in line 64, and so one-step rederivation can be omitted; in other words, the deletion phase of FBF is precise in this case. Moreover, if all branches are aborted, then $B = D$ and $P = \emptyset$ and so line 63 is superfluous and line 64 becomes equal to line 42—that is, the algorithm becomes equivalent to DRed.

Procedure Check does not directly prove F; instead, it calls Saturate to try to prove F from the explicit facts, the nonrecursive rules, or the facts proved thus far. If F cannot be proved, since Saturate already considers all nonrecursive rules, the only remaining possibility is that F can be proved using a recursive rule; hence, the procedure then examines in lines 81–85 each instance r' of a recursive rule that derives F in the 'old' materialisation and it tries to prove all body atoms of r' from the current stratum. If at any point F becomes proved, there is no need to consider the remaining rule instances (line 85). Line 81 optimises backward chaining: as per our discussion above, the status of all facts in $(D \cup \Delta_D) \setminus (A \cup B)$ will be known after the deletion phase and all such facts will be eventually added to D; moreover, as we have already stated above, $D \setminus B$ and P are disjoint. Thus, if there exists a body atom $F \in \mathsf{b}^+(r')$ satisfying $F \in (D \cup \Delta_D) \setminus (A \cup B)$, atom F cannot be proved, and so the rule instance cannot be considered in Saturate and we can consequently skip the rule instance in backward chaining.

Theorem 25, proved in Appendix B, summarises the properties of the FBF algorithm. In particular, property (a) ensures the algorithm's correctness. Property (b) says that the deletion step considers at most once the rule instances from the 'old' materialisation, and property (c) says that Saturate, the one-step rederivation step, and the insertion step (i.e., Insert) jointly consider at most once the rule instances from the 'new' materialisation. Property (d) says that backward chaining considers at most once the instances of recursive rules from the 'old' materialisation. Property (e) says that, if branches are never aborted, then backward chaining is precise and one-step rederivation is unnecessary. Finally, property (f) says that DRed is a special case of FBF if backward chaining is 'switched off', which implies Theorem 19 as a corollary. Note that, if $E^- = E$ and $E^+ \cap E^- = \emptyset$ and the rules do not contain any constants, then the algorithm considers all rule instances from the 'old' materialisation in DeleteUnproved, and all rule instances from the 'new' materialisation in Insert. Moreover, Proposition 30 in Section 8.3 shows that inputs exist on which FBF considers all recursive rule instances from the 'old' materialisation during backward chaining.

Theorem 25. *Let Π be a program, let λ be a stratification of Π, let E be a dataset, let $I = \mathsf{mat}(\Pi, E)$, and let E^- and E^+ be two datasets. Then,*

(a) *Algorithm 5 correctly updates I to $I' = \mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$;*
(b) *lines 68 and 75 consider rule instances from $\bigcup_{r \in \Pi_r} \mathsf{inst}_r[I]$ without repetition;*
(c) *lines 53, 58, 64, 88, and 94 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r[I']$ without repetition, and moreover the rule instances considered in lines 88, 94, and 64 are also in $\bigcup_{r \in \Pi} \mathsf{inst}_r[I]$;*
(d) *line 81 considers recursive rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r[I]$ without repetition;*
(e) *if branches are never aborted in line 79, then one can safely omit line 63 and one-step rederivation in line 64; and*
(f) *if branches are always aborted in line 79, then the algorithm becomes the same as DRed.*

8. Comparing counting, DRed, and FBF

The three algorithms we presented in Sections 5, 6, and 7 solve the same problem in quite different ways, so it is natural to ask whether any of these approaches is generally 'better'. In this section we answer this question negatively by presenting for each algorithm an input on which the algorithm outperforms the other two. Hence, only an empirical comparison seems realistic, and we consider it in Section 10. Nevertheless, we can obtain a general description of the kinds of inputs that might be more suited to each of the three algorithms. In this section, we assume that FBF never aborts backward chaining in line 79.

8.1. Comparison metric

All three algorithms are based on variants of fixpoint evaluation, so it is straightforward to see that their worst-case complexity is the same as for standard datalog [11]: ExpTime in combined complexity (i.e., measured in the size of the program and the input dataset), and PTime in data complexity (i.e., measured in the size of the explicit dataset E only). Thus, computational complexity does not allow us to differentiate the three algorithms. In our experience, however, the cost of rule firing (either during forward or backward chaining) typically dominates the performance of all three algorithms: firing a rule requires matching the rule body, and deriving a fact requires duplicate elimination and index update. Hence, we use the number of considered rule instances as a conceptual comparison metric. An ideal materialisation maintenance algorithm would consider only the rule instances that no longer fire and rule instances that fire only after the update as

this would limit the algorithm's work precisely to the difference between the 'old' and the 'new' materialisation. We capture this intuition using the following optimality notion.

Definition 26. An incremental maintenance algorithm is *optimal* on Π, λ, E, E^-, and E^+ if it considers (in the sense of Theorems 12, 19, and 25) only the following rule instances, where $I = \mathsf{mat}(\Pi, E)$ and $I' = \mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$:

$$\bigcup_{r \in \Pi} \left[\left(\mathsf{inst}_r[I] \setminus \mathsf{inst}_r[I'] \right) \cup \left(\mathsf{inst}_r[I'] \setminus \mathsf{inst}_r[I] \right) \right].$$

Even an optimal algorithm can be less efficient than rematerialisation 'from scratch': if the entire explicit dataset E is deleted, then rematerialisation requires no work at all, whereas an optimal algorithm considers all rule instances from the 'old' materialisation. In Section 10 we explore empirically the trade-off between rematerialisation and materialisation maintenance.

In the rest of this section, we analyse the behaviour of our algorithms with respect to this optimality criterion. In particular, in Section 8.2 we argue that, for nonrecursive programs, counting is optimal in the above sense, whereas DRed and FBF are not as they use variants of backward chaining in order to check the existence of alternative derivations of overdeleted facts. In Section 8.3 we show that, for recursive programs, for each algorithm there exists a 'bad' input on which the algorithm redoes most of the work from the 'old' materialisation, and a 'good' input on which the algorithm exhibits the ideal behaviour. Interestingly, the same input can be 'bad' for one but 'good' for another algorithm. Moreover, in Section 8.4 we observe that DRed and FBF exhibit optimal behaviour on updates where no rule instances stop firing after the update. Finally, in Section 8.5 we discuss how program stratification affects the performance of all algorithms. In particular, we show that, although stratification usually reduces the number of rule instances considered, on some inputs it can surprisingly increase the overall work.

8.2. Nonrecursive programs

The counting algorithm is optimal for nonrecursive programs, as shown in Proposition 27.

Proposition 27. *The counting algorithm is optimal for Π, λ, E, E^-, and E^+ whenever Π is nonrecursive w.r.t. λ.*

Proof. Since Π is nonrecursive w.r.t. λ, each fact is associated with just one counter that reflects the number of derivations using facts from the previous strata. Thus, a fact is deleted if and only if its counter reaches zero. Moreover, I_{on} and I_{no} contain precisely the differences between the 'old' and 'new' materialisation, so line 23 considers precisely the rule instances that no longer fire, and line 24 considers precisely the rule instances that did not fire before the update. Hence, the algorithm is optimal. □

When Π is nonrecursive w.r.t. λ, then DRed and FBF exhibit similar behaviour: DRed reproves facts in the one-step rederivation step after overdeletion (line 42 of Algorithm 4), while FBF reproves facts eagerly (line 88 of Algorithm 5); however, since all rules are nonrecursive w.r.t. λ, all facts that still hold after the update get reproved before proceeding to subsequent strata. Thus, both DRed and FBF undo only rule instances that no longer fire and apply rule instances that fire only after the update, just like the counting algorithm. However, for each fact affected that still holds after the update, both algorithms consider at least one rule instance from the 'old' materialisation that proves the fact, so neither algorithm is optimal.

Proposition 28. *There exist a program Π, stratification λ, and datasets E, E^-, and E^+ such that Π is nonrecursive w.r.t. λ for which neither DRed nor FBF are optimal.*

8.3. Recursive programs

Proposition 29 shows that, on positive programs recursive w.r.t. a stratification, neither algorithm is optimal for deletions in general, and the counting algorithm may not be optimal even for insertions. Its proof suggests that FBF is more efficient than DRed if deleted facts have short proofs, whereas DRed is more efficient than FBF if not many facts depend on the deleted facts.

Proposition 29. *There exists a single-rule program Π that is recursive w.r.t. any stratification such that, for each of the three algorithms, there exist datasets E, E^-, and $E^+ = \emptyset$ on which at least one other algorithm performs fewer derivations, but the materialisation does not change and neither of the three algorithms is optimal. In addition, there exist E, $E^- = \emptyset$, and E^+ on which the materialisation does not change but the counting algorithm is not optimal.*

'Old' Trace								Facts	'New' Trace							
$\boldsymbol{N}_1^2$	$\boldsymbol{N}_2^2$	$\cdots$	$\boldsymbol{N}_{i-1}^2$	$\boldsymbol{N}_i^2$	$\boldsymbol{N}_{i+1}^2$	$\cdots$	$\boldsymbol{N}_n^2$		$\boldsymbol{N}_1^2$	$\boldsymbol{N}_2^2$	$\cdots$	$\boldsymbol{N}_{i-1}^2$	$\boldsymbol{N}_i^2$	$\boldsymbol{N}_{i+1}^2$	$\cdots$	$\boldsymbol{N}_{n-i+1}^2$
1								$B(a_1)$	1							
	1							$B(a_2)$		1						
								$\vdots$								
			1					$B(a_{i-1})$				1				
				1^*				$B(a_i)$	1^*				1			
					1^*			$B(a_{i+1})$		1^*						
								$\vdots$								
							1^*	$B(a_n)$								1^*

Fig. 5. The traces of the counting algorithm in the proof of Proposition 29.

Proof. Let Π contain just the recursive rule (148), and let E be as specified in (149) for some n. We consider stratification λ that maps $T(x, y)$ to stratum 1, and $B(y)$ and $B(x)$ to stratum 2, but the claims hold analogously if $T(x, y)$ is mapped to stratum 2.

$$T(x, y) \wedge B(x) \rightarrow B(y) \tag{148}$$

$$E = \{ B(a_1), T(a_1, a_2), \ldots, T(a_{n-1}, a_n) \} \tag{149}$$

We first consider the counting algorithm, whose trace on Π and E is shown on the left-hand side of Fig. 5, but facts with the predicate T in stratum 1 are omitted since they are not derived. Now consider adding $B(a_i)$ to E for some i with $1 < i \leq n$. The materialisation does not change since rule instance $T(a_{i-1}, a_i) \wedge B(a_{i-1}) \rightarrow B(a_i)$ has already been considered in the 'old' materialisation; however, the trace changes to the one shown on the right-hand side of Fig. 5: each fact $B(a_j)$ with $j \geq i$ is now derived for the first time in iteration $j - i + 1$ instead of i. To update the trace, the counting algorithm considers each rule instance $T(a_{j-1}, a_j) \wedge B(a_{j-1}) \rightarrow B(a_j)$ with $j > i$ twice: once in iteration $j - i + 1$ to increment the counter of $B(a_j)$ in $\boldsymbol{N}_j^2$, and once in iteration i to decrement the counter of $B(a_j)$ in $\boldsymbol{N}_j^2$. Conversely, if we now remove $B(a_i)$, the algorithm again considers all these rule instances to update the trace to the original one. All rule instances from the 'old' materialisation are considered for $i = 2$.

Now consider applying DRed and FBF to $E \cup \{B(a_i)\}$ with $i > 1$, $E^- = \{B(a_i)\}$, and $E^+ = \emptyset$. Again, the materialisation does not change, but both algorithms perform some work, so they are not optimal. The DRed algorithm then overdeletes each $B(a_j)$ with $j \geq i$ only to rederive it in the one-step rederivation and the insertion steps; each of these two steps considers $n - i + 1$ rule instances, and so the update considers $2(n - i + 1)$ rule instances in total. In contrast, the FBF algorithm proves $B(a_i)$ using backward chaining, which involves considering i rule instances during backward chaining and i rule instances during forward chaining; thus, the update considers $2i$ rule instances in total. Thus, FBF considers fewer rule instances if $i < (n + 1)/2$, DRed considers fewer rule instances if $i > (n + 1)/2$, and the algorithms are tied if $i = (n + 1)/2$. □

We can compare the algorithms based on the worst-case number of rule instances considered. Properties (b) and (c) of Theorems 12 and 19 ensure that the counting and the DRed algorithm both consider at most all rule instances from the 'old' and all rule instances from the 'new' materialisation, and so the proof of Proposition 29 illustrates their worst-case behaviour. The picture is more complex for FBF: property (b) of Theorem 25 ensures that DeleteUnproved considers at most all rule instances from the 'old' materialisation, and property (c) ensures that Saturate, one-step rederivation, and Insert consider at most all rule instances from the 'new' materialisation, which is analogous to DRed; however, property (d) shows that the instances of the recursive rules from the 'old' materialisation can also be considered once during backward chaining. Proposition 30 shows that, depending on the order in which facts are considered, the FBF algorithm can indeed consider all such rule instances.

Proposition 30. *There exist a positive, single-rule program Π that is recursive w.r.t. any stratification and datasets E, E^-, and $E^+ = \emptyset$ on which the FBF algorithm can consider all rule instances from the 'old' materialisation during backward chaining.*

Proof. Let Π contain rule (148), let $E = \{ B(a_1), B(a_n), T(a_1, a_2), \ldots, T(a_{n-1}, a_n) \}$, and let $E^- = \{B(a_1), B(a_n)\}$. Now assume that the FBF algorithm considers $B(a_n)$ before $B(a_1)$ in line 72. To check whether $B(a_n)$ holds after the update, the algorithm recursively checks each $B(a_i)$ only to find that none of these facts are proved, and in doing so it considers all rule instances from the 'old' materialisation. The algorithm next moves on to $B(a_1) \in E^-$ and, in order to remove all $B(a_i)$, it again considers all rule instances from the 'old' materialisation. Note that the latter step is necessary since the 'old' materialisation could contain other facts that depend on $B(a_i)$ and so need to be removed as well. □

8.4. Instance-increasing updates

We first consider updates where each rule instance that fires in the 'old' materialisation also fires in the 'new' materialisation; we call such updates *instance-increasing*. For example, addition updates with positive programs are instance-increasing.

Definition 31. Let Π be a program with a stratification λ, and let E, E^-, and E^+ be datasets. Then, updating E with E^- and E^+ is *instance-increasing* if $\mathsf{inst}_r\big[I\big] \subseteq \mathsf{inst}_r\big[I'\big]$ holds for each $r \in \Pi$, where $I = \mathsf{mat}(\Pi, E)$ and $I' = \mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$.

Proposition 32 shows that DRed and FBF are optimal, and that their performance does not depend on the chosen stratification. Thus, DRed and FBF can exhibit suboptimal behaviour only if rule instances stop firing due to the update.

Proposition 32. *Both DRed and FBF are optimal on Π, λ, E, E^-, and E^+ for instance-increasing updates.*

Proof. The assumption on the rule instances ensures that no work is done in the Overdelete and DeleteUnproved procedures, and so both DRed and FBF then behave exactly as the standard seminaïve algorithm (cf. Algorithm 1). This algorithm does not repeat inferences, so DRed and FBF consider only the rule instances that fire just in the 'new' materialisation, and are thus optimal for Π, λ, E, E^+, E^-. Moreover, these rule instances are the same for any valid stratification, so DRed and FBF are optimal for any such stratification. □

In contrast, Proposition 33 shows that the counting algorithm is optimal for instance-increasing updates if the program is nonrecursive, but may not be optimal otherwise.

Proposition 33. *The counting algorithm is optimal on Π, λ, E, E^-, and E^+ for instance-increasing updates when Π is nonrecursive. In contrast, there exist datasets E, $E^- = \emptyset$, and E^+, a positive program Π (so the update is instance-increasing), and a stratification λ such that Π is recursive w.r.t. λ and the algorithm is not optimal.*

Proof. Optimality on nonrecursive programs follows from Proposition 27 in Section 8.2, and the lack of optimality on recursive programs follows from Proposition 29 in Section 8.3. □

8.5. Effects of stratification

In Section 3 we observed that a program can admit several stratifications, each inducing different strata, and we also introduced a natural notion of stratification granularity. We now discuss how the choice of a stratification influences the behaviour of our algorithms. In particular, Proposition 34 shows that, for all three algorithms, using a more granular stratification can reduce the total number of rule instances considered. This, in fact, is the usual behaviour that can be expected in practice. For the counting algorithm, using a more granular stratification generally reduces the number of recursive rules, and so the algorithm's behaviour is closer to the optimal behaviour from Section 8.2. For DRed, it prevents the propagation of overdeletion to facts from earlier strata. Finally, for FBF, it prevents proving the facts from previous strata that survive the update.

Proposition 34. *For all three algorithms, there exist a program Π, stratifications λ_0 and λ_1 of Π where λ_1 is more granular than λ_0, and datasets E, E^-, and E^+ such that algorithm considers more rule instances on Π, E, E^-, E^+, and λ_0 than on λ_1.*

Proof. Let Π contain rules (150) for some $n > 2$, let λ_0 be the stratification of Π that assigns all rules to the same stratum, and let λ_1 be the stratification of Π that assigns each rule to the i-th stratum. Clearly, λ_1 is more granular than λ_0.

$$B_i(x) \rightarrow B_{i+1}(x) \qquad 1 \leq i < n \tag{150}$$

We first consider the counting algorithm applied to $E = \{B_1(a)\}$ and $E^+ = \{B_i(a)\}$ for some $1 \leq i < n$. With λ_0, the traces before and after the update are as in Fig. 5, but with $B(a_j)$ substituted by $B_j(a)$ for each $1 \leq j \leq n$. Analogously to the proof of Proposition 29, the counting algorithm considers $2(n-i)$ rule instances. In contrast, Π is nonrecursive w.r.t. λ_1 so, as we discussed in Section 8.2, the counting algorithm is optimal and it considers only the 'new' rule instance $B_i(a) \rightarrow B_{i+1}(a)$.

We next consider the DRed algorithm applied to $E = \{B_1(a), B_2(a)\}$ and $E^- = \{B_1(a)\}$. With λ_0, the algorithm overdeletes and later rederives each fact $B_i(a)$ with $2 \leq i \leq n$, and so it considers $2n-2$ rule instances in total. In contrast, with λ_1, the algorithm deletes and then immediately rederives the fact $B_2(a)$ in the first stratum, which requires two rule instances; moreover, the first stratum does not change so the algorithm does not perform any work in any of the subsequent strata.

Finally, we consider the FBF algorithm applied to $E = \{B_1(a), B_n(a)\}$ and $E^- = \{B_n(a)\}$. With λ_0, all rules are recursive, so FBF consecutively checks all facts $B_n(a), B_{n-1}(a), \ldots, B_1(a)$ using $n-1$ rule instances, and then it proves in line 94 these facts in reverse order using the same $n-1$ rule instances. In contrast, with λ_1, the algorithm performs no work before it

'Old' Trace for λ_0					Facts	'New' Trace for λ_0				
N_1^1	N_2^1	N_3^1	$\cdots$	N_n^1		N_1^1	N_2^1	N_3^1	$\cdots$	N_n^1
1^*					$A(a_2)$					
1					$B(a_1)$	1				
	2^*				$B(a_2)$		1^*			
		1			$B(a_3)$			1		
					$\vdots$					
				1	$B(a_n)$					1

Fig. 6. The traces for Proposition 35 with stratification λ_0.

'Old' Trace for λ_1					Facts	'New' Trace for λ_1					
N_1^1	N_1^2	N_2^2	$\cdots$	N_{n-1}^2		N_1^1	N_1^2	N_2^2	N_3^2	$\cdots$	N_n^2
1^*					$A(a_2)$						
	1				$B(a_1)$		1				
	1^*	1			$B(a_2)$			1			
		1^*			$B(a_3)$				1^*		
					$\vdots$						
				1^*	$B(a_n)$						1^*

Fig. 7. The traces for Proposition 35 with stratification λ_1.

reaches the final stratum, where it checks and consequently proves the fact $B_n(a)$ in one step in line 89 using a nonrecursive rule instance $B_{n-1}(a) \rightarrow B_n(a)$, thus using just two rule instances. □

While Proposition 34 demonstrates typical behaviour, Propositions 35 and 36 show that maximising the number of strata can actually make both the counting algorithm and FBF consider more rule instances. As we discuss in Section 10, we have observed these somewhat counterintuitive effects in the course of our empirical evaluation.

Proposition 35. *There exist a program Π, stratifications λ_0 and λ_1 of Π where λ_1 is more granular than λ_0, and datasets E, E^-, and $E^+ = \emptyset$ on which the counting algorithm considers more rule instances with λ_1 than with λ_0.*

Proof. Let Π contain (151) and (152), let λ_0 be the stratification of Π that maps all atoms to stratum one, let λ_1 map atom $A(x)$ to stratum one and all other atoms to stratum two, let $E = \{A(a_2), B(a_1), T(a_1, a_2), \ldots, T(a_{n-1}, a_n)\}$, and let $E^- = \{A(a_2)\}$.

$$A(x) \rightarrow B(x) \tag{151}$$

$$T(x, y) \wedge B(x) \rightarrow B(y) \tag{152}$$

With λ_0, the traces of the counting algorithm before and after the update are shown in Fig. 6 (without the T-facts as these do not change). Fact $B(a_2)$ has two derivations before the update, but both of them occur in the same iteration. Thus, deleting $A(a_2)$ requires just one rule instance in order to decrease the counter of $B(a_2)$.

With λ_1, the traces before and after the update are shown in Fig. 7. Fact $B(a_2)$ again has two derivations before the update, but these now occur in different iterations. Deleting $A(a_2)$ removes the first of these two derivations; thus, as in the proof of Proposition 29, the derivation of each $B(a_i)$ for $3 \leq i \leq n$ must be moved, which requires $2n - 5$ rule instances in total. □

Proposition 36. *There exist a program Π, stratifications λ_0 and λ_1 of Π where λ_1 is more granular than λ_0, and datasets E, E^-, and $E^+ = \emptyset$ on which that FBF algorithm considers more rule with λ_1 than with λ_0.*

Proof. Let Π contain rules (153) and (154), let $E = \{A(a), B(a), R(a, b_1), \ldots, R(a, b_n)\}$, and let $E^- = \{B(a)\}$. The materialisation of Π and E is E, and the update does not change the materialisation. Now let λ_0 be the stratification of Π that maps all atoms to stratum one, and let λ_1 map atom $R(x, y)$ to stratum one and all other atoms to stratum two.

$$A(x) \wedge R(x, y) \rightarrow B(x) \tag{153}$$

$$B(x) \rightarrow A(x) \tag{154}$$

With λ_0, FBF first checks $B(a)$, and so in line 81 it considers an instance $A(a) \wedge R(a, b_i) \rightarrow B(a)$ of rule (153) for some i with $1 \leq i \leq n$. This leads to checking $A(a)$ and $R(a, b_i)$, both of which are contained in $E \setminus E^-$ and are thus proved in line 88. The algorithm next applies rule (153) to the proved facts in line 94; since $A(a)$ and $R(a, b_i)$ are the only such facts,

only rule instance $A(a) \wedge R(a, b_i) \rightarrow B(a)$ fires and it and proves $B(a)$. Thus, the update is accomplished using one rule instance in backward chaining and one rule instance in forward chaining.

With λ_1, the first stratum contains no rules so the algorithm proceeds to the second stratum and checks $B(a)$. As in the previous case, the algorithm contains some instance of rule (153); this leads to $A(a)$ being checked and ultimately proved in line 88, and so the algorithm applies rule (153) to the proved facts in line 94. In contrast to the case of λ_0, facts $R(a, b_1), \ldots, R(a, b_n)$ are from a previous stratum and so they are all implicitly available; thus, all n instances of rule (153) fire. Consequently, the update requires one rule instance in backward chaining and n rule instances in forward chaining. □

9. Implementation issues

In this section we discuss certain issues that must be addressed before our algorithms can be used in practice. In particular, we first discuss possibilities for representing various sets of facts that our algorithms manipulate, and then we discuss how to apply rules to these sets according to the algebraic constraints specified in the algorithms.

9.1. Representing sets of facts

Most datalog implementations store their data in relational format (i.e., as sets of tuples). Moreover, to deal with large data sets, most implementations use some form of indexes to speed up the retrieval of facts matching an atom from a rule body. Thus, to use our algorithms in practice, we must decide how to represent various (multi)sets of facts that our algorithms maintain (e.g., I, D, A, and so on) in the relational model. We next briefly describe two common approaches and discuss their pros and cons.

A common solution is to introduce, for each predicate P and each set X, a distinct relation P^X that contains all facts from X with predicate P. Thus, to add a fact $P(t_1, \ldots, t_n)$ to X, we insert tuple $(t_1, \ldots, t_n)$ into relation P^X, and we handle fact removal analogously. In fact, related algorithms are often formalised in the literature by maintaining such relations (e.g., the seminaïve algorithm as presented by Abiteboul et al. [1]). The main benefit of this approach is that it allows us to index each P^X as desired, but there are two prominent drawbacks as well. First, each addition/removal of a fact to/from X requires updating all indexes on P^X, which can be very inefficient. Second, applying set operations to our sets of facts (e.g., $(I \setminus D) \cup A$) is problematic: we can recompute such sets every time from scratch, but this is very inefficient; alternatively, we can introduce and maintain a relation for each complex set (e.g., $P^{(I \setminus D) \cup A}$), which can also incur an overhead, particularly if the corresponding relations are indexed. When such an implementation approach is used, it may be beneficial in the DRed and FBF algorithms that set A does not overlap with I and D, as discussed at the end of Section 6, since this reduces the number of complex sets that must be maintained.

Another approach, which we applied in the implementations used in our experiments in Section 10, is to store all facts with predicate P from all sets in a single database relation, and to associate with each fact a bit-mask (e.g., using a distinct column) that reflects membership in various sets. We index the relations as desired, but independently of the bit-mask. Thus, adding/removing a fact to/from a set involves just updating the bit-mask; moreover, to determine whether a fact belongs to a complex set, we just examine the bit-mask using Boolean operations. A drawback of this solution is that indexes are not restricted to specific sets; thus, to retrieve only facts from set X, an index lookup will produce all facts in all sets and we must then skip facts not in X based on the bit-mask. We next argue that, at least in FBF (and thus DRed as well), this is not a significant problem on small updates since, as our experimental results from Section 10.3 suggest, the sets A, B, D, Δ_D, and P are then often much smaller than the set I. In particular, please note that the FBF algorithm evaluates various rule atoms in I, $I \setminus (D \setminus A)$, $(I \setminus D) \cup A)$, $I \cup A$, $I \setminus ((D \cup \Delta_D) \setminus (A \cup B))$, or $P \cup (\mathsf{Out}^{<s} \cap (I \setminus (D \setminus A)))$; thus, if I is much larger than the other sets, the overhead of filtering should be negligible. Moreover, the FBF algorithm also matches a *single* body atom to the sets $D \setminus A$, $A \setminus D$, Δ_D, Δ_A, Δ_P; now assuming these sets are small, it is reasonable to retrieve such matches first by retrieving *all* facts from the corresponding set. Thus, for each of these sets, we just need a single index of all facts in the set so that we can efficiently iterate over all facts in the set when we start matching our rules, which can be achieved using appropriate indexes over the bit-mask.

9.2. Matching the rules

All of our algorithms use $\mathsf{inst}_r\big[\,\cdot\,\big]$ to evaluate a rule over sets of facts, which is nontrivial in practice, particularly if repetition of derivations is to be prevented (cf. Theorems 12, 19, and 25). We next discuss how to implement this operation.

We first consider the simplified case where, given a rule r without negative atoms (i.e., r is of the form $B_1 \wedge \cdots \wedge B_m \rightarrow H$) and sets P and I satisfying $P \subseteq I$, we need to compute $\mathsf{inst}_r\big[I \,\vdots\, P\big]$—that is, we must determine all substitutions matching all body atoms of r in I and at least one body atom in P. We can solve the problem by evaluating the rule body m times where, for each i with $1 \leq i \leq m$, we match atom B_i in P, and we match all remaining atoms in I. This simple solution can still consider the same rule instance twice if atoms B_i and B_j with $i \neq j$ can both be matched in P, but we can address this by matching all atoms before B_i in $I \setminus P$. By putting these two observations together, we can solve our problem by evaluating the conjunctive query (155) for each i with $1 \leq i \leq m$, where the superscript of each atom identifies the set where the atom is matched.

$$B_1^{I \setminus P} \wedge \cdots \wedge B_{i-1}^{I \setminus P} \wedge B_i^P \wedge B_{i+1}^I \wedge \cdots \wedge B_m^I \tag{155}$$

Thus, we require $I \setminus P$ in addition to I and P, which we can realise as discussed in Section 9.1. If sets I and P are implemented using distinct relations, it is common to materialise and maintain $I \setminus P$: set I is typically large, so recomputing it from scratch every time can be expensive. In contrast, if sets are implemented using bit-masks, matching atoms in $I \setminus P$ is easy. Either way, set P is typically small, so it is reasonable to match B_i^P first by iterating over P and thus avoid the need for an index on P.

Now consider the general case where r is of the form $B_1 \wedge \cdots \wedge B_m \wedge \text{not}\, B_{m+1} \wedge \cdots \wedge \text{not}\, B_n$, and consider computing $\mathsf{inst}_r\big[I^+, I^- \vdots P, N\big]$ where $P \subseteq I^+$ and $N \cap I^- = \emptyset$. Analogously to above, we can solve the problem in two steps. First, for each i with $1 \leq i \leq m$, we evaluate query (156)—that is, we consider all cases when a positive atom B_i is matched in P and, to ensure nonrepetition, all atoms before B_i are then matched to $I^+ \setminus P$.

$$B_1^{I^+ \setminus P} \wedge \cdots \wedge B_{i-1}^{I^+ \setminus P} \wedge B_i^P \wedge B_{i+1}^{I^+} \wedge \cdots \wedge B_m^{I^+} \wedge \text{not}\, B_{m+1}^{I^-} \wedge \cdots \wedge \text{not}\, B_n^{I^-} \tag{156}$$

Second, for each i with $m+1 \leq i \leq n$, we evaluate query (157)—that is, we consider all cases where a negative atom B_i is matched in N; note that definition (65) requires one atom to be *inside* N, which is why B_i occurs positively in (157). To ensure nonrepetition, all positive atoms are matched in $I^+ \setminus P$ since they occur before B_i; however, $\text{not}\, B_j$ with $m < j < i$ are matched in $I^- \cup N$ because they should not satisfy the property that holds for B_i and should thus be *outside* N.

$$B_1^{I^+ \setminus P} \wedge \cdots \wedge B_m^{I^+ \setminus P} \wedge \text{not}\, B_{m+1}^{I^- \cup N} \wedge \cdots \wedge \text{not}\, B_{i-1}^{I^- \cup N} \wedge B_i^N \wedge \text{not}\, B_{i+1}^{I^-} \wedge \cdots \wedge \text{not}\, B_n^{I^-} \tag{157}$$

Nonrepetition is not essential for the correctness of DRed and FBF: we are free to simplify (156) and (157) by using I^+ and I^- instead of $I^+ \setminus P$ and $I^- \cup N$, respectively. Despite occasionally repeating inferences, depending on the implementation details such an implementation can actually be more efficient since it does not need to compute and maintain $I^+ \setminus P$ and $I^- \cup N$.

The counting algorithm, however, keeps track of the number of derivations for each fact and so nonrepetition of inferences is essential for correctness. In addition, the algorithm also uses $\mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1 \vdots P_2, N_2\big]$, where for each match of r there must be a positive body atom in P_1 or a negative body atom in N_1, as well as a positive body atom in P_2 or a negative body atom in N_2. In principle, we can use the same approach as above, but we must consider a variant of the rule body for each $1 \leq i, i' \leq n$. Thus, index i tracks the atom matched to P_1 or N_1, so we match each atom before i in $I^+ \setminus P_1$ or $I^- \cup N_1$; analogously, index i' tracks the atom matched to P_2 or N_2, so we match each atom before i' in $I^+ \setminus P_2$ or $I^- \cup N_2$. Finally, each atom can be restricted to one set only, so we must conjoin these conditions; for example, all positive atoms before i and i' are evaluated in $I^+ \setminus (P_1 \cup P_2)$, atoms between i and i' (assuming $i < i'$) are matched in $I^+ \setminus P_2$, and so on.

While correct, the approach outlined in the previous paragraph is impractical since it requires evaluating n^2 rules. A more pragmatic solution is to compute $\mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1\big]$ or $\mathsf{inst}_r\big[I^+, I^- \vdots P_2, N_2\big]$ and then filter out the rule instances not satisfying the other condition. For example, if $P_1 \cup N_1$ is smaller than $P_2 \cup N_2$, we first compute $\mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1\big]$, and then we remove each rule instance $r' \in \mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1\big]$ such that $\mathsf{b}^+(r') \cap P_2 = \mathsf{b}^-(r') \cap N_2 = \emptyset$. Depending on the implementation details, filtering can often be performed as rules are matched, which is beneficial for performance.

It is well known that the efficiency of evaluating conjunctive queries dramatically depends on the chosen query plan, and queries (156) and (157) are no exception. We can use any of the known query planning approaches to optimise these queries, but incremental computation introduces a complication. To make our discussion concrete, we assume that we have an index, which, given an atom A and a set of facts I, can quickly identify each substitution σ such that $A\sigma \in I$ holds, and that we evaluate queries using left-to-right index nested loop joins—that is, given a query of the form $B_1 \wedge \cdots \wedge B_n$, we identify each substitution σ_1 such that $B_1 \in I$, then for each σ_1 we identify each substitution σ_2 such that $B_2\sigma_1\sigma_2 \in I$, and so on. Thus, query planning amounts to reordering the query atoms in a way that minimises the overall work. The following example illustrates certain problems specific to maintenance algorithms, and it also shows that the number of rule instances (as we suggested in Section 8.1) may not always completely reflect the overall work. As we discuss in Section 10, these problems can sometimes be observed in practice.

Example 37. Let Π contain rule r as specified in (158), and let E be as specified in (159) for some n. The materialisation of Π w.r.t. E derives $A(a)$.

$$R(x_1, x_2) \wedge S(x_2, x_3) \wedge T(x_3, x_4) \rightarrow A(x_1) \tag{158}$$

$$E = \{R(a, b_1), S(b_1, c), \ldots, S(b_n, c), T(c, d)\} \tag{159}$$

Materialising Π requires evaluating the body of rule (158) as a query, for which we can obtain an efficient query plan statically (i.e., before computing the materialisation) using standard query planning techniques. In particular, if we order the body atoms as shown in the rule, left-to-right evaluation matches each body atom to just one fact.

Now assume that we wish to update the materialisation with $E^- = \{T(c, d)\}$. Each of our algorithms will need to compute $\mathsf{inst}_r\big[I, \Delta\big]$ for $\Delta = \{T(c, d)\}$, which amounts to evaluating query $R(x_1, x_2)^I \wedge S(x_2, x_3)^I \wedge T(x_3, x_4)^\Delta$. Applying the standard planning techniques statically is difficult since Δ is unknown at planning time; however, it is reasonable to assume that Δ

will be small and thus reorder the query as $T(x_3, x_4)^\Delta \wedge S(x_2, x_3)^I \wedge R(x_1, x_2)^I$. But then, left-to-right evaluation matches $T(x_3, x_4)^\Delta$ to $T(c, d)$, and so $S(x_2, c)^I$ matches to n atoms of which only one leads to a query match. Hence, even though these issues do not affect the number of rule instances considered, rule matching during materialisation maintenance can be less efficient than during initial materialisation due to suboptimal plans. Finally, it might be possible to compute a plan dynamically (i.e., when Δ is available), but this can easily become another source of overhead.

9.3. Implementing backward chaining

We now discuss the implementation of backward chaining in DRed (line 42 of Algorithm 4) and FBF (lines 64, 81, and 88 of Algorithm 5). Given a fact F, these steps must identify each instance $r' \in \mathsf{inst}_r\big[I^+, I^-\big]$ of a rule $r \in \Pi$ such that $\mathsf{h}(r') = F$.

To this end, we first identify rule r and substitution σ_1 such that $\mathsf{h}(r)\sigma_1 = F$. An indexing scheme for the rules in Π is often useful. For example, using a hash table we can associate each predicate P with the set of rules of Π that contain P in the head; then, given a fact F, the hash table provides us with the candidate rules. Moreover, term indexing techniques [67] from first-order theorem proving can further reduce the number of candidates in cases when the head atoms of Π contain constants.

For each rule r of the form (1) and substitution σ_1 identified in the previous step, we next evaluate conjunctive query (160); hence, for each substitution σ_2 matching the atoms (160) to I^+ and I^-, we obtain a distinct rule instance $r\sigma_1\sigma_2 \in \mathsf{inst}_r\big[I^+, I^-\big]$.

$$B_1^{I^+}\sigma_1 \wedge \cdots \wedge B_m^{I^+}\sigma_1 \wedge \text{not}\, B_{m+1}^{I^-}\sigma_1 \wedge \cdots \wedge \text{not}\, B_n^{I^-}\sigma_1 \tag{160}$$

Substitution σ_1 typically instantiates at least one atom of query (160), so the query can usually be evaluated efficiently. However, Example 38 shows that backward chaining can sometimes become a considerable source of overhead: even though Theorems 19 and 25 guarantee that backward chaining considers only rule instances from the 'old' materialisation, identifying the relevant instances can be much more costly. Hence, this uncovers another case when our metric from Section 8.1 does not fully reflect the overall work. As we discuss in Section 10, this effect can sometimes be observed in practice. In our discussion, we again assume that queries are evaluated left-to-right, as outlined in Section 9.2.

Example 38. Let Π be the program containing rule (161), and let E be as specified in (162). The materialisation of Π and E derives fact $S(b, b)$, and facts $S(b, c_i)$, $S(c_i, b)$, and $S(c_i, c_i)$ for $1 \le i \le n$.

$$R(x, y_1) \wedge R(x, y_2) \rightarrow S(y_1, y_2) \tag{161}$$

$$E = \{R(a_i, b), R(a_i, c_i) \mid 1 \le i \le n\} \tag{162}$$

The body of rule (161) can be efficiently evaluated during materialisation using left-to-right evaluation: if we first match $R(x, y_1)$ to either $R(a_i, b)$ or $R(a_i, c_i)$, atom $R(x, y_2)$ is instantiated as $R(a_i, y_2)$, and so our index finds the matching atoms $R(a_i, b)$ and $R(a_i, c_i)$. Since the body of (161) is symmetric, reordering the body atoms produces the same behaviour.

Now assume that we wish to update the materialisation with $E^- = \{R(a_i, c_i) \mid 1 \le i \le n\}$; thus, $S(b, b)$ and all $S(b, c_i)$, $S(c_i, b)$, and $S(c_i, c_i)$ must be deleted. Both DRed and FBF can identify these facts efficiently as in previous paragraph. However, both algorithms also try to reprove all of these facts using backward chaining, which can be problematic. In particular, for $S(c_i, c_i)$, we obtain query $R(x, c_i) \wedge R(x, c_i)$, which can be evaluated efficiently using a left-to-right plan. However, for $S(b, b)$, we obtain query $R(x, b) \wedge R(x, b)$; for $S(b, c_i)$, we obtain query $R(x, b) \wedge R(x, c_i)$; and for $S(c_i, b)$, we obtain query $R(x, c_i) \wedge R(x, b)$. All of these queries contain $R(x, b)$ and, if we order the body atoms of (161) statically, we have n queries where $R(x, b)$ is evaluated first; however, evaluating $R(x, b)$ using an index produces n candidate matches $R(a_j, b)$, but we obtain a rule instance only for $j = i$. Thus, we match $R(x, b)$ to at least $n(n-1)$ facts for which matching the second query atom does not produce a rule instance from the 'old' materialisation, which can be quite costly. This problem can be addressed by trying to choose a query plan dynamically based on the fact being proved, but this can easily become another source of overhead.

10. Experimental evaluation

As we have explained in Section 8, no materialisation maintenance algorithm is universally best: inputs exist on which one algorithm exhibits good performance, whereas others perform poorly. Hence, we conducted an extensive empirical comparison with two main objectives. First, we investigate the behaviour of our algorithms on small updates (i.e., when the number of facts being updated is orders of magnitude smaller than the number of explicit facts). Since the number of possible inputs to an algorithm is very large, one may wonder whether an algorithm's behaviour depends significantly on which facts are updated. We investigate this question statistically and thus call these *robustness experiments*. Second, we investigate how an algorithm's performance depends on the number of updated facts, and we also identify break-even points where rematerialisation 'from scratch' outperforms our incremental update algorithms. We call these *scalability experiments*.

In this section we present the results of our evaluation and discuss whether they confirm the behaviour expected from Section 8. Towards this goal, in Section 10.1 we first describe our test datasets, and in Section 10.2 we described our

test system and the experimental setup. Next, in Section 10.3 we present the results our robustness experiments, and in Section 10.4 we present the results of our scalability experiments. Finally, in Section 10.5 we summarise our findings. All systems, all datasets, and programs used in our tests are available online.[9] For the sake of brevity, we present here only a selection of the results that we consider interesting, but all of our results are given in an appendix available from the same Web page.

10.1. Test data

We are not aware of freely available repositories of 'native' datalog datasets. In contrast, a large number of large and complex datasets structured using complex OWL ontologies are available on the Semantic Web in the RDF format. Moreover, techniques are readily available for translating OWL ontologies into datalog programs of varying degrees of complexity, which allows us to study the behaviour of our algorithms on a range of programs. Therefore, in our evaluation we used the following six datasets, each comprising a set of RDF triples and an OWL 2 DL ontology. All but the last one are derived from real-world applications. Our test data is quite heterogeneous, and we see no reason to believe that its origin has affected our experimental outcomes.

- ChEMBL [68] contains information about the chemical properties of bioactive compounds.
- Claros is catalogue of archeological artefacts.[10]
- DBpedia [69] contains structured data extracted from Wikipedia.
- Reactome [70] describes biochemical pathways of proteins, drugs, and other agents.
- UniProt [71] combines several datasets describing protein sequences and their functional information.
- UOBM [72] is a synthetic dataset that extends the well known LUBM [73] benchmark.

We converted the RDF triples into facts as outlined in Section 3. Moreover, we converted each OWL 2 DL ontology into a set of first-order formulae of the form

$$B_1 \wedge \cdots \wedge B_m \rightarrow \left[\exists \vec{y}_1 . H_1^1 \wedge \cdots \wedge H_{k_1}^1\right] \vee \cdots \vee \left[\exists \vec{y}_\ell . H_1^\ell \wedge \cdots \wedge H_{k_\ell}^\ell\right] \tag{163}$$

using the well-known correspondences between OWL 2 DL and first-order logic [74]. Some of the resulting formulae contained disjunctions and/or existential quantification, which are not supported in datalog. To obtain programs of varying complexity, we converted these formulae into programs using one of the following transformations.

- The *lower bound* (L) program is obtained by simply deleting formulae that do not correspond to datalog rules—that is, that contain either disjunction or existential quantification. This transformation is analogous to the Description Logic Programs (DLP) transformation by Grosof et al. [75], and it is sound but incomplete: the lower bound program captures only, but not all consequences of the ontology. Semantic Web systems typically use this program for reasoning over Semantic Web data, so the lower bound program represents the typical workload for an important class of datalog applications.
- The *rewritten* (R) program is another sound but incomplete approximation developed by Kaminski et al. [76]. Roughly speaking, their transformation aims to preserve as many consequences as possible by encoding away existential quantifiers using the transformation by Hustadt et al. [77], transforming the result into linear disjunctive datalog to the extent possible, and then further transforming away disjunctions. Programs obtained in this way typically preserve more consequences than the lower bound programs and they tend to be complex and highly recursive.
- The *upper bound* (U) program is obtained by replacing in each formula all existentially quantified variables with fresh constants and converting all disjunctions into conjunctions. This transformation is complete but unsound: the resulting program captures all consequences of the ontology, but it may also derive additional facts. This transformation was used by Zhou et al. [78] as part of a more complex approach to query answering over OWL 2 DL ontologies. Rules obtained in this way also tend to be complex, and they contain constants in the rule heads.
- For Claros, we obtained the *lower bound extended* (LE) program by manually axiomatising the 'refers to the same topic' predicate. The materialisation of this program contains cliques of items that refer to the same topic, which turned out to be particularly difficult for our algorithms.

Some of the OWL 2 DL ontologies contain axioms with the *owl*:*sameAs* property, which is treated in RDF as equality. The semantics of equality presents many new challenges to the materialisation maintenance problem [79], and studying these is out of scope of this paper. Hence, in our evaluation we simply disregarded the semantics of equality and treated *owl*:*sameAs* as just another ordinary property without any predetermined meaning.

We call each combination of a (transformed) program and a dataset a *test instance* (or just *instance*), which we name by combining the name of the ontology and the transformation used. Not all transformations are applicable to or interesting

[9] http://www.cs.ox.ac.uk/isg/tools/RDFox/2017/RDFox-Journal/index.html.
[10] http://www.clarosnet.org/.

Table 1
Statistics about datasets and datalog programs.

	ChEMBL		Claros		DBpedia	Reactome			UniProt		UOBM	
Facts	289.2 M		18.8 M		386.7 M	12.5 M			123.1 M		254.8 M	
Variant	L	R	L	LE	L	L	R	U	L	R	L	R
Rules	259	2,265	1,310	1,337	5,936	541	21,385	842	451	12,018	210	2,379
Mat. Facts	323.4 M	1.0 G	73.8 M	533.3 M	643.8 M	19.8 M	369.4 M	45.6 M	174.7 M	556.7 M	426.2 M	1.3 G

Table 2
Number of rules (total and per stratum) for test programs.

Name	Number of rules (recursive/total)											
	All levels	Per stratification level										
ChEMBL_L	0/259	0/172	0/35	0/38	0/14							
ChEMBL_R	499/2,265	0/308	0/35	499/1,922								
Claros_L	300/1,310	240/240	46/364	10/227	4/210	0/89	0/32	0/38	0/12	0/12	0/44	0/42
Claros_{LE}	306/1,337	240/240	47/365	10/229	4/213	2/96	2/41	0/38	0/13	1/16	0/44	0/42
DBpedia_L	828/5,936	322/322	318/2,117	80/1,401	64/523	19/357	19/899	6/293	0/24			
Reactome_L	17/541	4/4	7/99	4/61	2/48	0/90	0/239					
Reactome_R	21,385/21,385	21,385/21,385										
Reactome_U	28/842	3/3	12/80	8/139	3/158	2/81	0/37	0/90	0/30	0/224		
UniProt_L	18/451	8/8	10/180	0/126	0/16	0/20	0/101					
UniProt_R	2,706/12,018	8/8	213/988	0/429	0/196	2,485/10,397						
UOBM-01K_L	49/210	6/6	6/72	2/40	0/17	35/75						
UOBM-01K_R	2,215/2,379	4/4	70/142	2/36	0/17	2,139/2,177	0/3					

for all ontologies, so in our evaluation we used the 12 test instances shown in Table 1. The table shows the number of facts in each dataset before materialisation, and, for each program, the total number of rules and the number of facts after materialisation. As one can see from the table, our datasets are large, and materialisation can increase the number of facts by a factor of almost 30.

As suggested by Example 17, updating the materialisation by strata can be much more efficient than processing the entire program at once. In order to see the effects this optimisation has on the performance, we ran our algorithms with all rules in a single stratum and with a granular stratification that groups rules by levels—that is, inductively starting from level 1, level i is the smallest nonempty set of rules such that each rule in level i depends on another rule in level i or $i-1$. Please note that stratification by levels is sufficient to obtain the maximum performance improvement in all of our algorithms. Table 2 shows the number of recursive and all rules in total and per stratification level for each program. As one can see, our programs are relatively large, comprising between hundreds and tens of thousands of rules arranged in at most 11 stratification levels. Finally, recursive rules constitute a substantial portion of seven of the test programs, and in fact all rules are recursive in Reactome_R.

OWL 2 DL ontologies are first-order theories so our programs do not contain negation. However, we see no reason to believe that this affects the outcomes of our evaluation since negative atoms are evaluated analogously to positive atoms in all three algorithms. For example, in line 46 of the overdeletion step of DRed, a positive atom must be matched to $D \setminus A$, whereas a negative atom must be matched to $A \setminus D$; hence, achieving the former is equal in difficulty as achieving the latter.

We used two sampling methods to obtain sets of facts to delete/add. Both of them are given an explicit dataset E, a random seed, and a size k, and they return a subset $E' \subseteq E$ with exactly k facts. The seed is used to initialise a pseudo-random number generator, so E' is uniquely determined by E, the seed, and the size k. Moreover, both methods are *stable* in the following sense: given the same seed and sizes k_1 and k_2, the subsets E'_1 and E'_2 of E produced for k_1 and k_2 satisfy $E'_1 \subseteq E'_2$ whenever $k_1 \leq k_2$ holds. In other words, if we keep the seed fixed but increase the size, we obtain a monotonically increasing sequence of subsets of E. In Sections 10.3 and 10.4 we describe how we used these subsets to obtain inputs for the robustness and scalability tests, respectively. We used the following two sampling methods.

- *Uniform sampling* identifies a subset of E of size k by randomly selecting facts from E without replacement.
- *Random walk sampling* identifies a subset of E by viewing E as a graph and making random transitions along the graph edges, and by restarting the process at random whenever a dead end is reached.

Uniform sampling is unbiased and so it tends to select unrelated facts, whereas random walk sampling tends to select assertions about the same objects. In this way, our experiments simulate two natural kinds of workload.

To summarise, each test run involves (i) a test instance from Table 1 consisting of a program Π and an explicit dataset E, (ii) a sampling method (i.e., random walk or uniform sampling), a sampling seed, and a sampling size, (iii) a stratification type, which either puts all rules into a single stratum (written 'N') or arranges rules by levels (written 'Y'), and (iv) a materialisation maintenance algorithm, which can be counting (written 'CNT'), DRed, or FBF. In some tests, we also consider rematerialisation 'from scratch' as a maintenance algorithm, and we distinguish the standard seminaïve approach (cf. Algorithm 1, written 'Remat') from the counting seminaïve variant (cf. Algorithm 2, written 'CRemat').

10.2. Test system and test setup

We have implemented the DRed and the FBF algorithm in the publicly available RDFox[11] system—a state of the art datalog-based RDF management engine. As in Section 8, our FBF implementation never aborts backward chaining. We have also implemented the counting algorithm in a separate branch of RDFox that associates counters with facts. Both systems have been written in C++. Although RDFox can parallelise materialisation and some maintenance steps as well, we conducted all tests on a single thread in order to investigate the performance of the basic algorithms; parallelising maintenance algorithms and studying the effects of this on the performance is beyond of scope of this paper.

We carried out all of our experiments on a Dell PowerEdge R730 server with 512 GB of RAM and two Intel Xeon E5-2640 2.6 GHz processors, each of which has eight physical and 16 virtual cores. The server runs Fedora 24, kernel 4.5.7. We cleared the system caches before each test using the following commands:

```
sync; echo 1 > /proc/sys/vm/drop_caches
sync; echo 2 > /proc/sys/vm/drop_caches
sync; echo 3 > /proc/sys/vm/drop_caches
```

Combined with the fact that we had sole access to the server for the duration of all tests, this ensured consistent performance of each test, so we conducted each test only once and recorded its wall-clock time.

To obtain a performance measure that is somewhat independent from implementation details, we also recorded the number derivations (i.e., considered rule instances) needed to perform an update; we did this in a separate run in order to not affect our time measurements. For the counting algorithm, we recorded the *deleted* (lines 23 and 36) and the *added* (lines 24 and 37) derivations. For DRed, we recorded the derivations in overdeletion (lines 46 and 50), one-step rederivation (line 42), and insertion/rederivation (lines 53 and 58). Finally, for FBF, we recorded the derivations in deletion propagation (lines 68 and 75), backward chaining and one-step rederivation (lines 64 and 81), forward chaining (lines 88 and 94), and insertion (line 53 and 58).

Running times and the numbers of derivations vary greatly between test instances, and so they cannot be used directly to compare algorithms across test instances. To compare algorithms across test instances, we introduce a *score* measure as follows. Assume that we take three measurements m_{CNT}, m_{DRed}, and m_{FBF} for our three algorithms on some test instance. If a smaller value indicates better performance (e.g., running times or numbers of derivations), then the score of algorithm $X \in \{\mathrm{CNT}, \mathrm{DRed}, \mathrm{FBF}\}$ on this test instance is defined as $m_X / \min\{m_{\mathrm{CNT}}, m_{\mathrm{DRed}}, m_{\mathrm{FBF}}\}$. In contrast, if a larger value indicates better performance (e.g., the number of explicit facts that can be deleted before rematerialisation becomes more efficient), then the score of algorithm X is defined as $\max\{m_{\mathrm{CNT}}, m_{\mathrm{DRed}}, m_{\mathrm{FBF}}\}/m_X$. Either way, a score of 1 always reflects the best-performing algorithm on a given test instance, and the average and median scores over all test instances illustrate the relative performance of our algorithms.

For brevity, in this paper we present only the test results which illustrate important features of the performance of the three materialisation maintenance algorithms. The complete results of all measurements are available online.[9]

10.3. Robustness experiments

The aim of our robustness experiments was to test our algorithms on small updates, and to determine whether and how the performance of our algorithms depends on the exact facts being updated. Small updates are very common in practice and so a practical materialisation maintenance algorithm should handle them efficiently.

Our datasets contain tens or hundreds of millions of facts, so in our robustness experiments we considered updates involving 1,000 facts—a small fraction of the explicit dataset. As we explained in Section 4, addition of facts can be efficiently handled using the seminaïve algorithm (on positive programs) and is thus computationally much simpler than deletion, which necessitates identifying alternative derivations of deleted facts. Therefore, given a program Π, an explicit dataset E, and a subset $E' \subseteq E$ induced by a sampling method, we conducted the following two kinds of experiments.

- In the *robustness deletion experiments*, we deleted $E^- = E'$ from $\mathsf{mat}(\Pi, E)$.
- In the *robustness addition experiments*, we added $E^+ = E'$ to $\mathsf{mat}(\Pi, E \setminus E')$.

As we discussed in Section 8, the performance of each algorithm can considerably depend on which facts are being updated, and so we would ideally determine the average running time μ_t over all possible subsets of E of size 1,000. Since this is clearly infeasible, we use the sampling theory [80, Chapter 7] to estimate μ_t and determine a confidence interval. Specifically, we choose uniformly at random n seeds that, for the chosen sampling method, identify n subsets $E'_1, \ldots, E'_n$ of E. We then run the test in question for each E'_i and record the corresponding running times $t_1, \ldots, t_n$, and finally we compute the average time $\bar{t}$ and the standard deviation S of the running times as

[11] http://www.cs.ox.ac.uk/isg/tools/RDFox/.

$$\bar{t} = \frac{\sum_{i=1}^{n} t_i}{n} \qquad \text{and} \qquad S = \sqrt{\frac{\sum_{i=1}^{n} (t_i - \bar{t})^2}{n-1}}. \tag{164}$$

The *central limit theorem* [80, Chapter 5.3] then ensures that the distribution of the average $\bar{t}$ converges to a normal distribution as n increases, which allows us to compute the 95% *z-confidence interval* for the average as

$$\bar{t} \pm \frac{1.96}{\sqrt{n}} S. \tag{165}$$

A frequentist interpretation of the z-confidence interval is as follows: if we repeatedly take samples of size n and for each sample we compute the average and the standard deviation as in (164), then, in at least 95% of the cases, the actual average time μ_t will be inside the z-confidence intervals computed as in (165). Please note that, since we do not know how running times are distributed, this statement holds only in the limit as the sample size n converges to infinity. In practice, we need a finite sample size n, and, without any prior assumptions, it is common practice to use $n = 30$ [80, page 218]: number n occurs in the denominator of (165), so larger values of n reduce the size of the confidence interval, and thus of the uncertainty as well; moreover, for $n = 30$, we have $1.96/\sqrt{n} = 0.357$, which produces narrow confidence intervals when the standard deviation is of the same order of magnitude as the average. In our robustness experiments we thus used the sample size of $n = 30$. We estimated the average number of derivations and computed its 95% z-confidence interval in an analogous way.

10.3.1. Robustness deletion experiments

Table 3 shows the results of our robustness deletion experiments where the subset of 1,000 facts was selected using random sampling. For each test instance, the table shows the running time and the number of derivations with their z-confidence intervals for the unstratified and stratified variants of the program. Table 4 shows the breakdown of the numbers of derivations—that is, the number of derivations performed in various phases of the algorithms, as discussed in Section 10.2. At the bottom, Table 3 shows the average running time and the average number of derivations taken over all inputs without stratification ('N'), with stratification ('Y'), or regardless of stratification ('All'). The table also shows the ratio of the running time without and with stratification averaged over all inputs, and it analogously shows the average ratio of the number of derivations. For brevity, Table 3 does not show the individual scores for the average running times and the numbers of derivations, but only shows the average and the median scores at the bottom. Finally, to analyse the factors determining the performance of our algorithms, Table 3 also shows the *Pearson correlation coefficient* [80, page 406] between the running time and the number of derivations, and the running time and the materialisation size. This coefficient indicates the strength of a linear relationship between two variables: +1 is total positive linear correlation, 0 is no linear correlation, and −1 is total negative linear correlation. We next discuss our results and identify strengths and weaknesses of our three algorithms.

Sampling accuracy In all cases other than Claros_{LE}, the widths of the confidence intervals are at least an order of magnitude smaller than the corresponding values for both running time and the number of derivations. Thus, the values presented in Table 3 seem to approximate well the average running time and the numbers of derivations across all inputs—that is, the sample size of 30 seems effective and increasing it further would most likely just reduce the widths of the confidence intervals by reducing the denominator of (165). On Claros_{LE}, however, the confidence intervals for the running times of DRed and FBF, and for the numbers of derivations of all three algorithms are of the same order of magnitude as the corresponding values. Thus, the performance of DRed and FBF varies greatly depending on which subset E' of 1,000 facts of E we select, for reasons we discuss shortly. Consequently, our algorithms exhibit robust performance in all cases, apart from DRed and FBF on Claros_{LE}.

Number of derivations The counting algorithm requires the least number of derivations, followed by FBF and then DRed. This is in line with our discussion from Section 8: by keeping track of the number of derivations, the counting algorithm can efficiently determine when a fact should be deleted; moreover, the backward chaining in FBF seems more effective than overdeletion in DRed. As we discuss next, however, the number of derivations does not directly determine the performance of all algorithms.

Running times As one can see from Table 3, DRed and FBF are faster than counting, often by several orders of magnitude, in all cases except for Claros_{LE}, and for Reactome_R and DRed. Moreover, DRed and FBF perform very badly on Claros_{LE}, which skews the average running times. However, the scores for the running times (cf. Section 10.2) provide us with a relative measure of the performance of the three algorithms, and by comparing the average and the median scores we can see that FBF is fastest on average across all test instances, followed by DRed and then counting. Indeed, DRed is faster than FBF only on ChEMBL_L, Claros_L, and DBPedia_L with stratification, but this difference is insignificant since the running times are in the order of tens of milliseconds. In contrast, FBF outperforms DRed by at least an order of magnitude on ChEMBL_R, Claros_{LE}, Reactome_R, and UOBM-01K_R. As one can see in Table 2, these programs contain a significant number of recursive rules, which, as we suggested in Section 8, makes maintenance of deletions difficult. However, the backward chaining of FBF seems generally very effective, and it makes the performance of FBF more consistent than that of DRed, as one can see from the average and median scores.

Table 3
Deleting 1,000 facts selected by uniform sampling.

Instance	Strat	Time (s)			Derivations		
		CNT	DRed	FBF	CNT	DRed	FBF
ChEMBL$_L$	N	11.98 ± 0.23	0.02 ± 0.00	0.02 ± 0.00	1.14 k ± 6.62	3.65 k ± 28.88	2.85 k ± 28.41
	Y	11.61 ± 0.23	0.01 ± 0.00	0.02 ± 0.00	1.05 k ± 8.72	1.29 k ± 11.72	1.29 k ± 11.72
ChEMBL$_R$	N	37.79 ± 1.14	9.36 ± 0.16	0.46 ± 0.02	6.07 k ± 54.00	113.24 M ± 1.50 M	44.54 k ± 370.85
	Y	37.68 ± 0.91	9.22 ± 0.15	0.95 ± 0.03	6.07 k ± 54.00	113.24 M ± 1.50 M	14.41 M ± 299.82 k
Claros$_L$	N	3.17 ± 0.06	0.36 ± 0.32	0.06 ± 0.00	7.77 k ± 92.78	35.13 k ± 20.36 k	15.66 k ± 167.07
	Y	2.84 ± 0.04	0.04 ± 0.00	0.05 ± 0.00	5.20 k ± 85.04	6.32 k ± 91.06	7.66 k ± 112.22
Claros$_{LE}$	N	26.15 ± 0.81	5.45 k ± 1.29 k	582.89 ± 458.10	2.24 M ± 1.90 M	16.38 G ± 3.90 G	1.96 G ± 1.60 G
	Y	26.31 ± 0.80	1.81 k ± 997.80	472.54 ± 374.46	2.22 M ± 1.91 M	7.02 G ± 3.90 G	1.99 G ± 1.63 G
DBpedia$_L$	N	24.41 ± 0.39	0.02 ± 0.00	0.03 ± 0.00	915.00 ± 16.47	1.46 k ± 44.28	1.40 k ± 37.68
	Y	24.73 ± 0.49	0.02 ± 0.00	0.03 ± 0.00	845.27 ± 18.72	1.27 k ± 34.36	1.29 k ± 33.78
Reactome$_L$	N	0.86 ± 0.02	0.07 ± 0.00	0.06 ± 0.00	2.22 k ± 41.63	10.19 k ± 229.79	5.38 k ± 77.98
	Y	0.76 ± 0.02	0.05 ± 0.00	0.05 ± 0.00	1.85 k ± 41.34	2.80 k ± 50.29	2.93 k ± 66.68
Reactome$_R$	N	15.84 ± 0.11	40.37 ± 2.22	0.81 ± 0.02	170.36 k ± 11.08 k	77.24 M ± 2.51 M	325.82 k ± 13.97 k
	Y	15.71 ± 0.10	40.63 ± 2.19	0.85 ± 0.02	170.36 k ± 11.08 k	77.24 M ± 2.51 M	325.82 k ± 13.97 k
Reactome$_U$	N	1.91 ± 0.04	2.80 ± 0.20	0.75 ± 0.08	7.23 k ± 135.36	46.10 k ± 875.24	18.13 k ± 248.22
	Y	1.71 ± 0.04	1.02 ± 0.12	0.98 ± 0.11	5.53 k ± 110.88	8.81 k ± 144.12	8.20 k ± 143.49
UniProt$_L$	N	6.47 ± 0.11	0.02 ± 0.00	0.02 ± 0.00	1.62 k ± 12.86	3.45 k ± 58.73	3.66 k ± 30.68
	Y	6.30 ± 0.06	0.02 ± 0.00	0.02 ± 0.00	1.35 k ± 9.46	2.45 k ± 52.47	2.15 k ± 18.89
UniProt$_R$	N	22.00 ± 0.15	8.10 ± 0.44	1.78 ± 0.04	6.71 k ± 119.71	35.78 M ± 502.85 k	23.34 k ± 920.14
	Y	21.94 ± 0.17	8.97 ± 0.70	4.09 ± 0.20	6.75 k ± 119.76	35.78 M ± 502.89 k	17.53 M ± 228.77 k
UOBM-01K$_L$	N	21.18 ± 0.19	0.64 ± 0.03	0.05 ± 0.00	3.14 k ± 45.46	372.25 k ± 14.50 k	7.84 k ± 132.50
	Y	16.78 ± 0.06	0.04 ± 0.00	0.04 ± 0.00	3.11 k ± 45.69	13.08 k ± 164.06	7.30 k ± 120.61
UOBM-01K$_R$	N	60.11 ± 0.64	15.36 ± 0.81	0.34 ± 0.00	10.84 k ± 382.39	11.93 M ± 470.97 k	114.01 k ± 751.83
	Y	50.54 ± 0.53	0.50 ± 0.01	0.21 ± 0.00	10.37 k ± 177.07	326.35 k ± 2.31 k	148.36 k ± 969.07
Average value:	N	19.32	460.65	48.94	205.03 k	1.39 G	163.60 M
	Y	18.07	155.80	39.99	202.35 k	603.84 M	168.38 M
	All	18.70	308.23	44.46	203.69 k	994.44 M	165.99 M
Average ratio unstratified/ stratified:		1.07	5.85	1.02	1.12	7.52	1.24
Average score:	N	247.86	29.50	2.85	1.00	2,747.83	76.19
	Y	269.97	11.47	2.56	1.00	2,300.81	491.16
	All	258.92	20.49	2.70	1.00	2,524.32	283.67
Median score:	N	68.55	5.34	1.00	1.00	62.46	2.49
	Y	53.90	1.02	1.15	1.00	3.01	1.75
	All	61.22	2.29	1.00	1.00	5.48	2.38
Correlation of time and # derivations:	N	0.12	1.00	1.00			
	Y	0.17	1.00	1.00			
	All	0.15	1.00	0.99			
Correlation of time and mat. size:	N	0.98	0.07	0.06			
	Y	0.99	0.07	0.07			
	All	0.99	0.06	0.06			

Performance of counting Although the counting algorithm generally made the least number of derivations, it was slowest overall, with an average score of 258.92 for the running time. After examining the performance of our system in a profiler, we noticed that most of the running time tends to be spent in reconstructing the sets I_o and I_n in lines 27, 29, and 32 of Algorithm 3. Although these steps do not evaluate rules, they manipulate large sets of facts; thus, even though the number of derivations is much lower than for DRed (average score of 2,524.32) or FBF (average score of 283.67), the reconstruction of I_o and I_n incurs a fixed overhead proportional to the size of the materialisation, which dominates the performance of the counting algorithm on small updates. The experiments with Claros$_{LE}$ confirm this: for reasons we discuss shortly, the number of derivations with DRed and FBF is much larger than with the counting algorithm, and so the former algorithms are much slower than counting despite the fixed overhead of reconstructing I_o and I_n.

Performance factors As one can see from Table 3, the running times of DRed and FBF are highly correlated with the numbers of derivations: the algorithms exhibit an almost perfect Pearson correlation coefficient of 1.00 and 0.99, respectively.

Table 4
Breakdown of the numbers of derivations in robustness deletion experiments reported in Table 3.

Instance	Strat	CNT		DRed			FBF		
		add	del	fwd	bwd	del	fwd	bwd	del
ChEMBL_L	N	48.40	1.10 k	1.03 k	543.50	2.08 k	1.55 k	244.40	1.05 k
	Y	0.00	1.05 k	0.00	239.40	1.05 k	0.00	239.40	1.05 k
ChEMBL_R	N	96.80	5.98 k	56.62 M	3.47 k	56.62 M	33.59 k	5.07 k	5.88 k
	Y	96.80	5.98 k	56.62 M	3.47 k	56.62 M	14.39 M	5.84 k	5.88 k
Claros_L	N	1.29 k	6.49 k	14.01 k	1.91 k	19.21 k	6.83 k	3.63 k	5.20 k
	Y	0.07	5.20 k	2.07	1.11 k	5.21 k	2.73	2.45 k	5.20 k
Claros_{LE}	N	13.45 k	2.23 M	8.19 G	118.11 k	8.19 G	1.89 G	74.65 M	2.22 M
	Y	111.20	2.22 M	3.51 G	52.19 k	3.51 G	1.85 G	131.80 M	2.22 M
DBpedia_L	N	45.87	869.13	305.60	20.57	1.13 k	195.53	378.13	823.27
	Y	11.00	834.27	213.00	18.73	1.04 k	115.10	348.93	823.27
Reactome_L	N	184.93	2.03 k	3.19 k	1.96 k	5.04 k	2.67 k	857.43	1.85 k
	Y	0.00	1.85 k	0.10	945.47	1.85 k	129.40	946.60	1.85 k
Reactome_R	N	19.34 k	151.02 k	38.51 M	102.87 k	38.64 M	155.58 k	38.56 k	131.68 k
	Y	19.34 k	151.02 k	38.51 M	102.87 k	38.64 M	155.58 k	38.56 k	131.68 k
Reactome_U	N	850.10	6.38 k	17.41 k	5.75 k	22.94 k	10.56 k	2.04 k	5.53 k
	Y	0.00	5.53 k	527.83	2.22 k	6.06 k	269.87	2.40 k	5.53 k
UniProt_L	N	134.33	1.49 k	640.63	815.60	1.99 k	1.49 k	810.30	1.35 k
	Y	0.00	1.35 k	212.20	676.00	1.56 k	86.90	707.97	1.35 k
UniProt_R	N	882.47	5.83 k	17.89 M	1.62 k	17.89 M	13.82 k	4.57 k	4.95 k
	Y	901.03	5.85 k	17.89 M	1.54 k	17.89 M	17.52 M	3.96 k	4.95 k
UOBM-01K_L	N	15.33	3.12 k	145.00 k	79.15 k	148.10 k	2.53 k	2.20 k	3.11 k
	Y	1.73	3.11 k	3.81 k	2.35 k	6.92 k	2.04 k	2.15 k	3.11 k
UOBM-01K_R	N	316.07	10.53 k	5.92 M	80.69 k	5.93 M	82.29 k	21.51 k	10.21 k
	Y	78.23	10.29 k	156.82 k	2.50 k	167.03 k	120.30 k	17.84 k	10.21 k

In contrast, the running times of counting have little correlation with the number derivations (correlation coefficient of just 0.15), but they correlative strongly with the size of the materialised dataset (correlation coefficient of 0.99). This confirms our observation from the previous paragraph that the performance of counting on small updates is dominated by the need to reconstruct I_o and I_n.

Impact of stratification As expected (see Section 8.5), stratifying the rules improves the performance of counting and DRed in terms of both the numbers of derivations and the running times. For counting, the breakdown of the numbers of derivations in Table 4 shows that stratification helps avoid redundant derivations that 'move' counters as in the proof of Proposition 34; for example, on ChEMB_L, Reactome_L, Reactome_U, and UniProt_L, the algorithm performs no such derivations when the rules are stratified, in contrast to the case when the rules are unstratified. DRed benefits from stratification considerably more than counting: the numbers of derivations and the running times decrease considerably on Claros_{LE}, Reactome_L, Reactome_U, UOBM-01K_L, and UOBM-01K_R. As one can see in Table 4, DRed rederives (column 'fwd') a considerable portion of the overdeleted facts implied by the large number of deletion derivations (column 'del'). Overall, the running times of DRed improve by a factor 5.85 on average, and the numbers of derivations reduce by a factor for 7.52, as shown in Table 3. Finally, the performance of FBF also improves with stratification, but to a lesser extent, as one can see from the average ratios reported in Table 3. Interestingly, the number of derivations of FBF actually increases with stratification on ChEMBL_R and UOBM-01K_R, and we determined that this is due to the behaviour described in the proof of Proposition 36. In the former case, this increase actually makes FBF slower with stratification than without, whereas in the latter case the impact of this effect on the performance is negligible.

Impact of recursion DRed and FBF differ only in their handling of recursive rules, and their performance diverges mainly on ChEMBL_R, Reactome_R, UOBM-01K_R, and UniProt_R, which contain many recursive rules (cf. Table 2). Recursive rules cannot be stratified, so DRed has no choice but to apply overdeletion and rederivation, and this incurs a considerable overhead as one can see from the breakdown of the numbers of derivations. In contrast, by eagerly searching for alternative proofs via backward chaining, FBF can prevent overdeletion: as one can see in Table 4, in all of these tests the number of derivations during deletion propagation (column 'del') is at least one order of magnitude smaller in FBF than in DRed. On programs without many recursive rules, the performance of DRed and FBF is similar. Finally, although the counting algorithm can perform redundant derivations with recursive rules that 'move' counters in a trace (see the proof of Proposition 29), the numbers of added derivations (column 'add') in Table 4 are negligible compared to the numbers of deleted derivations (column 'del') in all cases apart from Reactome_R.

Table 5
Deleting 1,000 facts selected by random walk sampling on Claros$_{LE}$.

Instance	Strat	Time (s)			Derivations		
		CNT	DRed	FBF	CNT	DRed	FBF
Claros$_{LE}$	N	25.94 ± 0.44	5.58 ± 2.78	4.69 ± 2.64	82.41 k ± 23.54 k	151.56 k ± 43.66 k	121.30 k ± 34.52 k
	Y	26.06 ± 0.65	4.18 ± 2.36	4.64 ± 2.66	80.78 k ± 23.56 k	81.77 k ± 23.61 k	114.09 k ± 33.74 k

Claros$_{LE}$ We now explain the poor performance of DRed and FBF on Claros$_{LE}$. The program of Claros$_{LE}$ contains a symmetric and transitive predicate *relatedPlaces*, so the materialisation consists of cliques of all constants reachable in the explicit facts via this predicate. The rule that axiomatises *relatedPlaces* as transitive contains three variables, so computing all edges in a clique is cubic in the number of constants involved in the clique. The materialisation contains several such cliques and the largest one connects 2,270 constants; thus, computing the closure of *relatedPlaces* dominates the materialisation time. To see how maintaining this clique dominates the update time, consider deleting an explicit fact *relatedPlaces*(a, b). Since *relatedPlaces* is symmetric and transitive, for all c and d belonging to the clique of a and b, fact *relatedPlaces*(c, d) depends (possibly indirectly) on *relatedPlaces*(a, b). Moreover, there are two possible outcomes of deletion: (i) the clique remains intact if a and b are reachable in the explicit dataset by a surviving path, or (ii) the clique breaks down into two cliques, one for a and one for b, if no paths from a to b survive in the explicit dataset. These observations lead to the following behaviour of our algorithms.

- The number of iterations needed to compute a clique is logarithmic in the number of objects in a clique, and each iteration derives many clique edges; thus, deleting *relatedPlaces*(a, b) is unlikely to significantly alter a trace of the counting algorithm, which thus typically just deletes only the derivations of facts involving a or b whose number is linear in the number of constants in the clique.
- The DRed algorithm always overdeletes the entire clique of a and b and then rederives the 'surviving' parts. In doing so, the algorithm repeats a substantial portion of the cubic work from the initial materialisation whenever deletion involves a fact of the form *relatedPlaces*(a, b) that is part of a large clique.
- If the clique of a and b remains intact, the FBF algorithm can reprove *relatedPlaces*(a, b) and thus prevent the deletion of the entire clique; however, this requires computing parts of the clique in the Saturate procedure. In contrast, if the clique of a and b breaks down, then the FBF algorithm considers each clique edge either in deletion propagation or in saturation and so the performance of FBF becomes similar to DRed.

As Table 3 shows, the performance of FBF is generally better than of DRed, which is due to the fact that the uniform sampling procedure sometimes produces 'easy' inputs. Specifically, FBF (with either stratified or unstratified rules) recomputed large cliques in five out of 30 samples, and DRed with stratified and unstratified rules did so in nine and 21 cases, respectively. However, neither DRed nor FBF seems to be robust on Claros$_{LE}$.

Random walk sampling We ran all of the robustness deletion experiments with random walk sampling as well, but, apart from Claros$_{LE}$, the results were almost identical to those reported in Table 3, so we only present the results for Claros$_{LE}$ in Table 5. Both DRed and FBF seem to exhibit much better performance in this case, which can be explained as follows. Random walk sampling selects related facts so, if the sampling procedure starts in a part of the dataset that is not involved in large cliques of constants connected by the *relatedPlaces* predicate, it is unlikely to select a subset of the explicit dataset involving a large clique. In fact, none of the 30 inputs generated using random walk sampling involved a large clique, and so both DRed and FBF could process the update quickly. In contrast, uniform sampling selects facts independently from each other, and so it is about 30 times more likely than random walk sampling to select a fact involved in a large clique. However, even with random walk sampling, the z-confidence intervals for both the running times and the numbers of derivations of DRed and FBF are comparable to the average values, suggesting that the two algorithms are not robust on Claros$_{LE}$ even on inputs produced by random walk sampling.

10.3.2. Robustness addition experiments

Table 6 shows the results of our robustness addition experiments where the subsets of 1,000 facts were selected using random sampling; the results for random walk sampling are almost identical on all test instances, so we do not discuss them further. As we observed in Section 8.4, DRed and FBF behave identically on such inputs, so we report their results jointly. For each test instance, the table shows the running time and the number of derivations with their z-confidence intervals for the unstratified and stratified variants of the program. The table also shows the breakdown of the numbers of derivations into added and deleted ones for the counting algorithm; in contrast, no further breakdown is possible for DRed/FBF since these algorithms only perform forward chaining on the given inputs. Finally, the table shows the averages, scores, medians, and Pearson correlation coefficients analogously to Table 3. We next discuss these results in detail.

Optimality of DRed/FBF As we argued in Section 8.4, DRed and FBF are optimal on such updates, and their performance should not be affected by the use of stratification. In practice, we observed a minor slowdown on ChEMBL$_R$ and Claros$_{LE}$ with stratification, which is due to the implementation issues we discuss in the following paragraph. The running time again correlates strongly with the number of derivations, and moreover the update was computed in under 50 ms in all

Table 6
Adding 1,000 facts selected by uniform sampling.

Instance	Strat	Time (s)		Derivations			
		CNT	DRed/FBF	CNT			DRed/FBF
				added	deleted	total	total
ChEMBL_L	N	11.72 ± 0.21	0.01 ± 0.00	1.10 k	48.40	1.14 k ± 6.62	1.05 k ± 8.72
	Y	11.37 ± 0.21	0.01 ± 0.00	1.05 k	0.00	1.05 k ± 8.72	1.05 k ± 8.72
ChEMBL_R	N	36.85 ± 0.38	0.04 ± 0.00	5.98 k	96.80	6.07 k ± 54.00	5.88 k ± 57.78
	Y	37.17 ± 0.32	0.04 ± 0.00	5.98 k	96.80	6.07 k ± 54.00	5.88 k ± 57.78
Claros_L	N	3.01 ± 0.02	0.02 ± 0.00	6.49 k	1.29 k	7.77 k ± 92.78	5.20 k ± 85.01
	Y	2.72 ± 0.02	0.02 ± 0.00	5.20 k	0.07	5.20 k ± 85.04	5.20 k ± 85.01
Claros_{LE}	N	26.70 ± 1.10	0.77 ± 0.51	2.23 M	13.45 k	2.24 M± 1.90 M	2.22 M ± 1.91 M
	Y	25.84 ± 0.79	0.71 ± 0.41	2.22 M	111.20	2.22 M± 1.91 M	2.22 M ± 1.91 M
DBpedia_L	N	24.11 ± 1.10	0.01 ± 0.00	869.13	45.87	915.00 ± 16.47	823.27 ± 18.61
	Y	24.01 ± 0.36	0.01 ± 0.00	834.27	11.00	845.27 ± 18.72	823.27 ± 18.61
Reactome_L	N	0.81 ± 0.02	0.01 ± 0.00	2.03 k	184.93	2.22 k ± 41.63	1.85 k ± 41.34
	Y	0.72 ± 0.02	0.01 ± 0.00	1.85 k	0.00	1.85 k ± 41.34	1.85 k ± 41.34
Reactome_R	N	15.67 ± 0.16	0.11 ± 0.00	151.02 k	19.34 k	170.36 k ± 11.08 k	131.68 k ± 11.09 k
	Y	15.44 ± 0.15	0.11 ± 0.00	151.02 k	19.34 k	170.36 k ± 11.08 k	131.68 k ± 11.09 k
Reactome_U	N	1.74 ± 0.01	0.02 ± 0.00	6.38 k	850.10	7.23 k ±135.36	5.53 k ±110.88
	Y	1.62 ± 0.05	0.02 ± 0.00	5.53 k	0.00	5.53 k ±110.88	5.53 k ±110.88
UniProt_L	N	6.28 ± 0.08	0.01 ± 0.00	1.49 k	134.33	1.62 k ± 12.86	1.35 k ± 9.46
	Y	6.25 ± 0.14	0.01 ± 0.00	1.35 k	0.00	1.35 k ± 9.46	1.35 k ± 9.46
UniProt_R	N	21.69 ± 0.38	0.04 ± 0.00	5.83 k	882.47	6.71 k ±119.71	4.95 k ± 80.45
	Y	21.78 ± 0.77	0.04 ± 0.00	5.85 k	901.03	6.75 k ±119.76	4.95 k ± 80.45
UOBM-01K_L	N	20.70 ± 0.63	0.02 ± 0.00	3.12 k	15.33	3.14 k ± 45.46	3.11 k ± 45.95
	Y	16.61 ± 0.19	0.02 ± 0.00	3.11 k	1.73	3.11 k ± 45.69	3.11 k ± 45.95
UOBM-01K_R	N	61.47 ± 2.15	0.05 ± 0.00	10.53 k	316.07	10.84 k ±382.39	10.21 k ±164.57
	Y	50.88 ± 1.78	0.05 ± 0.00	10.29 k	78.23	10.37 k ±177.07	10.21 k ±164.57
Average value:	N	19.23	0.09	201.98 k	3.05 k	205.03 k	198.92 k
	Y	17.87	0.09	200.64 k	1.71 k	202.35 k	198.92 k
	All	18.55	0.09	201.31 k	2.38 k	203.69 k	198.92 k
Average ratio unstratified/ stratified:		1.07	0.86			1.12	1.00
Average score:	N	757.16	1.00			1.18	1.00
	Y	590.76	1.00			1.06	1.00
	All	673.96	1.00			1.12	1.00
Median score:	N	619.02	1.00			1.16	1.00
	Y	520.64	1.00			1.00	1.00
	All	540.56	1.00			1.03	1.00
Correlation of time and # derivations:	N	0.13	1.00				
	Y	0.16	1.00				
	All	0.15	1.00				
Correlation of time and mat. size:	N	0.98	0.11				
	Y	0.99	0.11				
	All	0.98	0.11				

cases apart from Claros_{LE} and Reactome_R. The programs of these two test instances are highly recursive, but even then the updates took less than 800 ms.

Impact of stratification on DRed/FBF The derivations to be added are the same regardless of whether the rules are stratified or not, and so the numbers of derivations are exactly the same for the stratified and unstratified variants of each test instance. It is therefore counterintuitive that DRed and FBF seem to perform worse with stratified rules: the average ratio between the respective running times is 0.86 (row 'Average ratio unstratified/stratified' of Table 6). This, however, is due to how our system implements rule matching in line 53 of Algorithm 4: in each stratum, our system iterates over the content of $A \setminus D$ and $D \setminus A$ and searches for applicable rules. The number of such iterations increases with the number of strata, which incurs a small overhead.

Behaviour of counting The behaviour of the counting algorithm on the deletion and addition tests is symmetric: all deleted (resp. added) derivations during a deletion test become added (resp. deleted) derivations in the corresponding addition test. The running times for the addition tests are thus very close to the running times for the deletion tests and are again dominated by the overhead of reconstructing I_o and I_n; consequently, the counting algorithm is orders of magnitude slower than DRed/FBF. Moreover, the counting algorithm is not optimal for addition: in the presence of recursive rules, the algorithm may need to delete certain rule instances in order to 'move' counters as in the proof Proposition 34. In most cases this overhead is not pronounced, but it can be significant on programs such as Claros_{LE} and Reactome_R that contain difficult recursive rules. Stratification seems important in avoiding such redundant derivations.

10.4. Scalability experiments

The aim of our scalability experiments was to investigate how the behaviour of our algorithms depends on the update size. As we have explained in Section 4, deletion of facts is the main source of difficulty for materialisation maintenance algorithms, so one of our goals was to identify *break-even points*—subsets E' of the explicit datasets E such that incrementally deleting E' takes the same time as rematerialising $E \setminus E'$ 'from scratch'. An algorithm achieving a larger break-even point is more scalable since it can process larger updates before rematerialisation becomes more efficient. For each test instance, stratification type, and algorithm, many different break-even points may exist. However, to make our tests feasible, we determined just one such point by fixing a seed for the uniform sampling procedure and using binary search. That is, we first determined the subset $E_1 \subseteq E$ of size $|E|/2$, and then we measured the time t to delete $E^- = E_1$ from E incrementally and the time t' to rematerialise $E \setminus E_1$. We stopped the search if $|\frac{t-t'}{(t+t')/2}| < 2.5\%$—that is, if the difference between the two times was less than 2.5% of the average. Otherwise, we repeated these steps by identifying a subset $E_2 \subseteq E$ of size $3|E|/4$ if $t < t'$, and of size $|E|/4$ if $t > t'$. We performed at most ten such iterations in order to make the search practicable. We did not consider random walk sampling: in most cases the break-even points were large, so random walk sampling would be unlikely to produce very different results.

Please note that we compared counting with rematerialisation by Algorithm 2 (written 'CRemat'), and DRed and FBF with rematerialisation by Algorithm 1 (written 'Remat'). While CRemat is generally slower than Remat, our experiments show that the running times for the two algorithms are of the same orders of magnitude.

For each test instance and stratification type, we thus obtained three break-even points. Another important objective of our tests was to determine how our algorithms scale with update sizes. Thus, we also computed 25%, 50%, and 75% of each of the three break-even points, thus obtaining 12 increasing subsets $E_1 \subseteq \cdots \subseteq E_{12} \subseteq E$ of the explicit facts. On some test instances, some of these sets were duplicated (e.g., because of identical break-even points or simply by coincidence); we removed all such duplicates, so for some test instances there are fewer than 12 data points. Finally, for each remaining set E_i, we conducted the following two types of experiments.

- In the *scalability deletion experiments*, we deleted $E^- = E_i$ from $\text{mat}(\Pi, E)$.
- In the *scalability addition experiments*, we added $E^+ = E_i$ to $\text{mat}(\Pi, E \setminus E_{12})$ (since E_{12} is the largest subset of E).

10.4.1. Break-even points

Table 7 shows the break-even points for all test instances, stratification types, and algorithms, represented both as the absolute numbers of facts and the percentages of the explicit datasets. The table also shows the average, median, and maximum scores of all algorithms computed across all test instances without and with stratification.

As one can see from the table, the counting algorithm generally achieved the largest break-even points, and moreover its break-even point was never smallest. FBF achieved the largest break-even point on ChEMBL_R and UniProt_R without stratification, and on Reactome_R with stratification, but the difference in the percentages of the break-even point sizes for counting and FBF is under 10% in all these three cases. DRed was the worst performing algorithm in all cases apart from UniProt_L without and with stratification, and Reactome_L with stratification, where it was marginally better than FBF. Finally, DRed and FBF were roughly tied on ChEMBL_L and Claros_{LE} with stratification.

The scores (cf. Section 10.2) again allow us to compare the algorithms across all test instances, and they show that counting is the best on average, followed by FBF and then DRed. This is quite different from the robustness tests, where the counting algorithm generally exhibited the worst performance. As we explained in Section 10.3, the counting algorithm must reconstruct I_o and I_n, but this cost depends on the size of materialisation, rather than the size of the update. Thus, as the work involved in computing the update increases, this reconstruction cost becomes less significant. As we have already observed in Section 10.3, precise tracking of inferences usually allows the counting algorithm to make the smallest number of derivations, and this advantage becomes apparent when updates are large.

The average, median, and maximum scores for DRed and FBF with stratification are smaller than without stratification, suggesting that stratifying the rules allows DRed and FBF to close some (but not all) of the gap in the performance with respect to counting. In fact, stratification seems to most beneficial for DRed, as suggested by the biggest reduction in the scores. Moreover, the break-even points of DRed and FBF are reasonably close on lower bound (L) programs with stratification, whereas they are further apart when programs are unstratified. This is consistent with our discussion from Section 8: on nonrecursive rules, DRed and FBF become the same algorithm, and the only difference to counting is in one-step rederivation.

Table 7
The break-even points per algorithm, instance, and stratification type.

Instance	Break-even point without stratification			Break-even point with stratification		
	CNT	DRed	FBF	CNT	DRed	FBF
ChEMBL$_L$	137.8 M (48%)	36.1 M (12%)	81.9 M (28%)	142.9 M (49%)	126.5 M (44%)	126.5 M (44%)
ChEMBL$_R$	135.6 M (47%)	49.1 M (17%)	162.7 M (56%)	200.5 M (69%)	54.2 M (19%)	167.2 M (58%)
Claros$_L$	9.4 M (50%)	2.2 M (12%)	4.0 M (21%)	12.3 M (66%)	5.3 M (28%)	5.5 M (29%)
Claros$_{LE}$	1.5 M (8%)	18.4 k (0%)	73.4 k (0%)	2.0 M (11%)	146.8 k (1%)	146.8 k (1%)
DBpedia$_L$	205.4 M (53%)	60.4 M (16%)	72.5 M (19%)	193.4 M (50%)	95.1 M (25%)	98.6 M (25%)
Reactome$_L$	5.5 M (44%)	1.4 M (11%)	2.6 M (20%)	8.6 M (69%)	2.7 M (22%)	2.3 M (19%)
Reactome$_R$	3.5 M (28%)	1.8 M (14%)	4.3 M (34%)	3.3 M (27%)	1.8 M (14%)	4.3 M (34%)
Reactome$_U$	5.9 M (47%)	12.2 k (0%)	36.6 k (0%)	8.7 M (70%)	48.8 k (0%)	79.3 k (1%)
UniProt$_L$	65.4 M (53%)	41.6 M (34%)	38.2 M (31%)	69.7 M (57%)	51.0 M (41%)	42.3 M (34%)
UniProt$_R$	38.5 M (31%)	29.3 M (24%)	43.0 M (35%)	57.7 M (47%)	32.9 M (27%)	42.3 M (34%)
UOBM-01K$_L$	119.4 M (47%)	79.9 M (31%)	95.5 M (37%)	126.1 M (50%)	79.6 M (31%)	96.5 M (38%)
UOBM-01K$_R$	127.4 M (50%)	8.5 M (3%)	113.9 M (45%)	193.1 M (76%)	15.4 M (6%)	91.6 M (36%)
Average score:	1.05	50.06	16.34	1.02	18.65	11.75
Median score:	1.00	3.61	1.70	1.00	2.39	1.80
Max. score:	1.22	480.05	160.01	1.29	178.01	109.57

Counting outperforms DRed and FBF by several orders of magnitude on two test instances. The first one is Claros$_{LE}$, and the reasons for this are as in Section 10.3: the performance of FBF deteriorates significantly when an update involves a fact of the form *relatedPlaces*(a, b) with a and b belonging to a large clique of constants that does not survive the deletion. While DRed and FBF differ in performance on small updates (see in Section 10.3), facts are much less likely to survive deletion on large updates and so the advantage of FBF over DRed with stratification disappears. Counting also outperforms DRed and FBF by orders of magnitude on Reactome$_U$: both with and without stratification, the break-even points of counting are several orders of magnitude larger than for the other two algorithms. Our investigation revealed this to be due to the backward chaining problems described in Example 38 from Section 9: rule $\mathit{physicalEntity}(x, y) \wedge \mathit{physicalEntity}(x, z) \rightarrow \mathit{owl{:}sameAs}(y, z)$ form the Reactome$_U$ program corresponds to rule (161) in the example and the evaluation of its body during backward chaining requires enumerating many partial matches that do not produce a complete rule instance.

10.4.2. Scalability deletion experiments

Table 8 shows, for each test instance and stratification type, the running times and the numbers of derivations for the deletion experiment on the largest of the three break-even points; in the notation from the beginning of this section, this is the experiment of deleting $E^- = E_{12}$ from $\mathrm{mat}(\Pi, E)$. On each test instance, the rematerialisation approaches for DRed and FBF (Algorithm 1) and counting (Algorithm 2) perform the same derivations, but the latter also maintains counters, which can be a source of overhead. Thus, for each test instance, the table shows two rematerialisation times as Remat and CRemat, but just one number of derivations as (C)Remat. As one can see from the table, the difference between Remat and CRemat is generally small, which justifies our definition of break-even points from the beginning of the section. The table also shows the average times and numbers of derivations, the average and median scores of these values, and the correlation between the running times and the numbers of derivations and materialisation sizes. The scores take all five algorithms into account—that is, the best-performing algorithm is selected among counting, DRed, FBF, Remat, and CRemat. Please note that the break-even points with and without stratification for a test instance are generally different, and so the total work in the two cases is usually different; hence, the impact of stratification cannot be estimated by a direct comparison of the absolute running times and/or the numbers of derivations.

Comparison The counting algorithm was fastest in 19 cases, and FBF was fastest in the remaining five (ChEMBL$_R$, Reactome$_R$, UniProt$_R$, and UOBM-01K$_R$ without stratification, and Reactome$_R$ with stratification). DRed was never fastest, but it outperformed FBF in nine cases. All of these cases involve deletion of about 50% of the explicit facts, which is well beyond the break-even points of both DRed and FBF. Facts are less likely to survive deletion on such large updates, so FBF is less efficient as its searches for surviving proofs are usually unsuccessful. Moreover, the large difference between the average scores for DRed (5.78) and FBF (13.85) without stratification is due to the very poor performance of FBF on Reactome$_U$, which, as we explained in Section 10.4.1, is due to the problems with backward chaining described in Example 38. In contrast, the median score for FBF without stratification is slightly smaller than for DRed.

Numbers of derivations The counting algorithm performs the least number of derivation in all cases, showing that the ability to track derivations precisely is very effective at identifying the work needed for the update. In nine cases the numbers of derivations for DRed are smaller than for FBF; however, the average and the median scores are slightly smaller for FBF than for DRed. This can be explained by the observation that, when updates are large, facts are much less likely to survive deletion. Thus, most of the facts processed in the overdeletion phase of DRed will actually be deleted, whereas looking for alternative proofs using backward chaining becomes a source of overhead in FBF.

Table 8
The times and total numbers of derivations for deleting the largest break-even point.

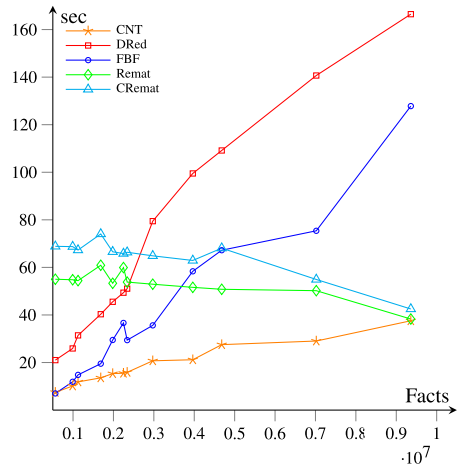
Instance	Strat	Facts	Time (s)					Derivations			
			CNT	DRed	FBF	Remat	CRemat	CNT	DRed	FBF	(C)Remat
ChEMBL_L	N	137.81 M	105.25	198.26	136.52	66.53	105.25	159.46 M	197.85 M	193.58 M	180.44 M
	Y	142.90 M	115.73	118.87	138.19	87.82	102.71	151.56 M	163.93 M	163.93 M	174.99 M
ChEMBL_R	N	162.67 M	3.17 k	4.87 k	2.64 k	2.64 k	2.78 k	1.07 G	2.71 G	1.86 G	1.28 G
	Y	200.51 M	2.74 k	4.79 k	2.90 k	2.59 k	2.68 k	1.35 G	2.70 G	2.04 G	1.01 G
Claros_L	N	9.36 M	37.56	166.47	127.82	38.30	42.51	69.09 M	97.98 M	102.72 M	73.38 M
	Y	12.33 M	51.14	105.74	131.47	35.71	41.54	75.48 M	81.16 M	97.61 M	52.74 M
Claros_{LE}	N	1.47 M	2.83 k	21.84 k	17.24 k	2.28 k	2.69 k	5.80 G	18.94 G	16.08 G	7.57 G
	Y	2.03 M	2.99 k	19.90 k	19.75 k	2.14 k	2.67 k	7.14 G	18.31 G	16.52 G	7.04 G
DBpedia_L	N	205.44 M	174.89	965.16	1.03 k	189.43	192.60	188.03 M	213.19 M	260.59 M	171.13 M
	Y	193.35 M	264.04	903.86	1.04 k	203.33	276.07	166.23 M	193.53 M	238.29 M	181.50 M
Reactome_L	N	5.46 M	18.95	44.69	30.50	17.91	18.97	11.32 M	18.98 M	17.13 M	14.19 M
	Y	8.63 M	14.62	26.69	39.56	6.65	13.06	15.62 M	18.05 M	18.07 M	8.38 M
Reactome_R	N	4.29 M	1.21 k	1.33 k	731.46	764.34	876.07	651.05 M	1.95 G	739.43 M	1.14 G
	Y	4.29 M	1.06 k	1.29 k	711.58	753.00	869.63	651.05 M	1.95 G	739.43 M	1.14 G
Reactome_U	N	5.85 M	27.29	929.09	3.42 k	32.81	35.04	41.16 M	85.30 M	60.25 M	52.10 M
	Y	8.68 M	28.20	4.17 k	4.27 k	23.54	33.56	54.13 M	61.95 M	62.54 M	32.74 M
UniProt_L	N	65.39 M	71.26	114.15	138.96	67.04	78.43	101.89 M	125.64 M	137.78 M	90.45 M
	Y	69.72 M	64.45	117.14	146.36	53.06	68.83	98.73 M	113.12 M	115.02 M	84.01 M
UniProt_R	N	43.03 M	2.49 k	3.06 k	2.24 k	2.13 k	2.19 k	290.10 M	674.31 M	469.25 M	586.85 M
	Y	57.70 M	2.08 k	3.01 k	2.54 k	1.68 k	1.73 k	382.98 M	709.73 M	546.76 M	500.35 M
UOBM-01K_L	N	119.42 M	330.01	400.11	349.40	280.23	307.06	371.93 M	488.77 M	588.05 M	424.78 M
	Y	126.13 M	342.93	428.25	375.15	243.76	286.81	389.04 M	497.53 M	544.19 M	404.20 M
UOBM-01K_R	N	127.38 M	4.20 k	5.76 k	3.66 k	2.76 k	4.40 k	1.34 G	3.74 G	2.69 G	1.60 G
	Y	193.06 M	2.66 k	5.31 k	3.93 k	1.88 k	1.94 k	1.90 G	3.50 G	3.01 G	996.20 M
Average value:	N		1.22 k	3.31 k	2.65 k	938.65	1.14 k	841.64 M	2.44 G	1.93 G	1.10 G
	Y		1.03 k	3.35 k	3.00 k	807.52	893.79	1.03 G	2.39 G	2.01 G	968.47 M
	All		1.13 k	3.33 k	2.82 k	873.08	1.02 k	936.63 M	2.40 G	1.97 G	1.03 G
Average score:	N		1.22	5.78	13.85	1.03	1.21	1.02	2.02	1.63	1.28
	Y		1.39	17.64	18.09	1.00	1.26	1.28	2.01	1.76	1.11
	All		1.31	11.71	16.06	1.02	1.23	1.15	2.01	1.70	1.19
Median score:	N		1.17	2.29	2.05	1.00	1.15	1.00	1.87	1.52	1.19
	Y		1.36	2.52	2.43	1.00	1.20	1.10	1.87	1.64	1.00
	All		1.24	2.35	2.07	1.00	1.17	1.00	1.87	1.52	1.11
Correlation of time and # derivations:	N		0.55	0.99	0.98	0.56	0.53				
	Y		0.70	0.97	0.97	0.57	0.66				
	All		0.60	0.98	0.97	0.57	0.58				
Correlation of time and mat. size:	N		0.86	0.35	0.23	0.83	0.86				
	Y		0.78	0.30	0.21	0.78	0.75				
	All		0.82	0.32	0.22	0.80	0.80				

Performance factors As in our robustness experiments, the running times for DRed and FBF are strongly correlated with the numbers of derivations. In contrast, the running times for counting are reasonably correlated with materialisation sizes, but they are not so strongly correlated with the numbers of derivations. We believe the latter to be mainly due to the difficulties in computing $\mathsf{inst}_r\big[I^+, I^- \vdots P_1, N_1 \vdots P_2, N_2\big]$ we discussed in Section 9.2.

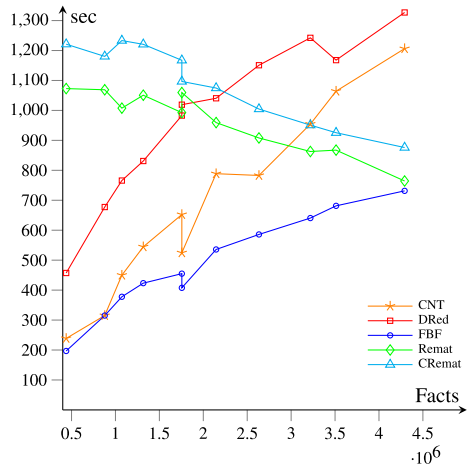
Scaling behaviour The results for all test instances and stratification types are available online.[9] By manually grouping similar results, we identified six representative test instances shown in Fig. 8. Each graph illustrates a particular relationship in the performance of our algorithms as summarised below the graph. These figures are consistent with our observations thus far: the counting and the FBF algorithms offer best performance overall, with the performance of counting being most consistent.

10.4.3. Scalability addition experiments

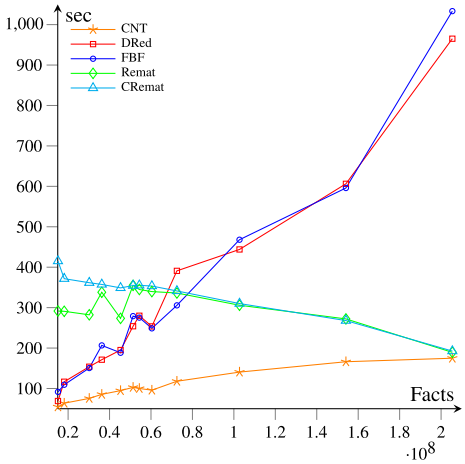
Table 9 shows the times and the numbers of derivations for the scalability addition experiment with the largest break-even point; in the notation from the beginning of this section, this is the experiment for adding $E^+ = E_{12}$ to $\mathsf{mat}(\Pi, E \setminus E_{12})$. As we discussed in Section 8.4, on such updates DRed and FBF amount to the same algorithm, so we report their times



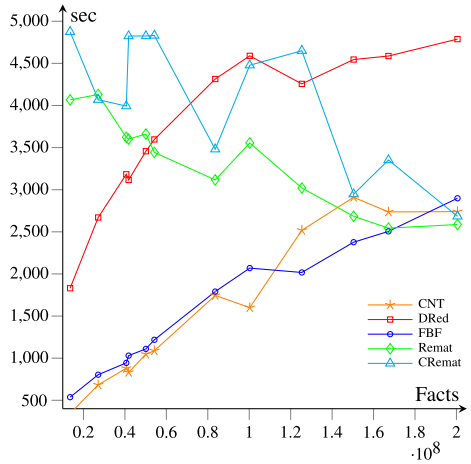
Claros$_L$ unstratified
(six cases: CNT outperforms FBF, which outperforms DRed)



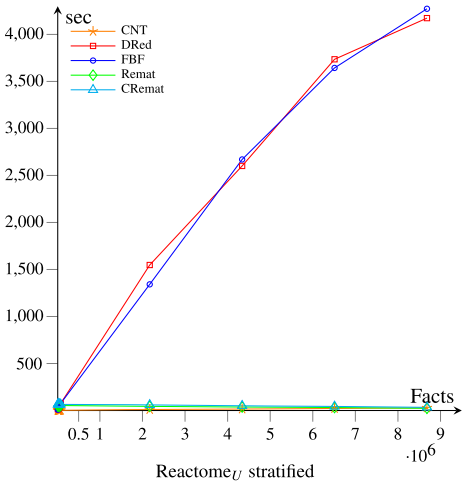
Reactome$_R$ unstratified
(five cases: FBF outperforms CNT, which outperforms DRed)



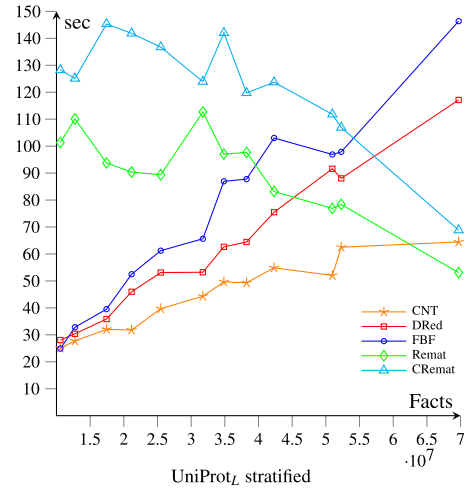
DBPedia$_L$ unstratified
(five cases: CNT outperforms DRed and FBF, which are similar)



ChEMBL$_R$ stratified
(four cases: FBF and CNT similar, and they outperform DRed)



Reactome$_U$ stratified
(two cases: CNT outperforms DRed and FBF by orders of magnitude)



UniProt$_L$ stratified
(two cases: CNT outperforms DRed, which outperforms FBF)

Fig. 8. Running time vs. the number of deleted facts for typical scalability deletion experiments.

Table 9
The times and total numbers of derivations for adding the largest break-even point.

Instance	Strat	Facts	Time (s)		Derivations	
			CNT	DRed/FBF	CNT	DRed/FBF
ChEMBL$_L$	N	137.81 M	79.71	52.56	159.46 M	146.11 M
	Y	142.90 M	97.31	59.37	151.56 M	151.56 M
ChEMBL$_R$	N	162.67 M	4.17 k	1.78 k	1.07 G	1.04 G
	Y	200.51 M	4.19 k	2.27 k	1.35 G	1.31 G
Claros$_L$	N	9.36 M	39.52	20.76	69.09 M	54.84 M
	Y	12.33 M	55.34	48.02	75.48 M	75.48 M
Claros$_{LE}$	N	1.47 M	2.68 k	1.43 k	5.80 G	5.06 G
	Y	2.03 M	2.76 k	1.23 k	7.14 G	5.59 G
DBpedia$_L$	N	205.44 M	252.06	169.20	188.03 M	171.96 M
	Y	193.35 M	262.97	266.23	166.23 M	161.59 M
Reactome$_L$	N	5.46 M	21.38	8.57	11.32 M	9.81 M
	Y	8.63 M	14.20	16.61	15.62 M	15.62 M
Reactome$_R$	N	4.29 M	1.61 k	393.02	651.05 M	485.91 M
	Y	4.29 M	1.45 k	504.90	651.05 M	485.91 M
Reactome$_U$	N	5.85 M	32.44	17.45	41.16 M	34.64 M
	Y	8.68 M	35.06	31.13	54.13 M	53.99 M
UniProt$_L$	N	65.39 M	67.56	47.87	101.89 M	92.29 M
	Y	69.72 M	79.83	60.77	98.73 M	98.73 M
UniProt$_R$	N	43.03 M	3.16 k	1.24 k	290.10 M	228.46 M
	Y	57.70 M	2.56 k	1.52 k	382.98 M	314.97 M
UOBM-01K$_L$	N	119.42 M	444.46	241.61	371.93 M	367.99 M
	Y	126.13 M	398.17	310.47	389.04 M	388.57 M
UOBM-01K$_R$	N	127.38 M	4.76 k	1.60 k	1.34 G	1.23 G
	Y	193.06 M	3.03 k	3.32 k	1.90 G	1.84 G
Average value:	N		1.44 k	583.63	841.64 M	743.42 M
	Y		1.24 k	803.27	1.03 G	874.02 M
	All		1.34 k	693.45	936.63 M	808.72 M
Average score:	N		2.20	1.00	1.15	1.00
	Y		1.51	1.02	1.08	1.00
	All		1.85	1.01	1.11	1.00
Median score:	N		1.89	1.00	1.13	1.00
	Y		1.30	1.00	1.02	1.00
	All		1.76	1.00	1.09	1.00
Correlation of time and # derivations:	N		0.44	0.59		
	Y		0.54	0.45		
	All		0.48	0.50		
Correlation of time and mat. size:	N		0.87	0.83		
	Y		0.81	0.91		
	All		0.84	0.86		

jointly. The table also shows the average times and numbers of derivation, the average and median scores of these values, and the correlation between the running times and the numbers of derivations and materialisation sizes. As in Section 10.4.2, the impact of stratification cannot be estimated by comparing the absolute values since the break-even points with and without stratification are generally different. In fact, the break-even point with stratification is generally larger, thus incurring more work.

Comparison As one can see, the counting algorithm is slower than DRed/FBF in all but two cases (DBpedia$_L$ and Reactome$_L$ with stratification). Moreover, the maximum score of counting (i.e., the maximum slowdown) is 4.10 (on Reactome$_R$). Finally, the scores for the counting algorithm decrease if the rules are stratified: the average score reduces from 2.20 to 1.51, and the median score reduces from 1.89 to 1.30. This is in line with our expectations: DRed and FBF are optimal on such updates (cf. Section 8.4), whereas recursive rules can lead to redundant derivations that 'move' the counters in the counting algorithm (cf. proof of Proposition 29 from Section 8).

Performance factors The running times of the counting algorithm and DRed/FBF are somewhat correlated with the numbers of derivations, but less strongly than in the robustness experiments. In contrast, the running times in both cases

correlate strongly with the materialisation sizes. We believe this is due to the way rules are matched in our systems. In particular, both of our systems maintain a single index over all facts with the same predicate, and they distinguish various sets of facts using bit-masks, as explained in Section 9. Thus, the bodies of the rules are matched by first retrieving all instances regardless of any restrictions, and the performance of this step is largely determined by the total number of facts in the materialisation.

10.5. Summary

Our experiments show a mixed picture of the practical behaviour of our three algorithms, with no clear overall winner. On small updates, which are of primary concern in many practical applications, FBF seems to be generally fastest, particularly with complex and recursive rules. In contrast, the overhead of reconstructing I_o and I_n usually significantly impacts the counting algorithm. Both DRed and FBF exhibit shortcomings on inputs such as Claros_{LE} where derivations are very complex.

On large updates, the overhead of reconstructing I_o and I_n in the counting algorithm becomes less significant, allowing the algorithm to offer good and consistent performance. DRed also becomes competitive on large updates since it does not waste time searching for proofs, and the performance of FBF can become dominated by the efficiency of backward chaining.

Since no algorithm is universally best, a practical system could use a portfolio-style implementation and select an approach based on the characteristics of the data and/or the update. A slight obstacle is that the counting algorithm requires maintaining counters, so the choice between counting and DRed/FBF must be made before data is first loaded into a system. However, if counters are not maintained, either DRed or FBF can handle each update—that is, one can freely switch between the two algorithms. These choices could be automated using machine-learned models to select the most promising algorithm for a given setting and/or update, complemented with expert knowledge and observations from this section and Section 8.

11. Conclusion and outlook

In this paper we have presented three algorithms for maintenance of datalog materialisations: the counting algorithm, the Delete/Rederive (DRed) algorithm, and the Forward/Backward/Forward (FBF) algorithm. To the best of our knowledge, ours is the first counting-based algorithm that correctly handles recursive rules, and our variant of DRed is the first one that guarantees nonrepetition of derivations. The FBF algorithm is a novel extension of DRed that uses backward chaining to find alternate proofs eagerly and thus reduce the overall work in many cases. We have extensively compared these algorithms in two ways. First, we have proposed a measure of optimal behaviour of a maintenance algorithm, and we have shown that no algorithm exhibits such behaviour on all inputs. Second, we have presented the results of an extensive, first-of-a-kind empirical comparison where we studied the algorithms' behaviour on both small and large updates, which has provided us with many insights into the practical applicability of our algorithms. In particular, FBF seems to offer the best performance on small updates, which are very common in practice. In contrast, the counting algorithm can generally delete the largest amount of data before the cost of maintenance outweighs the cost of rematerialisation. This comprehensive study opens several interesting possibilities for future work.

First, it is still unclear whether a practical algorithm can be developed that handles all updates optimally according to the criterion we introduced in Section 8. We conjecture that such an algorithm could be obtained by explicitly storing and indexing all derivation trees, but it is unclear whether such a solution would be practical due to excessive memory use.

Second, we believe that some of the difficult cases could be handled by developing specialised algorithms that can handle specific rules. For example, the Claros_{LE} test instance is difficult because both materialising and maintaining a symmetric and transitive relation require cubic time. However, all elements of a clique can be easily identified in linear time using breath-first search, after which the clique can be created using quadratic time. Thus, developing a modular approach to computing and maintaining materialisations, where certain rules are handled directly using specially crafted reasoning modules, might significantly improve the performance of datalog-based systems. The main challenges for this line of work are to develop a general reasoning algorithm into which one can plug arbitrary specialised reasoning modules, and to develop such modules for practically relevant rule subsets that are difficult for the general datalog evaluation techniques.

Third, datalog is often extended with nonstandard constructs such as aggregate functions, disjunction in rule bodies, or well-behaved unstratified negation (e.g., if all cyclic dependencies between atoms in an unstratified program go through an even number of negations, the program is monotonic and thus has an intuitive semantics). In such cases, the problem of materialisation maintenance becomes more complex and needs to be revisited. A particular challenge is to devise algorithms that do not repeat derivations. In fact, when rules contain disjunctions in their bodies, it is not clear how to define the notion of a rule instance since only a subset of the body atoms can be involved in a derivation.

Fourth, modern computer systems often provide many CPUs that can be used to parallelise computation. We believe that the counting and the DRed algorithm can be parallelised easily (e.g., using the approach by Motik et al. [81]). The situation is more complex for FBF due to backward chaining: this step does not seem amenable to parallelisation, at least not in an obvious way.

Acknowledgements

This work was funded by the EPSRC projects AnaLOG, MaSI3, and ED3.

Appendix A. Proofs in Section 5

We first prove that the modified version of the seminaïve evaluation from Algorithm 2 correctly computes the materialisation of a datalog program.

Theorem 10. *Let Π be a datalog program, let λ be a stratification of Π with maximum stratum index S, let E be a dataset, and let $\overline{\boldsymbol{N}}$ be the trace of Π w.r.t. E and λ as computed by Algorithm 2. Then,* $\mathsf{mat}(\Pi, E) = \bigcup_{s=1}^{S} \bigcup_{i \geq 1} N_i^s$.

Proof. Algorithms 1 and 2 differ only in that the latter records in each iteration a multiset version of the set N. Hence, Algorithm 2 terminates with $I = \mathsf{mat}(\Pi, E)$. Now consider an arbitrary fact $F \in \mathsf{mat}(\Pi, E) = I$; fact F was added to I in line 16 for some s and i, and so we have $F \in \boldsymbol{N}_i^s$ and $F \in \bigcup_{s=1}^{S} \bigcup_{i \in \mathbb{N}} N_i^s$. Conversely, consider an arbitrary fact $F \in N_i^s$ for some s and i. In iteration i of stratum s, we have that F is either already in I or it is added to Δ in line 14 and then added to I in line 16; but then, $I = \mathsf{mat}(\Pi, E)$ implies $F \in \mathsf{mat}(\Pi, E)$. $\square$

Next, we prove Lemma 11 showing how one can efficiently compute the symmetric difference between the rule instances that fire in the 'old' and the 'new' trace.

Lemma 11. *For each rule r and all pairs of datasets (I_1, Δ_1) and (I_2, Δ_2), the following equalities hold.*

$$\mathsf{inst}_r\big[I_1\big] \setminus \mathsf{inst}_r\big[I_2\big] = \mathsf{inst}_r\big[I_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \tag{68}$$

$$\mathsf{inst}_r\big[I_1 \vdots \Delta_1\big] \setminus \mathsf{inst}_r\big[I_2 \vdots \Delta_2\big] = \mathsf{inst}_r\big[I_1 \vdots \Delta_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \cup \mathsf{inst}_r\big[(I_1 \cap I_2) \setminus \Delta_2, I_1 \cup I_2 \vdots (\Delta_1 \cap I_2) \setminus \Delta_2\big] \tag{69}$$

$$\mathsf{inst}_r\big[I_1 \vdots \Delta_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \cap \mathsf{inst}_r\big[(I_1 \cap I_2) \setminus \Delta_2, I_1 \cup I_2 \vdots (\Delta_1 \cap I_2) \setminus \Delta_2\big] = \emptyset \tag{70}$$

Proof. Consider an arbitrary rule r and an arbitrary instance r' of r, and let $B^+ = \mathsf{b}^+(r')$ and $B^- = \mathsf{b}^-(r')$. The following sequence of equalities proves property (68):

$$\begin{aligned}
& r' \in \mathsf{inst}_r\big[I_1\big] \setminus \mathsf{inst}_r\big[I_2\big] && \text{iff} \\
& B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge \neg(B^+ \subseteq I_2 \wedge B^- \cap I_2 = \emptyset) && \text{iff} \\
& B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge (B^+ \nsubseteq I_2 \vee B^- \cap I_2 \neq \emptyset) && \text{iff} \\
& B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge (B^+ \cap (I_1 \setminus I_2) \neq \emptyset \vee B^- \cap (I_2 \setminus I_1) \neq \emptyset) && \text{iff} \\
& r' \in \mathsf{inst}_r\big[I_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big].
\end{aligned}$$

The following sequence of equalities proves properties (69) and (70), where the fourth step ensures disjointness of the two disjuncts, and the fifth step is obtained by distributing $\vee$ over $\wedge$:

$$\begin{aligned}
& r' \in \mathsf{inst}_r\big[I_1 \vdots \Delta_1\big] \setminus \mathsf{inst}_r\big[I_2 \vdots \Delta_2\big] && \text{iff} \\
& [B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge B^+ \cap \Delta_1 \neq \emptyset] \wedge \neg[B^+ \subseteq I_2 \wedge B^- \cap I_2 = \emptyset \wedge B^+ \cap \Delta_2 \neq \emptyset] && \text{iff} \\
& [B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge B^+ \cap \Delta_1 \neq \emptyset] \wedge [(B^+ \nsubseteq I_2 \vee B^- \cap I_2 \neq \emptyset) \vee B^+ \cap \Delta_2 = \emptyset] && \text{iff} \\
& [B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge B^+ \cap \Delta_1 \neq \emptyset] \wedge \\
& \qquad [(B^+ \nsubseteq I_2 \vee B^- \cap I_2 \neq \emptyset) \vee (B^+ \subseteq I_2 \wedge B^- \cap I_2 = \emptyset \wedge B^+ \cap \Delta_2 = \emptyset)] && \text{iff} \\
& [B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge B^+ \cap \Delta_1 \neq \emptyset \wedge (B^+ \nsubseteq I_2 \vee B^- \cap I_2 \neq \emptyset)] \vee \\
& \qquad [B^+ \subseteq (I_1 \cap I_2) \setminus \Delta_2 \wedge B^- \cap (I_1 \cup I_2) = \emptyset \wedge B^+ \cap \Delta_1 \neq \emptyset] && \text{iff} \\
& [B^+ \subseteq I_1 \wedge B^- \cap I_1 = \emptyset \wedge B^+ \cap \Delta_1 \neq \emptyset \wedge (B^+ \cap (I_1 \setminus I_2) \neq \emptyset \vee B^- \cap (I_2 \setminus I_1) \neq \emptyset)] \vee \\
& \qquad [B^+ \subseteq (I_1 \cap I_2) \setminus \Delta_2 \wedge B^- \cap (I_1 \cup I_2) = \emptyset \wedge B^+ \cap ((\Delta_1 \cap I_2) \setminus \Delta_2) \neq \emptyset] && \text{iff} \\
& r' \in \mathsf{inst}_r\big[I_1 \vdots \Delta_1 \vdots I_1 \setminus I_2, I_2 \setminus I_1\big] \cup \mathsf{inst}_r\big[(I_1 \cap I_2) \setminus \Delta_2, I_1 \cup I_2 \vdots (\Delta_1 \cap I_2) \setminus \Delta_2\big]. \quad \square
\end{aligned}$$

We next prove a lemma that, intuitively, shows that lines 23 and 23, and 36 and 37 of Algorithm 3 indeed update the 'old' trace to the 'new' one.

Lemma 39. *Let Π be a datalog program and let (I_o, Δ_o) and (I_n, Δ_n) be two pairs of datasets, and let $I_{on} = I_o \setminus I_n$ and $I_{no} = I_n \setminus I_o$. Then, the following identities hold:*

$$\Pi\,[\![I_n]\!] = \Big(\Pi\,[\![I_o]\!] \ominus \Pi\,[\![I_o \mathrel{:} I_{on}, I_{no}]\!]\Big) \oplus \Pi\,[\![I_n \mathrel{:} I_{no}, I_{on}]\!] \tag{A.1}$$

$$\begin{aligned}\Pi\,[\![I_n \mathrel{:} \Delta_n]\!] = \Big(&\Pi\,[\![I_o \mathrel{:} \Delta_o]\!] \ominus (\Pi\,[\![I_o \mathrel{:} \Delta_o \mathrel{:} I_{on}, I_{no}]\!] \oplus \Pi\,[\![(I_o \cap I_n) \setminus \Delta_n, I_o \cup I_n \mathrel{:} (\Delta_o \cap I_n) \setminus \Delta_n]\!])\Big) \\ &\oplus (\Pi\,[\![I_n \mathrel{:} \Delta_n \mathrel{:} I_{no}, I_{on}]\!] \oplus \Pi\,[\![(I_n \cap I_o) \setminus \Delta_o, I_n \cup I_o \mathrel{:} (\Delta_n \cap I_o) \setminus \Delta_o]\!]).\end{aligned} \tag{A.2}$$

Proof. We first note that the following equations hold for any two families of sets A_i and B_i. For (A.3), note that each occurrence of some x in $\bigoplus_i (A_i \cup B_i)$ comes from some A_i or B_i, but not both; thus, the number of the occurrences of x in $\bigoplus_i (A_i \cup B_i)$ is equal to the sum of the occurrences of x in $\bigoplus_i A_i$ and in $\bigoplus_i B_i$. The argument for (A.4) is similar.

$$\text{If for each } i \text{ we have } A_i \cap B_i = \emptyset, \text{ then } \bigoplus_i (A_i \cup B_i) = \bigoplus_i A_i \oplus \bigoplus_i B_i. \tag{A.3}$$

$$\text{If for each } i \text{ we have } B_i \subseteq A_i, \text{ then } \bigoplus_i (A_i \setminus B_i) = \bigoplus_i A_i \ominus \bigoplus_i B_i. \tag{A.4}$$

Now let α_o and α_n be arbitrary arguments to operator $\Pi\,[\![\cdot]\!]$—that is, α_o is of the form $I^+, I^- \mathrel{:} P_1, N_1 \mathrel{:} P_2, N_2$, and α_n is of a similar form. The definition of the set operations ensure that properties (A.5)–(A.7) hold for each rule $r \in \Pi$.

$$\mathsf{inst}_r[\alpha_n] = \Big(\mathsf{inst}_r[\alpha_o] \setminus \big(\mathsf{inst}_r[\alpha_o] \setminus \mathsf{inst}_r[\alpha_n]\big)\Big) \cup \big(\mathsf{inst}_r[\alpha_n] \setminus \mathsf{inst}_r[\alpha_o]\big) \tag{A.5}$$

$$\emptyset = \Big(\mathsf{inst}_r[\alpha_o] \setminus \big(\mathsf{inst}_r[\alpha_o] \setminus \mathsf{inst}_r[\alpha_n]\big)\Big) \cap \big(\mathsf{inst}_r[\alpha_n] \setminus \mathsf{inst}_r[\alpha_o]\big) \tag{A.6}$$

$$\big(\mathsf{inst}_r[\alpha_o] \setminus \mathsf{inst}_r[\alpha_n]\big) \subseteq \mathsf{inst}_r[\alpha_o] \tag{A.7}$$

Now (A.5) clearly implies (A.8). Furthermore, by (A.3) and (A.6), we can distribute the outer union in (A.8) to obtain (A.9). Moreover, by (A.4) and (A.7), we can distribute the inner union in (A.9) to obtain (A.10).

$$\bigoplus_{r \in \Pi} \mathsf{inst}_r[\alpha_n] = \bigoplus_{r \in \Pi} \Big[\Big(\mathsf{inst}_r[\alpha_o] \setminus \big(\mathsf{inst}_r[\alpha_o] \setminus \mathsf{inst}_r[\alpha_n]\big)\Big) \cup \big(\mathsf{inst}_r[\alpha_n] \setminus \mathsf{inst}_r[\alpha_o]\big)\Big] \tag{A.8}$$

$$= \Big[\bigoplus_{r \in \Pi} \Big(\mathsf{inst}_r[\alpha_o] \setminus \big(\mathsf{inst}_r[\alpha_o] \setminus \mathsf{inst}_r[\alpha_n]\big)\Big)\Big] \oplus \bigoplus_{r \in \Pi} \big(\mathsf{inst}_r[\alpha_n] \setminus \mathsf{inst}_r[\alpha_o]\big) \tag{A.9}$$

$$= \Big[\Big(\bigoplus_{r \in \Pi} \mathsf{inst}_r[\alpha_o]\Big) \ominus \bigoplus_{r \in \Pi} \big(\mathsf{inst}_r[\alpha_o] \setminus \mathsf{inst}_r[\alpha_n]\big)\Big] \oplus \bigoplus_{r \in \Pi} \big(\mathsf{inst}_r[\alpha_n] \setminus \mathsf{inst}_r[\alpha_o]\big) \tag{A.10}$$

Finally, the following identity holds trivially for each α.

$$\bigoplus_{r \in \Pi} \{\!\{\, \mathsf{h}(r') \mid r' \in \mathsf{inst}_r[\alpha] \,\}\!\} = \{\!\{\, \mathsf{h}(r') \mid r' \in \bigoplus_{r \in \Pi} \mathsf{inst}_r[\alpha] \,\}\!\} \tag{A.11}$$

Let $\alpha_o = I_o$ and $\alpha_n = I_n$. By property (68) of Lemma 11, then $\bigoplus_{r \in \Pi} \big(\mathsf{inst}_r[I_o] \setminus \mathsf{inst}_r[I_n]\big) = \bigoplus_{r \in \Pi} \mathsf{inst}_r[I_o \mathrel{:} I_o \setminus I_n, I_n \setminus I_o]$, and $\bigoplus_{r \in \Pi} \big(\mathsf{inst}_r[I_n] \setminus \mathsf{inst}_r[I_o]\big) = \bigoplus_{r \in \Pi} \mathsf{inst}_r[I_n \mathrel{:} I_n \setminus I_o, I_o \setminus I_n]$. This, together with (A.10) and (A.11) imply (A.1).

We next proceed with proving (A.2). Due to (A.11), property (A.2) holds provided that the left-hand side of (A.12) is equivalent to (A.15)–(A.17). Moreover, (A.12)–(A.14) holds by (A.10) and (A.11). Thus, to prove (A.2), it suffices to show that (A.12)–(A.14) and (A.15)–(A.17) are equivalent as well.

$$\bigoplus_{r \in \Pi} \mathsf{inst}_r[I_n \mathrel{:} \Delta_n] = \Big(\bigoplus_{r \in \Pi} \mathsf{inst}_r[I_o \mathrel{:} \Delta_o] \tag{A.12}$$

$$\ominus \bigoplus_{r \in \Pi} (\mathsf{inst}_r[I_o \mathrel{:} \Delta_o] \setminus \mathsf{inst}_r[I_n \mathrel{:} \Delta_n])\Big) \tag{A.13}$$

$$\oplus \bigoplus_{r \in \Pi} (\mathsf{inst}_r[I_n \mathrel{:} \Delta_n] \setminus \mathsf{inst}_r[I_o \mathrel{:} \Delta_o]) \tag{A.14}$$

$$= \Big(\bigoplus_{r \in \Pi} \mathsf{inst}_r[I_o \mathrel{:} \Delta_o] \tag{A.15}$$

$$\ominus (\bigoplus_{r \in \Pi} \mathsf{inst}_r[I_o \mathrel{:} \Delta_o \mathrel{:} I_{on}, I_{no}] \oplus \bigoplus_{r \in \Pi} \mathsf{inst}_r[(I_o \cap I_n) \setminus \Delta_n, I_o \cup I_n \mathrel{:} (\Delta_o \cap I_n) \setminus \Delta_n])\Big) \tag{A.16}$$

$$\oplus \big(\bigoplus_{r \in \Pi} \mathsf{inst}_r\big[I_n \vdots \Delta_n \vdots I_{no}, I_{on}\big] \oplus \bigoplus_{r \in \Pi} \mathsf{inst}_r\big[(I_n \cap I_o) \setminus \Delta_o, I_n \cup I_o \vdots (\Delta_n \cap I_o) \setminus \Delta_o\big]\big) \quad \text{(A.17)}$$

Now let $\alpha_o = I_o \vdots \Delta_o$ and $\alpha_n = I_n \vdots \Delta_n$, and let $I_{on} = I_o \setminus I_n$ and $I_{no} = I_n \setminus I_o$. Then, (69) of Lemma 11 ensures (A.18); moreover, property (70) of Lemma 11 and (A.3) ensure that we can distribute $\bigoplus$ over $\cup$ on the righthand side and obtain (A.19).

$$\bigoplus_{r \in \Pi} \big(\mathsf{inst}_r\big[I_o \vdots \Delta_o\big] \setminus \mathsf{inst}_r\big[I_n \vdots \Delta_n\big]\big) =$$

$$= \bigoplus_{r \in \Pi} \big(\mathsf{inst}_r\big[I_o \vdots \Delta_o \vdots I_{on}, I_{no}\big] \cup \mathsf{inst}_r\big[(I_o \cap I_n) \setminus \Delta_n, I_o \cup I_n \vdots (\Delta_o \cap I_n) \setminus \Delta_n\big]\big) \quad \text{(A.18)}$$

$$= \big(\bigoplus_{r \in \Pi} \mathsf{inst}_r\big[I_o \vdots \Delta_o \vdots I_{on}, I_{no}\big]\big) \oplus \big(\bigoplus_{r \in \Pi} \mathsf{inst}_r\big[(I_o \cap I_n) \setminus \Delta_n, I_o \cup I_n \vdots (\Delta_o \cap I_n) \setminus \Delta_n\big]\big) \quad \text{(A.19)}$$

Thus, (A.19) allows us to substitute (A.13) with (A.16). In a completely analogous way we can derive an expression that allows us to substitute (A.14) with (A.17). □

We are now ready to prove correctness of our counting-based materialisation maintenance algorithm.

Theorem 12. *Let Π be a program, let λ be a stratification of Π with maximum stratum index S, let E be a dataset, let $\overline{\boldsymbol{N}}$ be the trace of Π w.r.t. E and λ, and let E^- and E^+ be datasets. When applied to this input, Algorithm 3 terminates and*

(a) *$\overline{\boldsymbol{N}}$ contains upon termination the trace of Π w.r.t. $(E \setminus E^-) \cup E^+$ and λ;*
(b) *lines 23 and 36 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[\mathsf{mat}(\Pi, E)\big]$ without repetition; and*
(c) *lines 24 and 37 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[\mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)\big]$ without repetition.*

Proof. (Property a) Assume that $E^- \subseteq E$ and $E^+ \cap E = \emptyset$; this is without loss of generality due to line (19) of Algorithm 3. Now let $E|_o = E$ and consider a run of Algorithm 2 on $E|_o$. Let the trace of Π w.r.t. $E|_o$ and λ consist of multisets $\boldsymbol{N}_i^s|_o$. Moreover, for each stratum index s with $1 \leq s \leq S$, let $I_i^s|_o$ and $\Delta_i^s|_o$ be the contents of I and Δ, respectively, after line 16 in iteration $i \geq 1$; let $k^s|_o$ be the value of the index i for which the condition in line 15 becomes satisfied; and let $k^0|_o = 1$, and $\boldsymbol{N}_1^0|_o = I_1^0|_o = \Delta_1^0|_o = \emptyset$. Now let $E|_n = (E \setminus E^-) \cup E^+$ and consider a run of Algorithm 2 on $E|_n$; then $\boldsymbol{N}_i^s|_n$, $I_i^s|_n$, $\Delta_i^s|_n$, and $k^s|_n$ are defined analogously. Finally, let k^s be the maximum of $k^s|_o$ and $k^s|_n$ for each s with $0 \leq s \leq S$.

Now consider a run of Algorithm 3 on E, E^-, and E^+, and let $\boldsymbol{N}_i^s$, N_i^s, Δ_o, Δ_n, I_o, I_n, I_{on}, and I_{no} denote the (multi)sets as the computation progresses. We next prove by double induction on s and i that the following invariants hold after line 30 for each s with $0 \leq s \leq S$ and each i with $1 \leq i \leq k^s$.

$$\boldsymbol{N}_j^{s'} = \boldsymbol{N}_j^{s'}|_o \qquad \text{for each } s' \text{ and } j \text{ such that either } s < s' \leq S \text{ and } 1 \leq j \leq k^{s'}, \text{ or } s' = s \text{ and } i < j \leq k^s \quad \text{(A.20)}$$

$$\boldsymbol{N}_j^{s'} = \boldsymbol{N}_j^{s'}|_n \qquad \text{for each } s' \text{ and } j \text{ such that either } 0 \leq s' < s \text{ and } 1 \leq j \leq k^{s'}, \text{ or } s' = s \text{ and } 1 \leq j \leq i \quad \text{(A.21)}$$

$N_o = N_i^s|_o$ (A.22) $\quad \Delta_o = \Delta_i^s|_o$ (A.24) $\quad I_o = I_i^s|_o$ (A.26) $\quad I_{on} = I_o \setminus I_n$ (A.28)

$N_n = N_i^s|_n$ (A.23) $\quad \Delta_n = \Delta_i^s|_n$ (A.25) $\quad I_n = I_i^s|_n$ (A.27) $\quad I_{no} = I_n \setminus I_o$ (A.29)

Invariant (A.21) corresponds to property (a) of Theorem 12 for $s = S$ and $i = k^S$. The base case for $s = 0$ holds trivially since line 19 of Algorithm 3 initialises all sets in the same way as line 9 of Algorithm 2. Hence, we consider an arbitrary $s > 0$ such that (A.20)–(A.29) hold for $s - 1$, and we prove by another induction on i for $1 \leq i \leq k^s$ that (A.20)–(A.29) hold for s.

For the base case, consider $i = 1$; for convenience of notation, let $m = k^{s-1}$. Invariants A.20–A.29 hold for $s - 1$ and they ensure that, just before line 22, we have $\boldsymbol{N}_1^s = \boldsymbol{N}_1^s|_o$, $I_o = I_m^{s-1}|_o$, $I_n = I_m^{s-1}|_n$, $I_{on} = I_o \setminus I_n$, and $I_{no} = I_n \setminus I_o$; thus, line 22 ensures (A.22). Moreover, lines 23–24 update only $\boldsymbol{N}_1^s$, so invariant (A.21) holds for $j > 1$. Furthermore, line 12 of Algorithm 2 computes $\boldsymbol{N}_1^s|_o$ using $\Pi_{\mathsf{nr}}^s \llbracket I_m^{s-1}|_o \rrbracket = \Pi_{\mathsf{nr}}^s \llbracket I_o \rrbracket$, and $\boldsymbol{N}_1^s|_n$ using $\Pi_{\mathsf{nr}}^s \llbracket I_m^{s-1}|_n \rrbracket = \Pi_{\mathsf{nr}}^s \llbracket I_n \rrbracket$; but then, property (A.1) of Lemma 39 ensures that lines 23–24 update $\boldsymbol{N}_1^s$ to $\boldsymbol{N}_1^s|_n$, as required for (A.21). Line 25 ensures (A.23), and line 27 ensures (A.24) and (A.25). Now if $\Delta_o = \Delta_n = \emptyset$ holds in line 28, then $I_1^s|_o = I_m^{s-1}|_o$ and $I_1^s|_n = I_m^{s-1}|_n$, and so $k^s = 1$ and (A.26)–(A.29) hold; hence, consider the case when the loop does not terminate in line 28. But then, line 29 updates I_o and I_n in the same way as line 16 of Algorithm 2, which ensures invariants (A.26) and (A.27). Finally, line 30 updates I_{on} and I_{no} to ensure invariants (A.28) and (A.29).

For the inductive step, consider an arbitrary $i > 1$ such that (A.20)–(A.29) hold for s and $i - 1$. Thus, after line 30 we have $\boldsymbol{N}_i^s = \boldsymbol{N}_i^s|_o$, $\Delta_o = \Delta_{i-1}^s|_o$, $\Delta_n = \Delta_{i-1}^s|_n$, $I_o = I_{i-1}^s|_o$, $I_n = I_{i-1}^s|_n$, $I_{on} = I_o \setminus I_n$, and $I_{no} = I_n \setminus I_o$. Now if the condition in line 31 holds, the computations of Algorithm 2 on $E|_o$ and $E|_n$ coincide—that is, $I_j^s|_o = I_j^s|_n$ and $N_j^s|_o = N_j^s|_n$ hold for each j with $i \leq j \leq N_k^s$; hence, we can exit the loop because line 32 ensures that invariants (A.26) and (A.27) hold. Furthermore, line 12 of Algorithm 2 computes $\boldsymbol{N}_i^s|_o$ using $\Pi_{\mathsf{r}}^s \llbracket I_{i-1}^s|_o \vdots \Delta_{i-1}^s|_o \rrbracket = \Pi_{\mathsf{r}}^s \llbracket I_o \vdots \Delta_o \rrbracket$, and $\boldsymbol{N}_i^s|_n$ using

$\Pi_r^s[\![I_{i-1}^s|_n \vdots \Delta_{i-1}^s|_n]\!] = \Pi_r^s[\![I_n \vdots \Delta_n]\!]$; but then, property (A.2) of Lemma 39 ensures that lines 36–37 update $\boldsymbol{N}_i^s$ to $\boldsymbol{N}_i^s|_n$, as required for (A.21). All remaining invariants hold analogously to the base case.

(Properties b and c) Lemma 11 straightforwardly ensures that, whenever Algorithm 3 considers a rule instance in lines 23 and 36 (resp. lines 24 and 37) for some s and i, then Algorithm 2 considers that rule instance on $E|_o$ (resp. $E|_n$) for the same s and i as well. Furthermore, Algorithm 2 has the nonrepetition property so, on $E|_o$ or $E|_n$, it considers each rule instance in lines 12 and 18 at most once; consequently, Algorithm 3 does not repeatedly consider the same rule instance either. □

Appendix B. Proofs in Section 7

Theorem 25. *Let Π be a program, let λ be a stratification of Π, let E be a dataset, let $I = \mathsf{mat}(\Pi, E)$, and let E^- and E^+ be two datasets. Then,*

(a) *Algorithm 5 correctly updates I to $I' = \mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$;*
(b) *lines 68 and 75 consider rule instances from $\bigcup_{r \in \Pi_r} \mathsf{inst}_r[I]$ without repetition;*
(c) *lines 53, 58, 64, 88, and 94 consider rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r[I']$ without repetition, and moreover the rule instances considered in lines 88, 94, and 64 are also in $\bigcup_{r \in \Pi} \mathsf{inst}_r[I]$;*
(d) *line 81 considers recursive rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r[I]$ without repetition;*
(e) *if branches are never aborted in line 79, then one can safely omit line 63 and one-step rederivation in line 64; and*
(f) *if branches are always aborted in line 79, then the algorithm becomes the same as DRed.*

Due to line 59 of Algorithm 5, without loss of generality we assume that $E^- \subseteq E$ and $E^+ \cap E = \emptyset$. Now let $E|_o = E$ and let $I^s|_o = \emptyset$. Moreover, for each s with $1 \le s \le S$, let $I_0^s|_o, I_1^s|_o, \ldots$ be the sequence of datasets where $I_0^s|_o = I^{s-1}|_o \cup (E|_o \cap \mathsf{Out}^s)$, and $I_i^s|_o = I_{i-1}^s|_o \cup \Pi^s[I_{i-1}^s|_o]$. Clearly, an index k exists at which the sequence reaches the fixpoint (i.e., $I_k^s|_o = I_{k-1}^s|_o$), so let $I^s|_o = I_k^s|_o$. Finally, let $I|_o = I^S|_o$; we clearly have $I|_o = \mathsf{mat}(\Pi, E|_o)$—that is, $I|_o$ is the 'old' materialisation. Now let $E|_n = (E|_o \setminus E^-) \cup E^+$, and let $I_i^s|_n$, $I^s|_n$, and $I|_n$ be defined analogously, so $I|_n$ is the 'new' materialisation.

Now consider a run of Algorithm 5 on $I|_o$, E^-, and E^+. Then, let $D^0 = R^0 = A^0 = \emptyset$ and, for each s with $1 \le s \le S$, let D^s, R^s, and A^s be the values of D, R, and A, respectively, after the loop in lines 60–65 finishes for stratum index s. Note that, during the execution of Algorithm 5, the set I is equal to $I|_o$ up to before line 66. For convenience, let $BDP^s = B^s \cap D^s \cap P^s$. Furthermore, let $\{1, \ldots, t^s\}$ denote the points in the execution of Algorithm 5 for stratum s at which a call to Saturate or Check finishes; then, let $B_0^s = C_0^s = P_0^s = Y_0^s = D_0^s = \Delta_{D_0^s} = \emptyset$ and, for each i with $0 \le i \le t^s$, let B_i^s, C_i^s, P_i^s, Y_i^s, D_i^s, $\Delta_{D_i^s}$ be the 'snapshots' of the respective sets at point i; finally, let B^s, C^s, P^s, and Y^s be the values of these sets when stratum s finishes. Each fact $F \in C^s$ is added to C during a specific call Check(F) that finishes at point i_F; we call i_F the *finishing point for F*. Moreover, note that each call to Check(F) occurs within a top-level call to Check(G) initiated in line 72 for some fact G that may be different from F; we call the point j_F at which this top-level call finishes the *top-level finishing point for F*.

We next prove that properties (B.1) and (B.3) hold for each s with $0 \le s \le S$, and that property (B.2) holds for each s with $1 \le s \le S$; then, property (B.3) for $s = S$ and the way in which I is updated in line 66 of Algorithm 5 imply property (a) of Theorem 25. Property (B.1) says that the difference between the 'old' and 'new' materialisation ends up being deleted, and that only facts from the 'old' materialisation are deleted. Property (B.2) captures the effects of one-step rederivation. Finally, property (B.3) says that Insert correctly computes the 'new' materialisation. We prove (B.1)–(B.3) by induction on s. The base case for $s = 0$ is trivial since all relevant sets are empty. For the inductive step, we consider an arbitrary s with $1 \le s \le S$ such that these properties hold for $s - 1$, and we show that they also hold for s.

$$I^s|_o \setminus I^s|_n \subseteq D^s \subseteq I^s|_o \tag{B.1}$$

$$\mathsf{Out}^s \cap (D^s \setminus P^s) \cap \Big((E|_o \setminus E^-) \cup \Pi^s[I|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})), I|_o \cup A^{s-1}]\Big) \subseteq R^s \subseteq I^s|_n \tag{B.2}$$

$$(I^s|_o \setminus D^s) \cup A^s = I^s|_n \tag{B.3}$$

The proof of these properties is quite lengthy, so we break it into several claims. Moreover, we structure the claims into two parts: B.1 formalises the properties of backward chaining (i.e., procedure Check) and inner forward chaining (i.e., procedure Saturate), where as B.2 deals with deletion, one-step rederivation, and insertion (i.e., procedures DeleteUnproved and Insert, and line 64). Finally, B.3 deals with the remaining claims of the theorem. Before proceeding, we prove the following auxiliary claim that is used in both Appendix B.1 and Appendix B.2.

Claim 40. *For each rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r[I|_o]$, if $\mathsf{b}^+(r') \cap (D^s \setminus A^{s-1}) \neq \emptyset$ or $\mathsf{b}^-(r') \cap (A^{s-1} \setminus D^s) \neq \emptyset$ holds, then $\mathsf{h}(r') \in N_D$ holds in line 70 at some point in the execution of lines 69–76.*

Proof. If $\mathsf{b}^+(r') \cap (D^{s-1} \setminus A^{s-1}) \neq \emptyset$ or $\mathsf{b}^-(r') \cap (A^{s-1} \setminus D^{s-1}) \neq \emptyset$ holds, then $r' \in \mathsf{inst}_r[I|_o \vdots D^{s-1} \setminus A^{s-1}, A^{s-1} \setminus D^{s-1}]$ holds by definition (65), and so $\mathsf{h}(r')$ is added to N_D in line 68. Assume now that $\mathsf{b}^+(r') \cap (D^{s-1} \setminus A^{s-1}) = \mathsf{b}^-(r') \cap (A^{s-1} \setminus D^{s-1})$

$= \emptyset$ holds. Then, $\mathsf{b}^-(r') \cap I|_o = \emptyset$ and the induction assumption for property (B.1) imply $\mathsf{b}^-(r') \cap D^{s-1} = \emptyset$; thus, $\mathsf{b}^-(r') \cap A^{s-1} = \emptyset$ holds as well and it implies $\mathsf{b}^-(r') \cap (I|_o \cup A^{s-1}) = \emptyset$. Moreover, $\mathsf{b}^+(r') \cap (D^s \setminus A^{s-1}) \neq \emptyset$ then ensures $\mathsf{b}^+(r') \cap (D^s \setminus D^{s-1}) \neq \emptyset$. Thus, a point in the execution of lines 69–76 exists when $\Delta_D \cap \mathsf{b}^+(r') \neq \emptyset$ holds in line 75 for the first time for some D and Δ_D. At this point we have $\mathsf{b}^+(r') \cap (D \setminus D^{s-1}) = \emptyset$, which together with $\mathsf{b}^+(r') \cap (D^{s-1} \setminus A^{s-1}) = \emptyset$ implies $\mathsf{b}^+(r') \subseteq I|_o \setminus (D \setminus A^{s-1})$, and thus $r' \in \mathsf{inst}_r\big[I|_o \setminus (D \setminus A^{s-1}), I|_o \cup A^{s-1} \vdots \Delta_D\big]$ holds. Finally, $\Delta_D \subseteq D^s \setminus D^{s-1} \subseteq \mathsf{Out}^s$ ensures $r \in \Pi^s_{\mathsf{r}}$ (i.e., rule r is recursive), so $\mathsf{h}(r')$ is added to N_D in line 75. □

B.1. Properties of backward chaining

The key goal in this part of the proof is to show that backward chaining allows the one-step rederivation to consider only facts that are blocked, which is a consequence of two key properties.

First, Claim 43 says that each fact F that is checked but not blocked or proved remains disproved—that is, the proved status of such F is fixed once the top-level call leading to Check(F) terminates; as a straightforward consequence, Claim 44 says that deleted and proved facts are blocked. Because of this, when the algorithm rederives all deleted and proved facts in line 63, we know that all such facts are blocked and so we can use $B \cap D \cap P$ instead of just $D \cap P$; this, in turn, shows that there is nothing to do if facts are never blocked. Claim 44 is used in the proof Claim 51 from B.2.

Second, Claim 45 shows that each checked fact is eventually proved, blocked, or deleted. Claim 49 from B.2 uses this property to show that one-step rederivation in line 64 can be restricted to blocked facts. Roughy speaking, if a deleted fact F is derivable in one-step derivation, since all deleted and proved facts are rederived in line 63, fact F cannot be proved. Moreover, if we assume that F is not blocked, since F can be derived from facts that are checked but neither blocked nor deleted, F would be proved, which is a contradiction. Hence, F is blocked, and so it is rederived since line 64 considers all such facts.

The proofs of Claims 43, 44, and 45 rely on two auxiliary properties. First, Claim 41 essentially says that Saturate closes the set P under Π^s—that is, P contains each consequence of a rule $r \in \Pi^s$ where the body atoms of r from stratum s (if any) are matched to P and the body atoms of r from previous strata are matched to $I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})$. Second, Claim 42 essentially says that, for each fact F that is checked but not proved or blocked, Check(F) explores all facts from which F is derived in the 'old' materialisation, and that none of these facts are blocked since otherwise line 84 would make F blocked as well.

Claim 41. *For each i with $0 \leq i \leq t^s$, properties (B.4) and (B.5) hold, and each rule instance considered in line 88 or 94 of Algorithm 5 is contained in $\bigcup_{r \in \Pi^s} (\mathsf{inst}_r\big[I^s|_o\big] \cap \mathsf{inst}_r\big[I^s|_n\big])$.*

$$\Big((E|_o \setminus E^-) \cup \Pi^s\big[P^s_i \cup (I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})), I^{s-1}|_o \cup A^{s-1}\big]\Big) \cap C^s_i \subseteq P^s_i \subseteq I^s|_o \cap I^s|_n \tag{B.4}$$

$$\Pi^s_{\mathsf{r}}\big[P^s_i \cup (I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})), I^{s-1}|_o \cup A^{s-1}\big] \setminus C^s_i \subseteq Y^s_i \subseteq I^s|_o \cap I^s|_n \tag{B.5}$$

Proof. We prove the claim by induction on i. For $i = 0$, we have $C^s_0 = P^s_0 = Y^s_0 = \emptyset$; thus, $P^s_0 \cup I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1}) \subseteq \mathsf{Out}^{<s}$, and so $\Pi^s_{\mathsf{r}}\big[P^s_0 \cup I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1}), I^{s-1}|_o \cup A^{s-1}\big] = \emptyset$ since each recursive rule from Π^s_{r} contains at least one positive body atom from stratum s; hence, properties (B.4) and (B.5) both hold. For the inductive step, assume that (B.4) and (B.5) hold for $i-1$. If i does not correspond to an invocation of Saturate, then $C^s_i = C^s_{i-1}$, $P^s_i = P^s_{i-1}$, and $Y^s_i = Y^s_{i-1}$ and so (B.4) and (B.5) hold trivially by the induction assumption; hence, in the rest of this proof we assume that i corresponds to a call Saturate(F) for a fact F.

We prove the right-hand inclusion of (B.4) and (B.5) and the claim about rule instances by a straightforward induction on the construction of P^s_i and Y^s_i: clearly, $E|_o \setminus E^- \subseteq I|_o \cap I|_n$ holds; moreover, for some value of P and Y, a fact G can be added to P or Y in line 88 or 94 using a rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r\big[P \cup (I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})), I^{s-1}|_o \cup A^{s-1}\big]$ where $\mathsf{h}(r') = G$; but then, $\mathsf{b}^+(r') \subseteq P \cup (I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})) \subseteq I^s|_o \cap I^s|_n$ holds by definition (65) and the induction assumption; moreover, definition (65) and property (B.3) by the induction assumption for $s-1$ ensure that $\mathsf{b}^-(r') \cap (I^{s-1}|_o \cup A^{s-1}) = \mathsf{b}^-(r') \cap I^{s-1}|_n = \mathsf{b}^-(r') \cap I^s|_n = \emptyset$ holds; thus, we have $r' \in \mathsf{inst}_r\big[I^s|_o\big] \cap \mathsf{inst}_r\big[I^s|_n\big]$. Finally, rule r is recursive whenever G is added to Y, as required for (B.5).

Finally, we prove that the left-hand inclusions of (B.4) and (B.5) also hold for i. First, consider an arbitrary fact $G \in C^s_i$ such that $G \in (E|_o \setminus E^-) \cup \Pi^s_{\mathsf{nr}}\big[P^s_i \cup I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1}), I^{s-1}|_o \cup A^{s-1}\big]$ holds: all body atoms in Π^s_{nr} are from strata prior to s, so $G \in (E|_o \setminus E^-) \cup \Pi^s_{\mathsf{nr}}\big[I|_o \setminus (D^{s-1} \setminus A^{s-1}), I|_o \cup A^{s-1}\big]$ holds; hence, $G \in C^s_{i-1}$ implies $G \in P^s_i$ by the induction assumption, and $G \in C^s_i \setminus C^s_{i-1}$ implies $G = F$ and thus F is added to P^s_i via lines 89, 91, and 93; either way, property (B.4) holds. Second, consider an arbitrary fact $G \in \Pi^s_{\mathsf{r}}\big[P^s_i \cup I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1}), I^{s-1}|_o \cup A^{s-1}\big]$: then, there exist a rule $r \in \Pi^s_{\mathsf{r}}$ and its instance $r' \in \mathsf{inst}_r\big[P^s_i \cup I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1}), I^{s-1}|_o \cup A^{s-1}\big]$ such that $\mathsf{h}(r') = G$. We have the following two cases.

- $\mathsf{b}^+(r') \cap (P^s_i \setminus P^s_{i-1}) = \emptyset$. But then, we have $G \in \Pi^s_{\mathsf{r}}\big[P^s_{i-1} \cup I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1}), I^{s-1}|_o \cup A^{s-1}\big]$, so the induction assumption ensures that $G \in C^s_{i-1}$ implies $G \in P^s_{i-1}$, and $G \notin C^s_{i-1}$ implies $G \in Y^s_{i-1}$. But then, if $G \in C^s_{i-1} \subseteq C^s_i$, then

$G \in P^s_{i-1} \subseteq P^s_i$ holds; if $G \notin C^s_i$, then $G \notin C^s_{i-1}$ and $G \in Y^s_{i-1} \subseteq Y^s_i$ hold; and if $G \in C^s_i \setminus C^s_{i-1}$, then $G = F$ and $G \in Y^s_{i-1}$ hold, so lines 89, 91, and 93 ensure $F \in P^s_i$. Either way, properties (B.4) and (B.5) hold for G.

- $\mathsf{b}^+(r') \cap (P^s_i \setminus P^s_{i-1}) \neq \emptyset$. Then, a point in the execution of Saturate(F) exists when $\Delta_P \cap \mathsf{b}^+(r') \cap (P^s_i \setminus P^s_{i-1}) \neq \emptyset$ holds after line 93 for the last time for some values of P, Δ_P, Y, and D. Clearly, $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P$ holds at this point, and we clearly have $\mathsf{Out}^{<s} \cap (I|_o \setminus (D \setminus A^{s-1})) = I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})$. Thus, $r' \in \mathsf{inst}_r\big[P \cup (\mathsf{Out}^{<s} \cap (I|_o \setminus (D \setminus A^{s-1}))), I|_o \cup A^{s-1} \mathbin{\vdots} \Delta_P\big]$, so lines 94 and 91 ensure $G \in P^s_i$ if $G \in C^s_i$ or $G \in Y^s_i$ if $G \notin C^s_i$. Thus, properties (B.4) and (B.5) hold for G. □

Claim 42. *For each fact $F \in C^s_{i_F} \setminus (B^s_{i_F} \cup P^s_{i_F})$, rule $r \in \Pi^s_{\mathsf{r}}$, and instance $r' \in \mathsf{inst}_r\big[I|_o \setminus ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus (A^{s-1} \cup B^s_{i_F})), I|_o \cup A^{s-1}\big]$ of r such that $\mathsf{h}(r') = F$, we have $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s_{i_F} \cup (C^s_{i_F} \setminus B^s_{i_F})$.*

Proof. Consider a call to Check(F) for F as stated above. Since $F \in C^s_{i_F}$, the search is not aborted in line 79 and lines 81–85 are executed. Since $F \notin P^s_{i_F}$, the procedure does not return in line 85, and so it considers all rule instances mentioned in the claim—that is, for each recursive rule $r \in \Pi^s_{\mathsf{r}}$, each instance r' of r satisfying the condition in line 81, and each fact $G \in \mathsf{b}^+(r') \cap \mathsf{Out}^s$, a recursive call Check(G) is made in line 83; now since $F \notin B^s_{i_F}$ holds, line 84 is not executed for G and so either $G \in P^s_{i_F}$ or $G \notin B^s_{i_F}$ holds; moreover, the latter clearly ensures $G \in C^s_{i_F}$. Hence, we have $G \in P^s_{i_F} \cup (C^s_{i_F} \setminus B^s_{i_F})$, as required. □

Claim 43. *For each fact $F \in C^s \setminus B^s$ with top-level finishing time point j_F such that $F \notin P^s_{j_F}$ holds, we have $F \notin P^s$.*

Proof. Let Υ be the set containing precisely each $F \in C^s \setminus B^s$ where $F \notin P^s_{j_F}$ and $F \in P^s$. To show that $\Upsilon = \emptyset$ holds, we assume the opposite. Then, there exists a point i corresponding to the execution of Saturate during which, just before line 93, $\Delta_P \cap \Upsilon \neq \emptyset$ holds for the first time for some P and Δ_P. Now choose an arbitrary fact $F \in \Delta_P \cap \Upsilon$. Then, $F \notin P^s_{j_F}$ holds by the definition of Υ and it implies $j_F < i$. Note that F cannot be added to Δ_P via lines 88–89 since that would imply $j_F \geq i$. Hence, F is added to Δ_P via lines 94 and 91, and so a recursive rule $r \in \Pi^s_{\mathsf{r}}$ and its instance

$$r' \in \mathsf{inst}_r\big[P \cup (I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})), I|_o \cup A^{s-1}\big] \tag{B.6}$$

exist such that $\mathsf{h}(r') = F$. Rule $r \in \Pi^s_{\mathsf{r}}$ is recursive so $\emptyset \subsetneq \mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P$ holds. Now let i_F be the finishing point for F. We next argue that $P \cap ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus B^s_{i_F}) = \emptyset$ holds: if some $H \in P \cap ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus B^s_{i_F})$ were to exist, then $H \in D^s_{i_F} \cup \Delta_{D^s_{i_F}}$ implies that H is added to $D^s_{i_F} \cup \Delta_{D^s_{i_F}}$ in line 73 because $H \notin P^s_{j_H}$ holds for $j_H < i_F$; but then, $H \notin B^s_{i_F}$ implies $H \in C^s_{i_F}$, which in turn implies $H \in \Upsilon$; finally, $H \in P$ implies that H is added to P before F, which contradicts our assumption that F is the first fact from Υ that is added to P. Now $P \subseteq I|_o$ holds due to (B.4), which together with $P \cap ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus B^s_{i_F}) = \emptyset$ and (B.6) ensures that $r' \in \mathsf{inst}_r\big[I|_o \setminus ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus (A^{s-1} \cup B^s_{i_F})), I|_o \cup A^{s-1}\big]$ holds. Finally, $F \in C^s \setminus B^s$ clearly implies $F \in C^s_{i_F} \setminus B^s_{i_F}$, so together with $F \notin P^s_{i_F}$, Claim 42 ensures $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s_{i_F} \cup (C^s_{i_F} \setminus B^s_{i_F})$.

At the same time, (B.4) holds for j_F, and so $F \notin P^s_{j_F}$ implies $r' \notin \mathsf{inst}_r\big[P^s_{j_F} \cup (I^{s-1}|_o \setminus (D^{s-1} \setminus A^{s-1})), I^s|_o \cup A^{s-1}\big]$; thus, there exists a fact $G \in \mathsf{b}^+(r') \cap \mathsf{Out}^s$ such that $G \in P \setminus P^s_{j_F}$ holds. Clearly, we also have $G \notin P^s_{i_F}$, and so by the observation from the previous paragraph we have $G \in C^s_{i_F} \setminus B^s_{i_F}$. Now $G \in C^s_{i_F}$ line 78 ensure that G is not added to B at a later point, and so we have $G \in C^s \setminus B^s$. Finally, $G \in C^s_{i_F}$ also ensures $j_G \leq j_F$, which together with $G \notin P^s_{j_F}$ implies $G \notin P^s_{j_G}$. Consequently, $G \in \Upsilon$ holds and G is added to P before F, which contradicts our assumption that F is the first such fact. □

Claim 44. $D^s \cap P^s \subseteq B^s$.

Proof. Consider an arbitrary fact $F \in D^s \setminus B^s$. Then, F is added to D via line 73 because of $F \notin P^s_{j_F}$, and moreover $F \notin B^s$ ensures $F \in C^s$. But then, Claim 43 implies $F \notin P^s$ as well. □

Claim 45. $C^s \subseteq P^s \cup D^s \cup B^s$.

Proof. We prove by induction on the sequence $I^s_0|_o, I^s_1|_o, \ldots$ of datasets used to construct $I^s|_o$ that (B.7) holds for each i; this implies our claim since each fact considered in line 71 or 82 is contained in $I^s|_o \cap \mathsf{Out}^s$, and so $C^s \subseteq I^s|_o \cap \mathsf{Out}^s$ holds.

$$C^s \cap I^s_i|_o \subseteq P^s \cup D^s \cup B^s \tag{B.7}$$

For the base case, consider an arbitrary fact $F \in C^s \cap I^s_0|_o = C^s \cap E|_o$: if $F \in E^- \cap \mathsf{Out}^s$, then F is added to N_D in line 68 and hence $F \in D^s \cup P^s$ eventually holds; otherwise, we have $F \in E|_o \setminus E^-$, so property (B.4) implies $F \in P^s$. For the inductive case, assume that $I^s_{i-1}|_o$ satisfies (B.7), and consider an arbitrary fact $F \in C^s \cap I^s_i|_o$. Then, a rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r\big[I|_o\big]$ exist such that $\mathsf{h}(r') = F$. Now if either $\mathsf{b}^+(r') \cap (D^s \setminus A^{s-1}) \neq \emptyset$ or $\mathsf{b}^-(r') \cap (A^{s-1} \setminus D^s) \neq \emptyset$ holds,

then Claim 40 ensures that $F \in N_D$ holds in line 71 at some point in the execution of lines 69–76, and so eventually $F \in D^s \cup P^s$ holds. The only remaining case is when $\mathsf{b}^+(r') \cap (D^s \setminus A^{s-1}) = \mathsf{b}^-(r') \cap (A^{s-1} \setminus D^s) = \emptyset$ holds. Set D grows monotonically so $D^s_{i_F} \cup \Delta_D{}^s_{i_F} \subseteq D^s$ holds, and we have $\mathsf{b}^+(r') \subseteq I|_o \setminus ((D^s_{i_F} \cup \Delta_D{}^s_{i_F}) \setminus A^{s-1})$; moreover, we also have $\mathsf{b}^-(r') \cap (I|_o \cup A^{s-1}) = \emptyset$. We next assume for the sake of a contradiction that $F \notin P^s \cup B^s$ holds. Then, Claim 42 ensures $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s_{i_F} \cup (C^s_{i_F} \setminus B^s_{i_F})$, and so (B.7) ensures by the induction assumption that $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq D^s \cup P^s$ holds; together with $\mathsf{b}^+(r') \cap (D^s \setminus A^{s-1}) = \emptyset$ this implies $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s$. But then, (B.4) implies $F \in P^s$, which contradicts our assumption that $F \notin P^s$ holds. □

B.2. Claims about deletion, one-step rederivation, and insertion

In this section we prove Properties B.1–B.3 via a series of six claims that essentially prove soundness and completeness of deletion in line 62, one-step rederivation in lines 63 and 64, and insertion in line 65. The soundness claims are relatively straightforward and are proved by induction on rule application, whereas completeness claims are more involved. In particular, the proof of Claim 51 uses Claim 44 to ensure that we can consider only blocked facts in line 63, and the proof of Claim 49 uses Claim 45 to ensure that we can consider only blocked facts in one-step rederivation in line 64.

Claim 46. *The right-hand inclusion of* (B.1) *holds.*

Proof. For each rule $r \in \Pi$, Algorithm 5 considers in lines 68 and 75 only instances of r that are contained $\mathsf{inst}_r\big[I|_o\big]$. Thus, the claim holds by a straightforward induction on the construction of the set D in Algorithm 5. □

Claim 47. *The left-hand inclusion of* (B.1) *holds.*

Proof. We show by induction that (B.8) holds for each i.

$$I^s_i|_o \setminus I^s|_n \subseteq D^s \tag{B.8}$$

For the base case, note that $I^s_0|_o = I^{s-1}|_o \cup (E|_o \cap \mathsf{Out}^s)$ and that $I^{s-1}|_o \setminus I^{s-1}|_n \subseteq D^{s-1}$ holds for $s-1$ by the induction assumption. Now consider an arbitrary fact $F \in E|_o \cap \mathsf{Out}^s$ such that $F \notin I^s|_n$ holds. Then, the latter ensures $F \notin E|_o \setminus E^-$, which implies $F \in E^-$ and so F is added to N_D in line 68. We next show that F is added to D in line 76: $F \notin I^s|_n$ and the contrapositive of the right-hand inclusion of (B.4) imply $F \notin P^s$, which ensures $F \notin P^s_{j_F}$ and so the condition in line 73 is satisfied. Hence, $F \in D^s$ holds, as required.

For the inductive step, assume that $I^s_{i-1}|_o$ satisfies (B.8) for $i > 0$, and consider arbitrary $F \in I^s_i|_o \setminus I^s|_n$. If $F \in I^s_{i-1}|_o$, then $F \in D^s$ holds by the induction assumption. Otherwise, a rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r\big[I^s_{i-1}|_o\big]$ exist such that $\mathsf{h}(r') = F$. Definition (65) ensures $\mathsf{b}^+(r') \subseteq I^s_{i-1}|_o \subseteq I|_o$ and $\mathsf{b}^-(r') \cap I^s_{i-1}|_o = \emptyset$, and $\mathsf{b}^-(r') \subseteq \mathsf{Out}^{<s}$ implies $\mathsf{b}^-(r') \cap I^{s-1}|_o = \mathsf{b}^-(r') \cap I|_o = \emptyset$. Finally, $F \notin I^s|_n$ implies $r' \notin \mathsf{inst}_r\big[I^s|_n\big]$, so by definition (65) we have one of the following two possibilities.

- $\mathsf{b}^+(r') \not\subseteq I^s|_n$. Thus, a fact $G \in \mathsf{b}^+(r')$ exists such that $G \in I^s_{i-1}|_o \setminus I^s|_n$ holds. The induction assumption for (B.8) implies $G \in D^s$, and $G \notin I^s|_n$ implies $G \notin I^{s-1}|_n$, so the induction assumption for (B.3) ensures $G \notin A^{s-1}$; hence, $G \in D^s \setminus A^{s-1}$.
- $\mathsf{b}^-(r') \cap I^s|_n = \mathsf{b}^-(r') \cap I^{s-1}|_n \neq \emptyset$. Thus, a fact $G \in \mathsf{b}^-(r')$ exists such that $G \in I^{s-1}|_n \setminus I^{s-1}|_o$ holds; but then, the right-hand inclusion of (B.1) implies $G \notin D^{s-1}$, and the induction assumption for (B.3) implies $G \in A^{s-1}$; hence, $G \in A^{s-1} \setminus D^{s-1}$.

Either way, Claim 40 ensures that F is added to N_D in line 68 or 75, and so in the same way as in the proof of the base case we have $F \in D^s$, as required. □

Claim 48. *The right-hand inclusion of property* (B.2) *holds, and each rule instance considered in line 64 of Algorithm 5 is contained in* $\bigcup_{r \in \Pi^s} (\mathsf{inst}_r\big[I^s|_o\big] \cap \mathsf{inst}_r\big[I^s|_n\big])$.

Proof. The left-hand side of (B.1) is equivalent to $I^s|_o \setminus D^s \subseteq I^s|_n$, and $A^{s-1} \subseteq I^{s-1}|_n$ holds for $s-1$ by the induction assumption for (B.3), so $(I^s|_o \setminus D^s) \cup A^{s-1} \subseteq I^s|_n$ holds. We next consider ways by which a fact F is added to R^s in line 64. If $F \in E|_o \setminus E^-$, then $F \in I^s|_n$ clearly holds; and if $F \in Y^s$, then $F \in I^s|_n$ holds by property (B.5). Otherwise, a rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r\big[I|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})), I|_o \cup A^{s-1}\big]$ exist where $\mathsf{h}(r') = F$. Definition (65) and the right-hand side of (B.4) ensure

$$\mathsf{b}^+(r') \subseteq I^s|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})) \subseteq (I^s|_o \setminus D^s) \cup BDP^s \cup A^{s-1} \subseteq (I^s|_o \setminus D^s) \cup P^s \cup A^{s-1} \subseteq I^s|_n.$$

Definition (65) also ensures $\mathsf{b}^-(r') \cap (I^{s-1}|_o \cup A^{s-1}) = \emptyset$, and the induction assumption for $s-1$ for property (B.3) implies $I^{s-1}|_n = (I^{s-1}|_o \setminus D^{s-1}) \cup A^{s-1} \subseteq I^{s-1}|_o \cup A^{s-1}$; these two observations imply $\mathsf{b}^-(r') \cap I^{s-1}|_n = \mathsf{b}^-(r') \cap I^s|_n = \emptyset$. Consequently, we have $r' \in \mathsf{inst}_r\big[I^s|_n\big]$, so $F \in I^s|_n$ holds, as required. Finally, $r' \in \mathsf{inst}_r\big[I^s|_o\big]$ holds obviously. □

Claim 49. *The left-hand inclusion of property* (B.2) *holds.*

Proof. Consider an arbitrary fact $F \in \mathsf{Out}^s \cap (D^s \setminus P^s) \cap ((E|_o \setminus E^-) \cup \Pi^s\big[I|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})), I|_o \cup A^{s-1}\big])$; we show that $F \in B^s$ holds and so F is added to R^s in line 64 of Algorithm 5. By our assumption, we have $F \in D^s$ and $F \notin P^s$. Now for the sake of a contradiction, assume $F \notin B^s$. Then, $F \in D^s$ and $F \notin B^s$ imply $F \in C^s$, and so the condition in line 88 is checked for F, and $F \notin P^s$ implies $F \notin (E|_o \setminus E^-) \cup \Pi^s_{\mathsf{nr}}\big[I|_o \setminus (D^{s-1} \setminus A^{s-1}), I|_o \cup A^{s-1}\big]$; since $BDP^s \subseteq \mathsf{Out}^s$ and all body atoms in the rules in Π^s_{nr} are from previous strata, we have $F \notin (E|_o \setminus E^-) \cup \Pi^s_{\mathsf{nr}}\big[I|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})), I|_o \cup A^{s-1}\big]$. Thus, the only remaining possibility is that a recursive rule $r \in \Pi^s_{\mathsf{r}}$ and its instance $r' \in \mathsf{inst}_r\big[I|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})), I|_o \cup A^{s-1}\big]$ exist such that $\mathsf{h}(r') = F$. Thus, definition (65) together with the fact that $D^s_{i_F} \cup \Delta_{D^s_{i_F}} \subseteq D^s$ ensures

$$\mathsf{b}^+(r') \subseteq I|_o \setminus (D^s \setminus (BDP^s \cup A^{s-1})) \subseteq I|_o \setminus ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus (BDP^s \cup A^{s-1})).$$

To see that $\mathsf{b}^+(r') \subseteq I|_o \setminus ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus (A^{s-1} \cup B^s_{i_F}))$ holds as well, consider an arbitrary fact $G \in \mathsf{b}^+(r')$. If $G \notin D^s_{i_F} \cup \Delta_{D^s_{i_F}}$, then we clearly have $G \in I|_o \setminus ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus (A^{s-1} \cup B^s_{i_F}))$. Otherwise, we have $G \in D^s_{i_F} \cup \Delta_{D^s_{i_F}}$ and $G \in BDP^s \cup A^{s-1}$; but then, $G \in BDP^s$ implies $G \in B^s$, and no fact is added to B^s after being added to $D^s_{i_F} \cup \Delta_{D^s_{i_F}}$, which ensures that $G \in B^s_{i_F}$ holds; consequently, we have $G \in I|_o \setminus ((D^s_{i_F} \cup \Delta_{D^s_{i_F}}) \setminus (A^{s-1} \cup B^s_{i_F}))$. In addition, $F \in C^s \setminus (B^s \cup P^s)$ implies $F \in C^s_{i_F} \setminus (B^s_{i_F} \cup P^s_{i_F})$, which together with Claim 42 ensures $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s_{i_F} \cup (C^s_{i_F} \setminus B^s_{i_F})$, and this clearly also implies $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s \cup (C^s \setminus B^s)$. Moreover, by our assumption on r', we have $\mathsf{b}^+(r') \cap \mathsf{Out}^s \cap (D^s \setminus BDP^s) = \emptyset$. We next show that these properties imply $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s$: if $G \in \mathsf{b}^+(r') \cap \mathsf{Out}^s$ such that $G \notin P^s$ were to exist, then $G \in C^s \setminus B^s$ holds so Claim 45 implies $G \in D^s$; moreover, $G \notin B^s$ ensures $G \notin BDP^s$, which in turn implies $G \in D^s \setminus BDP^s$ that leads to a contradiction. Thus, $\mathsf{b}^+(r') \cap \mathsf{Out}^s \subseteq P^s$ holds, so property (B.4) implies $F \in P^s$, which contradicts our assumption that $F \notin P^s$ holds. □

Claim 50. *The $\subseteq$ direction of property* (B.3) *holds; each rule instance considered in line 53 or 58 of Algorithm 5 is contained in $\bigcup_{r \in \Pi^s} \mathsf{inst}_r\big[I^s|_n\big]$; and $I^s|_o \cap A^s \subseteq D^s$ holds.*

Proof. We prove by induction on the construction of A in Insert that $(I^s|_o \setminus D^s) \cup A \subseteq I^s|_n$ and $I^s|_o \cap A \subseteq D^s$ hold. We first consider the base case. Set A is equal to $BDP^s \cup A^{s-1}$ before the loop in lines 54–58; thus, property (B.3) is equivalent to $(I^s|_o \setminus D^s) \cup BDP^s \cup A^{s-1} \subseteq I^s|_n$; now $(I^s|_o \setminus D^s) \cup A^{s-1} \subseteq I^s|_n$ holds as in the proof of Claim 48, whereas $BDP^s \subseteq I^s|_n$ holds by property (B.4). Moreover, $I^s|_o \cap A^{s-1} = I^{s-1}|_o \cap A^{s-1} \subseteq D^{s-1} \subseteq D^s$ holds by the induction assumption, and $BDP^s \subseteq D^s$ holds by the definition of BDP^s, so therefore $I^s|_o \cap (BDP^s \cup A^{s-1}) \subseteq D^s$ holds.

For the inductive step, we assume that $(I^s|_o \setminus D^s) \cup A \subseteq I^s|_n$ and $I^s|_o \cap A \subseteq D^s$ hold, and we consider ways in which Algorithm 5 can add a fact F to A. If $F \in E^+ \cap \mathsf{Out}^s$, then $F \in I^s|_n$ clearly holds. Moreover, if $F \in R^s$, then $F \in I^s|_n$ holds by (B.2). Otherwise, F is derived in line 53 or 58, so a rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r\big[(I|_o \setminus D^s) \cup A\big]$ exist such that $\mathsf{h}(r') = F$. But then, definition (65) ensures $\mathsf{b}^+(r') \subseteq (I^s|_o \setminus D^s) \cup A \subseteq I^s|_n$ and $\mathsf{b}^-(r') \cap ((I^s|_o \setminus D^s) \cup A) = \emptyset$, which together with $\mathsf{b}^-(r') \subseteq \mathsf{Out}^{<s}$ implies $\mathsf{b}^-(r') \cap I^s|_n = \emptyset$. Consequently, we have $r' \in \mathsf{inst}_r\big[I^s|_n\big]$, and so $F \in I^s|_n$ holds, as required. Finally, line 55 ensures that F is added to A only if $F \notin I^s|_o \setminus D^s$ holds, which clearly ensures $I^s|_o \cap (A \cup \{F\}) \subseteq D^s$, as required. □

Claim 51. *The $\supseteq$ direction of* (B.3) *holds*

Proof. We show by induction that (B.9) holds for each i.

$$(I^s|_o \setminus D^s) \cup A^s \supseteq I^s_i|_n \tag{B.9}$$

For the base case, we have $I^s_0|_n = I^{s-1}|_n \cup (E|_n \cap \mathsf{Out}^s) = (I^{s-1}|_o \setminus D^{s-1}) \cup A^{s-1} \cup (E|_n \cap \mathsf{Out}^s)$ by the induction assumption for (B.3). Clearly, $I^{s-1}|_o \setminus D^{s-1}|_o \subseteq I^s|_o \setminus D^s|_o$ and $A^{s-1} \subseteq A^s$ hold. Consider arbitrary $F \in E|_n \cap \mathsf{Out}^s$. If $F \in E^+$, Algorithm 5 ensures $F \in (I^s|_o \setminus D^s) \cup A^s$ via lines 53, 55, and 57. If $F \notin E^+$, then $F \in E|_n$ implies $F \in E|_n \setminus E^+ = E|_o \setminus E^-$, and so we have $F \in I^s|_o$, and we have three possibilities: if $F \notin D^s$, then $F \in (I^s|_o \setminus D^s) \cup A^s$ clearly holds; if $F \in D^s \cap P^s$, then Claim 44 implies $F \in B^s$, and so Algorithm 5 ensures $F \in A^s \subseteq (I^s|_o \setminus D^s) \cup A^s$ via line 63; finally, if $F \in D^s \setminus P^s$, then (B.2) implies $F \in R^s$, so Algorithm 5 ensures $F \in (I^s|_o \setminus D^s) \cup A^s$ via lines 53, 55, and 57.

For the inductive step, assume that $I^s_{i-1}|_n$ satisfies (B.9) for $i > 0$, and consider arbitrary $F \in I^s_i|_n$. If $F \in I^s_{i-1}|_n$, then (B.9) holds by the induction assumption. Otherwise, a rule $r \in \Pi^s$ and its instance $r' \in \mathsf{inst}_r\big[I^s_{i-1}|_n\big]$ exist where $\mathsf{h}(r') = F$. Definition (65) ensures $\mathsf{b}^+(r') \subseteq I^s_{i-1}|_n \subseteq (I^s|_o \setminus D^s) \cup A^s \subseteq (I|_o \setminus D^s) \cup A^s$, where (B.9) ensures the next-to-last inclusion by induction assumption. In addition, definition (65) also ensures $\mathsf{b}^-(r') \cap I^s_{i-1}|_n = \emptyset$, and $\mathsf{b}^-(r') \subseteq \mathsf{Out}^{<s}$ and the induction assumption for (B.9) clearly imply $\mathsf{b}^-(r') \cap ((I|_o \setminus D^s) \cup A^s) = \emptyset$, and so $\mathsf{b}^-(r') \cap A^{s-1} = \emptyset$. Let $A' = BDP^s \cup A^{s-1}$; we consider the following cases.

- $\mathsf{b}^+(r') \cap (A^s \setminus A') \neq \emptyset$. Facts in $A^s \setminus A'$ are added to A via Δ_A and line 57, so a point in the execution of Algorithm 5 exists where $\mathsf{b}^+(r') \cap (A^s \setminus A') \cap \Delta_A \neq \emptyset$ holds in line 58 for the last time for A and Δ_A. Since $\Delta_A \subseteq A$ holds at this point, we have $\mathsf{b}^+(r') \subseteq (I|_o \setminus D^s) \cup A$; moreover, $A \subseteq A^s$ ensures $\mathsf{b}^-(r') \cap ((I|_o \setminus D^s) \cup A) = \emptyset$. But then, $r' \in \mathsf{inst}_r\big[(I|_o \setminus D^s) \cup A \mathrel{\vdots} \Delta_A\big]$ holds, so $F \in N_A$ holds after line 58, and Algorithm 5 ensures $F \in (I^s|_o \setminus D^s) \cup A^s$ via lines 55 and 57.
- $\mathsf{b}^+(r') \cap (A^s \setminus A') = \emptyset$, so $\mathsf{b}^+(r') \subseteq (I|_o \setminus D^s) \cup A'$ holds. We have the following possibilities.
  - $\mathsf{b}^+(r') \cap (A' \setminus D^s) \neq \emptyset$ or $\mathsf{b}^-(r') \cap I|_o \neq \emptyset$. In the latter case, $\mathsf{b}^-(r') \cap (I|_o \setminus D^s) = \emptyset$ and $\mathsf{b}^-(r') \cap A' = \emptyset$ clearly imply $\mathsf{b}^-(r') \cap (D^s \setminus A') \neq \emptyset$. Thus, definition (65) ensures that $r' \in \mathsf{inst}_r\big[(I|_o \setminus D^s) \cup A' \mathrel{\vdots} A' \setminus D^s, D^s \setminus A'\big]$ holds as well. Hence, $F \in N_A$ holds after line 53, so Algorithm 5 ensures $F \in (I^s|_o \setminus D^s) \cup A^s$ via lines 55 and 57.
  - $\mathsf{b}^+(r') \cap (A' \setminus D^s) = \mathsf{b}^-(r') \cap I|_o = \emptyset$; but then, $\mathsf{b}^-(r') \cap (I|_o \cup A') = \emptyset$ holds as well. Next, we argue that each fact $G \in \mathsf{b}^+(r') \subseteq (I|_o \setminus D^s) \cup A'$ satisfies $G \in I|_o \setminus (D^s \setminus A')$: this is clear if $G \in I|_o \setminus D^s$; and if $G \in A'$, then $G \notin A' \setminus D^s$ ensures $G \in D^s$, and the right-hand inclusion of (B.1) ensures $G \in I^s|_o \subseteq I|_o$. But then, definition (65) ensures $r' \in \mathsf{inst}_r\big[I|_o \setminus (D^s \setminus A'), I|_o \cup A'\big]$, which implies $F \in I^s|_o$. Now if $F \notin D^s$, then $F \in (I^s|_o \setminus D^s) \cup A^s$ clearly holds. Moreover, if $F \in D^s \setminus P^s$, then (B.2) ensures $F \in R^s$; thus, $F \in N_A$ holds after line 68, and so Algorithm 5 ensures that $F \in (I^s|_o \setminus D^s) \cup A^s$ holds via lines 55 and 57. Finally, if $F \in D^s \cap P^s$, then Claim 44 ensures $F \in B^s$, and so line 63 ensures $F \in A^s$, which in turn ensures $F \in (I^s|_o \setminus D^s) \cup A^s$. □

B.3. Properties (b)–(f) of Theorem 25

We finally prove the remaining properties of Theorem 25. Soundness claims from B.2 already show that Algorithm 5 considers rule instances from 'old' and 'new' materialisation as specified in the theorem, and the key remaining issue is to show that rule instances are considered at most once in the deletion phase, as well as in the backward chaining, one-step rederivation, and the insertion phases combined.

Claim 52. *Property (b) of Theorem 25 holds.*

Proof. Lines 68 and 75 of Algorithm 5 clearly consider only the rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[I|_o\big]$. Furthermore, each rule instance considered in line 68 either has a positive body atom in $D^{s-1} \setminus A^{s-1}$ or a negative body atom in $A^{s-1} \setminus D^{s-1}$, whereas each rule instance considered in line 75 has no positive body atoms in $D^{s-1} \setminus A^{s-1}$ and no negative body atom in A^{s-1}; thus, a rule instance considered in line 68 is never considered in line 75. Finally, let r be an arbitrary rule instance considered in line 75; by definition (65) there exists a fact $G \in \mathsf{b}^+(r) \cap \Delta_D$ that is added to D in line 76. We clearly have $G \in \mathsf{Out}^s$, so $G \notin A^{s-1}$ holds, and therefore from this point onwards we have $G \notin I|_o \setminus (D \setminus A^{s-1})$; hence, r cannot be considered again in line 75. □

Claim 53. *Property (c) of Theorem 25 holds.*

Proof. Claim 50 proves that Algorithm 5 considers in lines 53 and 58 only the rule instances from $\bigcup_{r \in \Pi} \mathsf{inst}_r\big[I|_n\big]$, and Claims 48 and 41 prove that the algorithm considers in lines 64, 88, and 94 only the rule instances from $\bigcup_{r \in \Pi} (\mathsf{inst}_r\big[I|_o\big] \cap \mathsf{inst}_r\big[I|_n\big])$. We next show that the algorithm does not consider the same rule instance more than once.

First we show that rule instances are considered without repetition in lines 53 and 58. For r' a rule instance considered in line 58, $\mathsf{b}^+(r')$ contains at least one fact in Δ_A, and line 55 ensures that each fact in Δ_A is freshly added to $(I^s|_n \setminus D^s) \cup A$; hence, r' could not have been previously considered in line 53 or 58.

Next we show that there is no overlap between the rule instances considered in line 53 or 58 and the ones considered in line 64, 88, or 94. To this end, let $A' = BDP^s \cup A^{s-1}$; Claim 44 ensures that $D^s \cap P^s = BDP^s$, and $P^s \subseteq I^s|_o$ by property (B.4), which jointly imply $P^s \subseteq I^s|_o \setminus (D^s \setminus A')$. Hence, $r' \in \mathsf{inst}_r\big[I^s|_o \setminus (D^s \setminus A'), I|_o \cup A'\big]$ holds for each instance r' of a rule $r \in \Pi^s$ considered in line 64, 88, or 94. In contrast, we next show that, for each instance r' of a rule $r \in \Pi^s$ considered in line 53 or 58, we have $r' \notin \mathsf{inst}_r\big[I^s|_o \setminus (D^s \setminus A'), I|_o \cup A'\big]$. We consider the following possibilities.

- Instance r' is considered in line 53 and $\mathsf{b}^+(r') \cap (A' \setminus D^s) \neq \emptyset$. Thus, there exists a fact $G \in \mathsf{b}^+(r')$ such that $G \in A'$ and $G \notin D^s$; but then, Claim 50 ensures $I^s|_o \cap A^s \subseteq D^s$, which ensures $G \notin I|_o$, and so $\mathsf{b}^+(r') \not\subseteq I|_o \setminus (D^s \setminus A')$.
- Instance r' is considered in line 53 and $\mathsf{b}^-(r') \cap (D^s \setminus A') \neq \emptyset$. Since $\mathsf{b}^-(r') \subseteq \mathsf{Out}^{<s}$, we have $\mathsf{b}^-(r') \cap (D^{s-1} \setminus A') \neq \emptyset$, and so there exists a fact $G \in D^{s-1} \subseteq I|_o$ where the last inclusion holds by (B.1); hence, $\mathsf{b}^-(r') \cap (I|_o \cup A') \neq \emptyset$.
- Instance r' is considered in line 58 and there exists a fact $G \in \mathsf{b}^+(r') \cap (A^s \setminus A')$. We next prove $\mathsf{b}^+(r') \not\subseteq I|_o \setminus (D^s \setminus A')$, which clearly holds if $G \notin I|_o$. Now if $G \in I|_o$, then Claim 50 ensures $I^s|_o \cap A^s \subseteq D^s$ and so $G \in D^s$ holds, which together with $G \notin A'$ implies $G \notin I|_o \setminus (D^s \setminus A')$.

Finally, we show that lines 64, 88, and 94 do not repeat rule instances either.

- Let r' be an instance of a rule r considered in line 88 and let $F = \mathsf{h}(r')$. Line 78 ensures that $\textsc{Saturate}(F)$ is called for F at most once, so r' is considered in line 88 at most once; moreover, r is not recursive and so it is not considered in

line 94. Moreover, F is added to P^s via lines 89 and 93, and so $F \notin D^s \setminus P^s$, which ensures that r' is not considered in line 64.

- Let r' be an instance of a rule r considered in line 94 and let $F = \mathsf{h}(r')$. Lines 91 and 93 ensure that each fact is added to Δ_P at most once, so line 94 also considers each instance of the recursive rules at most once. Now $F \in P^s$ implies $F \notin D^s \setminus P^s$, and $F \notin P^s$ implies $F \in Y^s$ by properties (B.4) and (B.5); either way, r' is not considered in line 64. □

Claim 54. *Properties (d)–(f) of Theorem 25 hold.*

Proof. For property (d), note that, for each fact F, the condition in line 78 becomes true at most once during the algorithm's execution, so each rule instance r' in the loop in lines 81–85 satisfies $r' \in \bigcup_{r \in \Pi_r} \mathsf{inst}_r\big[I|_o\big]$ and it is considered at most once. Property (e) holds trivially since, if no branches are aborted, then we have $B^s = \emptyset$ so no fact is added to A^s in line 63 or to R^s in line 64. Finally, to show (f), observe that aborting every branch is equivalent to never calling Check(F) in line 72, so DeleteUnproved becomes equivalent to Overdelete with $P^s = Y^s = \emptyset$ and $B^s = D^s$; moreover, $P^s = \emptyset$ ensures that line 63 has no effect, whereas $B^s = D^s$ and $Y^s = \emptyset$ makes line 64 equivalent to line 42. □

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley, 1995.
[2] B. Motik, B.C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, OWL 2 Web Ontology Language: Profiles, 2009, W3C Recommendation.
[3] B. Luteberget, C. Johansen, M. Steffen, Rule-based consistency checking of railway infrastructure designs, in: E. Ábrahám, M. Huisman (Eds.), Proc. of the 12th Int. Conf. on Integrated Formal Methods, IFM 2016, in: LNCS, vol. 9681, Springer, 2016, pp. 491–507.
[4] R. Piro, Y. Nenov, B. Motik, I. Horrocks, P. Hendler, S. Kimberly, M. Rossman, Semantic technologies for data analysis in health care, in: P.T. Groth, E. Simperl, A.J.G. Gray, M. Sabou, M. Krötzsch, F. Lécué, F. Flöck, Y. Gil (Eds.), Proc. of the 15th Int. Semantic Web Conf., Part II, ISWC 2016, in: LNCS, vol. 9982, Springer, 2016, pp. 400–417.
[5] M. Aref, Datalog for enterprise software: from industrial applications to research (invited talk), in: M.V. Hermenegildo, T. Schaub (Eds.), Technical Communications of the 26th Int. Conf. on Logic Programming, ICLP 2010, in: LIPIcs, vol. 7, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2010, p. 1.
[6] B.T. Loo, T. Condie, M.N. Garofalakis, D.E. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, I. Stoica, Declarative networking, Commun. ACM 52 (2009) 87–95.
[7] Z. Wu, G. Eadon, S. Das, E.I. Chong, V. Kolovski, M. Annamalai, J. Srinivasan, Implementing an inference engine for RDFS/OWL constructs and user-defined rules in oracle, in: G. Alonso, J.A. Blakeley, A.L.P. Chen (Eds.), Proc. of the 24th Int. Conf. on Data Engineering, ICDE 2008, IEEE Computer Society, 2008, pp. 1239–1248.
[8] P. Gärdenfors (Ed.), Belief Revision, Cambridge Tracts in Theoretical Computer Science, vol. 29, Cambridge University Press, 2003.
[9] H.M. Dewan, D. Ohsie, S.J. Stolfo, O. Wolfson, S.D. Silva, Incremental database rule processing in PARADISER, J. Intell. Inf. Syst. 1 (1992) 177–209.
[10] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Incremental update of datalog materialisation: the backward/forward algorithm, in: B. Bonet, S. Koenig (Eds.), Proc. of the 29th AAAI Conf. on Artificial Intelligence, AAAI 2015, AAAI Press, 2015, pp. 1560–1568.
[11] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (2001) 374–425.
[12] J. Urbani, A. Margara, C.J.H. Jacobs, F. van Harmelen, H.E. Bal, DynamiTE: parallel materialization of dynamic RDF data, in: H. Alani, L. Kagal, A. Fokoue, P.T. Groth, C. Biemann, J.X. Parreira, L. Aroyo, N.F. Noy, C. Welty, K. Janowicz (Eds.), Proc. of the 12th Int. Semantic Web Conf., ISWC 2013, in: LNCS, vol. 8218, Springer, 2013, pp. 657–672.
[13] A. Gupta, I.S. Mumick, Maintenance of materialized views: problems, techniques, and applications, IEEE Data Eng. Bull. 18 (1995) 3–18.
[14] D. Vista, View maintenance in relational and deductive databases by incremental query evaluation, in: J.E. Botsford, A. Gawman, W.M. Gentleman, E. Kidd, K.A. Lyons, J. Slonim, J.H. Johnson (Eds.), Proc. of the 1994 Conf. of the Centre for Advanced Studies on Collaborative Research, CASCON 1994, IBM, 1994, pp. 70–76.
[15] J.A. Blakeley, P. Larson, F.W. Tompa, Efficiently updating materialized views, in: C. Zaniolo (Ed.), Proc. of the ACM SIGMOD Int. Conf. on Management of Data, SGIGMOD 1986, ACM Press, 1986, pp. 61–71.
[16] E.N. Hanson, A performance analysis of view materialization strategies, in: U. Dayal, I.L. Traiger (Eds.), Proc. of the ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 1987, ACM Press, 1987, pp. 440–453.
[17] S. Ceri, J. Widom, Deriving production rules for incremental view maintenance, in: G.M. Lohman, A. Sernadas, R. Camps (Eds.), Proc. of the 17th Int. Conf. on Very Large Data Bases, VLDB 1991, Morgan Kaufmann, 1991, pp. 577–589.
[18] N. Roussopoulos, An incremental access method for viewcache: concept, algorithms, and cost analysis, ACM Trans. Database Syst. 16 (1991) 535–563.
[19] X. Qian, G. Wiederhold, Incremental recomputation of active relational expressions, IEEE Trans. Knowl. Data Eng. 3 (1991) 337–341.
[20] T. Griffin, L. Libkin, H. Trickey, An improved algorithm for the incremental recomputation of active relational expressions, IEEE Trans. Knowl. Data Eng. 9 (1997) 508–511.
[21] T. Griffin, L. Libkin, Incremental maintenance of views with duplicates, in: M.J. Carey, D.A. Schneider (Eds.), Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data, ACM Press, 1995, pp. 328–339.
[22] D. Vista, Integration of incremental view maintenance into query optimizers, in: H. Schek, F. Saltor, I. Ramos, G. Alonso (Eds.), Proc. of the 6th International Conference on Extending Database Technology, EDBT 1998, in: LNCS, vol. 1377, Springer, 1998, pp. 374–388.
[23] D. Vista, Optimizing Incremental View Maintenance Expressions in Relational Databases, Ph.D. thesis, University of Toronto, Ont., Canada, 1997.
[24] J.-M. Nicolas, K. Yazdanian, An outline of BDGEN: a deductive DBMS, in: R.E.A. Mason (Ed.), Proc. of the IFIP 9th World Computer Congress, Elsevier Science, 1983, pp. 711–717.
[25] A. Gupta, I.S. Mumick, V.S. Subrahmanian, Maintaining views incrementally, in: P. Buneman, S. Jajodia (Eds.), Proc. of the ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 1993, ACM Press, 1993, pp. 157–166.
[26] A. Gupta, D. Katiyar, I.S. Mumick, Counting solutions to the view maintenance problem, in: K. Ramamohanarao, J. Harland, G. Dong (Eds.), Proc. of the Workshop on Deductive Databases, in conjunction with the Joint Int. Conf. and Symposium on Logic Programming, in: Technical Report, vol. CITRI/TR-92-6, Department of Computer Science, University of Melbourne, 1992, pp. 185–194.
[27] O. Wolfson, H.M. Dewan, S.J. Stolfo, Y. Yemini, Incremental evaluation of rules and its relationship to parallelism, in: J. Clifford, R. King (Eds.), Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 1991, ACM Press, 1991, pp. 78–87.
[28] M. Staudt, M. Jarke, Incremental maintenance of externally materialized views, in: T.M. Vijayaraman, A.P. Buchmann, C. Mohan, N.L. Sarda (Eds.), Proc. of the 22nd Int. Conf. on Very Large Data Bases, VLDB 1996, Morgan Kaufmann, 1996, pp. 75–86.

[29] J. Kotowski, F. Bry, S. Brodt, Reasoning as axioms change — incremental view maintenance reconsidered, in: S. Rudolph, C. Gutierrez (Eds.), Proc. of the 5th Int. Conf. on Web Reasoning and Rule Systems, RR 2011, in: LNCS, vol. 6902, Springer, 2011, pp. 139–154.
[30] J.J. Lu, G. Moerkotte, J. Schü, V.S. Subrahmanian, Efficient maintenance of materialized mediated views, in: M.J. Carey, D.A. Schneider (Eds.), Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data, ACM Press, 1995, pp. 340–351.
[31] K.R. Apt, J.-M. Pugin, Maintenance of stratified databases viewed as a belief revision system, in: M.Y. Vardi (Ed.), Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1987, ACM Press, 1987, pp. 136–145.
[32] D.F. Barbieri, D. Braga, S. Ceri, E.D. Valle, M. Grossniklaus, Incremental reasoning on streams and rich background knowledge, in: L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, T. Tudorache (Eds.), Proc. of the 7th Extended Semantic Web Conference, ESWC 2010, in: LNCS, vol. 6088, Springer, 2010, pp. 1–15.
[33] V. Küchenhoff, On the efficient computation of the difference between consecutive database states, in: C. Delobel, M. Kifer, Y. Masunaga (Eds.), Proc. of the 2nd Int. Conf. on Deductive and Object-Oriented Databases, DOOD 1991, in: LNCS, vol. 566, Springer, 1991, pp. 478–502.
[34] T. Urpí, A. Olivé, A method for change computation in deductive databases, in: L. Yuan (Ed.), Proc. of the 18th Int. Conf. on Very Large Data Bases, VLDB 1992, Morgan Kaufmann, 1992, pp. 225–237.
[35] J.V. Harrison, S.W. Dietrich, Maintenance of materialized views in a deductive database: an update propagation approach, in: K. Ramamohanarao, J. Harland, G. Dong (Eds.), Proc. of the Workshop on Deductive Databases, held at Joint Int. Conf. and Symposium on Logic Programming, in: Technical Report, vol. CITRI/TR-92-65, Department of Computer Science, University of Melbourne, 1992, pp. 56–65.
[36] G. Dong, R.W. Topor, Incremental evaluation of datalog queries, in: J. Biskup, R. Hull (Eds.), Proc. of the 4th Int. Conf. on Database Theory, ICDT 1992, in: LNCS, vol. 646, Springer, 1992, pp. 282–296.
[37] G. Dong, J. Su, R.W. Topor, Nonrecursive incremental evaluation of datalog queries, Ann. Math. Artif. Intell. 14 (1995) 187–223.
[38] G. Dong, J. Su, Incremental and decremental evaluation of transitive closure by first-order queries, Inf. Comput. 120 (1995) 101–106.
[39] G. Dong, C. Pang, Maintaining transitive closure in first order after node-set and edge-set deletions, Inf. Process. Lett. 62 (1997) 193–199.
[40] G. Dong, K. Ramamohanarao, Maintaining constrained transitive closure by conjunctive queries, in: F. Bry, R. Ramakrishnan, K. Ramamohanarao (Eds.), Proc. of the 5th Int. Conf. on Deductive and Object-Oriented Databases, DOOD, 1997, in: LNCS, vol. 1341, Springer, 1997, pp. 35–51.
[41] C. Pang, K. Ramamohanarao, G. Dong, Incremental $FO(+,<)$ maintenance of all-pairs shortest paths for undirected graphs after insertions and deletions, in: C. Beeri, P. Buneman (Eds.), Proc. of the 7th Int. Conf. on Database Theory, ICDT, 1999, in: LNCS, vol. 1540, Springer, 1999, pp. 365–382.
[42] G. Dong, J. Su, Increment boundedness and nonrecursive incremental evaluation of datalog queries, in: G. Gottlob, M.Y. Vardi (Eds.), Proc. of the 5th Int. Conf. on Database Theory, ICDT, 1995, in: LNCS, vol. 893, Springer, 1995, pp. 397–410.
[43] G. Dong, J. Su, Arity bounds in first-order incremental evaluation and definition of polynomial time database queries, J. Comput. Syst. Sci. 57 (1998) 289–308.
[44] G. Dong, J. Su, Deterministic FOIES are strictly weaker, Ann. Math. Artif. Intell. 19 (1997) 127–146.
[45] G. Dong, L. Libkin, L. Wong, On impossibility of decremental recomputation of recursive queries in relational calculus and SQL, in: P. Atzeni, V. Tannen (Eds.), Proc. of the 5th Int. Workshop on Database Programming Languages DBPL-5, in: Electronic Workshops in Computing, Springer, 1995, p. 7.
[46] L. Libkin, L. Wong, Incremental recomputation of recursive queries with nested sets and aggregate functions, in: S. Cluet, R. Hull (Eds.), Proc. of the 6th Int. Workshop on Database Programming Languages, DBPL-6, in: LNCS, vol. 1369, Springer, 1997, pp. 222–238.
[47] G. Dong, L. Libkin, J. Su, L. Wong, Maintaining transitive closure of graphs in SQL, Int. J. Inf. Technol. 5 (1999) 46–78.
[48] S. Patnaik, N. Immerman, Dyn-FO: a parallel, dynamic complexity class, J. Comput. Syst. Sci. 55 (1997) 199–209.
[49] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, T. Zeume, Reachability is in DynFO, in: M.M. Halldórsson, K. Iwama, N. Kobayashi, B. Speckmann (Eds.), Proc. of the 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, in: LNCS, vol. 9135, Springer, 2015, pp. 159–170.
[50] T. Zeume, T. Schwentick, Dynamic conjunctive queries, J. Comput. Syst. Sci. 88 (2017) 3–26.
[51] J.A. Blakeley, N. Coburn, P. Larson, Updating derived relations: detecting irrelevant and autonomously computable updates, ACM Trans. Database Syst. 14 (1989) 369–400.
[52] C. Elkan, Independence of logic database queries and updates, in: D.J. Rosenkrantz, Y. Sagiv (Eds.), Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1990, ACM Press, 1990, pp. 154–160.
[53] A.Y. Levy, Y. Sagiv, Queries independent of updates, in: R. Agrawal, S. Baker, D.A. Bell (Eds.), Proc. of the 19th Int. Conf. on Very Large Data Bases, VLDB 1993, Morgan Kaufmann, 1993, pp. 171–181.
[54] A. Gupta, H.V. Jagadish, I.S. Mumick, Data integration using self-maintainable views, in: P.M.G. Apers, M. Bouzeghoub, G. Gardarin (Eds.), Proc. of the 5th Int. Conf. on Extending Database Technology, EDBT, 1996, in: LNCS, vol. 1057, Springer, 1996, pp. 140–144.
[55] F.W. Tompa, J.A. Blakeley, Maintaining materialized views without accessing base data, Inf. Syst. 13 (1988) 393–406.
[56] A. Segev, J. Park, Updating distributed materialized views, IEEE Trans. Knowl. Data Eng. 1 (1989) 173–184.
[57] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom, View maintenance in a warehousing environment, in: M.J. Carey, D.A. Schneider (Eds.), Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 1995, ACM Press, 1995, pp. 316–327.
[58] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, H. Trickey, Algorithms for deferred view maintenance, in: H.V. Jagadish, I.S. Mumick (Eds.), Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 1996, ACM Press, 1996, pp. 469–480.
[59] T. Griffin, R. Hull, A framework for implementing hypothetical queries, in: J. Peckham (Ed.), Proc. of the 1997 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD 1997, ACM Press, 1997, pp. 231–242.
[60] B.G. Lindsay, L.M. Haas, C. Mohan, H. Pirahesh, P.F. Wilms, A snapshot differential refresh algorithm, in: C. Zaniolo (Ed.), Proc. of the 1986 ACM SIGMOD International Conference on Management of Data, SIGMOD 1986, ACM Press, 1986, pp. 53–60.
[61] B. Kähler, O. Risnes, Extending logging for database snapshot refresh, in: P.M. Stocker, W. Kent, P. Hammersley (Eds.), Proc. of 13th Int. Conf. on Very Large Data Bases, VLDB 1987, Morgan Kaufmann, 1987, pp. 389–398.
[62] J. de Kleer, An assumption-based TMS, Artif. Intell. 28 (1986) 127–162.
[63] J. Doyle, A truth maintenance system, Artif. Intell. 12 (1979) 231–272.
[64] C.L. Forgy, Rete: a fast algorithm for the many pattern/many object pattern match problem, Artif. Intell. 19 (1982) 17–37.
[65] G. Klyne, J.J. Carroll, B. McBride, RDF 1.1: concepts and abstract syntax, https://www.w3.org/TR/rdf11-concepts/, 2014.
[66] K.A. Ross, Modular stratification and magic sets for datalog programs with negation, J. ACM 41 (1994) 1216–1266.
[67] I.V. Ramakrishnan, R. Sekar, A. Voronkov, Term indexing, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, vol. II, Elsevier Science, 2001, pp. 1853–1964.
[68] A. Gaulton, L.J. Bellis, A.P. Bento, J. Chambers, M. Davies, A. Hersey, Y. Light, S. McGlinchey, D. Michalovich, B. Al-Lazikani, J.P. Overington, ChEMBL: a large-scale bioactivity database for drug discovery, Nucleic Acids Res. 40 (2012) 1100–1107.
[69] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, C. Bizer, DBpedia — a large-scale, multilingual knowledge base extracted from Wikipedia, Semant. Web 6 (2015) 167–195.
[70] A. Fabregat, K. Sidiropoulos, P. Garapati, M. Gillespie, K. Hausmann, R. Haw, B. Jassal, S. Jupe, F. Korninger, S.J. McKay, L. Matthews, B. May, M. Milacic, K. Rothfels, V. Shamovsky, M. Webber, J. Weiser, M. Williams, G. Wu, L. Stein, H. Hermjakob, P. D'Eustachio, The reactome pathway knowledgebase, Nucleic Acids Res. 44 (2016) 481–487.

[71] T.U. Consortium, UniProt: a hub for protein information, Nucleic Acids Res. 43 (2015) 204–212.
[72] L. Ma, Y. Yang, Z. Qiu, G.T. Xie, Y. Pan, S. Liu, Towards a complete OWL ontology benchmark, in: Y. Sure, J. Domingue (Eds.), Proc. of the 3rd European Semantic Web Conf., ESWC 2006, in: LNCS, vol. 4011, Springer, 2006, pp. 125–139.
[73] Y. Guo, Z. Pan, J. Heflin, LUBM: a benchmark for OWL knowledge base systems, J. Web Semant. 3 (2005) 158–182.
[74] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation and Applications, 2nd edition, Cambridge University Press, 2007.
[75] B.N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: combining logic programs with description logic, in: Proc. of the 12th Int. World Wide Web Conference, WWW 2003, ACM Press, 2003, pp. 48–57.
[76] M. Kaminski, Y. Nenov, B.C. Grau, Datalog rewritability of Disjunctive Datalog programs and non-Horn ontologies, Artif. Intell. 236 (2016) 90–118.
[77] U. Hustadt, B. Motik, U. Sattler, Reasoning in description logics by a reduction to disjunctive datalog, J. Autom. Reason. 39 (2007) 351–384.
[78] Y. Zhou, B. Cuenca Grau, I. Horrocks, Z. Wu, J. Banerjee, Making the most of your triple store: query answering in OWL 2 using an RL reasoner, in: D. Schwabe, V.A.F. Almeida, H. Glaser, R.A. Baeza-Yates, S.B. Moon (Eds.), Proc. of the 22nd Int. World Wide Web Conference, WWW 2013, ACM Press, 2013, pp. 1569–1580.
[79] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Combining rewriting and incremental materialisation maintenance for datalog programs with equality, in: Q. Yang, M. Wooldridge (Eds.), Proc. of the 24th Int. Joint Conf. on Artificial Intelligence, IJCAI 2015, AAAI Press, 2015, pp. 3127–3133.
[80] J.A. Rice, Mathematical Statistics and Data Analysis, 3rd edition, Brooks Cole, 2006.
[81] B. Motik, Y. Nenov, R. Piro, I. Horrocks, D. Olteanu, Parallel materialisation of datalog programs in centralised, main-memory RDF systems, in: C.E. Brodley, P. Stone (Eds.), Proc. of the 28th AAAI Conf. on Artificial Intelligence, AAAI 2014, AAAI Press, 2014, pp. 129–137.