Reasoning as Axioms Change
Incremental View Maintenance Reconsidered

Jakub Kotowski, François Bry, and Simon Brodt

Institute for Informatics, University of Munich
http://pms.ifi.lmu.de

Abstract. We present a novel incremental algorithm to compute changes to materialized views in logic databases like those used by rule-based reasoners. Such reasoners have to address the problem of changing axioms in the presence of materializations of derived atoms. Existing approaches have drawbacks: some require to generate and evaluate large transformed programs that are in Datalog$^\neg$ while the source program is in Datalog and significantly smaller; some recompute the whole extension of a predicate even if only a small part of this extension is affected by the change. The method presented in this article overcomes both drawbacks, arguably at an acceptable price: a slight adaptation of the semi-naïve forward chaining.

1 Introduction and Motivation

As the mostly read-only Web becomes a predominantly read/write social Semantic Web [4,3], information becomes more volatile and systems that can handle changes efficiently grow more important. Most Semantic Web applications make use of the Resource Description Framework (RDF) [21] and the Web Ontology Language (OWL) [37]. A large part of RDF semantics [19] and for example of the OWL 2 RL profile [27], an OWL 2 sublanguage, can be axiomatized using rules and implemented in logic databases [35,31]. The work presented in this paper is part of the development of a social semantic platform one application of which is a semantic wiki that employs RDF and OWL.[1] Wikis are typically affluent with user activity that creates and changes facts either directly or as a side-effect. In a semantic wiki equipped with reasoning about metadata, this trait poses high demands on the reasoner especially in terms of response times as users expect a nimble user interface with up-to-date information. Hence the need for materialization of views and their maintenance. Materialization of views, i.e. storing atoms derived from rules and base facts, is a technique often used in databases for improving the speed of query evaluation and it has been argued feasible for Semantic Web data management systems [7,6] too. Efficient incremental maintenance of materialized views is highly desirable as rules and base facts change. This is true also for the Semantic Web as semantic wikis can be

[1] This social semantic platform[29] is the outcome of the project KiWi http://kiwi-project.eu/

seen as a testbed for the Semantic Web or "Semantic Web in the small" [30]. The problem of incremental view maintenance has been studied in deductive databases and is related to a wider range of fields, see related work in Section 3. Let us now define the problem more formally.

2 Incremental View Maintenance

Let P be a definite range restricted logic program and $D \subseteq P$ a subset of P. The *view maintenance problem* is the problem of computing $T^{\omega}_{P \setminus D}$ given the fixpoint T^{ω}_{P}, where T_P is the immediate consequence operator.

The view maintenance problem has a trivial solution which is computing the fixpoint $T^{\omega}_{P \setminus D}$ directly from $P \setminus D$, disregarding T^{ω}_{P}. This solution is obviously inefficient because it necessarily repeats much of the computation used to generate T^{ω}_{P}. The view maintenance problem can also be seen as a version of the *frame problem*[20] as noted in [28] and [9]. This paper studies how to solve the view maintenance problem *incrementally* by leveraging information available from from the fixpoint computed before axioms (that is, base facts or rules) change. Here, this modified task is called the *incremental view maintenance problem*.

To solve the incremental view maintenance problem, it is necessary to decide amongst other for each atom $g \in T^{\omega}_{P}$ whether $g \in T^{\omega}_{P \setminus D}$. This amounts to deciding whether g can be derived from $P \setminus D$. Any algorithm that decides this question has to demonstrate that there is a derivation of g with respect to $P \setminus D$. In other words, the incremental maintenance problem can be solved for example by keeping track of derivations of each atom. This focus of this paper is on solving the view maintenance problem without keeping information other than the view materialization T^{ω}_{P}.

It is important to clearly distinguish the incremental view maintenance problem from the view update problem [15]. The view update problem is to change a program so that a given formula cannot (resp. can) be derived from it.

3 Related Work

The view maintenance problem is essentially a problem of changing knowledge and especially a problem of updating derived facts or beliefs upon changes in base facts or assumptions. As such, the problem is related to a wealth of literature ranging from epistemology, to logic, to databases. One of the overarching concepts is defeasible reasoning [22] which includes two subfields related to this work: belief revision (an epistemological approach) and reason maintenance (a logical approach). The problem, as described here, was studied mainly in the deductive database community and recently also in the Semantic Web community.

Belief revision is developed as a formal theory by Alchourrón, Gärdenfors and Makinson [1,2], often it is called the *AGM theory*. It views all formulas as equally important in general and it aims at revising a theory so that only a minimal change occurs in its deductive closure. In particular, it makes no distinction

between base and derived facts and thus it is more relevant for example to beliefs in multi-agent systems than to views in databases. Indeed, databases typically manage large sets of base facts and only a few views and the distinction between base facts and derived facts is an important one.

Reason maintenance [13] (originally truth maintenance [12]) refers to a variety of knowledge base update techniques originally developed for use in problem solvers which share a common conceptual design – they distinguish between an inference engine and a separate reason maintenance system which communicate via an interface [26] with each other and, in addition to derived atoms, they keep a record of derivations in form of a "data dependency network". In contrast, the method developed in this paper assumes that only the atoms are stored. One of the reason maintenance approaches has been specialized to RDF(S) reasoning and implemented by Broekstra et al. [7] in the area of semantic web as part of the Sesame triple store [5].

Belief revision and reason maintenance are closely related and are compared in the literature for example by Jon Doyle [14], the founder of reason maintenance, and by Alvaro Val [34].

The incremental view maintenance problem has been studied in the field of *deductive databases* on and off since around 1980, see for example a survey article [16] by Gupta and Mumick, authors of the probably most popular DRed algorithm [17]. Most of the original algorithms do not directly handle rule changes but they can be extended to do so. The method described in this paper automatically handles rule changes too.

The most prominent incremental view maintenance algorithms are the DRed (derive and rederive) algorithm [17] and the PF (propagate filter) algorithm [18]. Both work on the same principle of deriving an overestimation of deleted atoms and then finding alternative derivations for them. The DRed algorithm first derives the whole overestimation and only then finds alternative derivations. The PF algorithm, which was originally developed in the context of top-down memoing [10], finds alternative derivations (filters) as soon as an atom to possibly be deleted is derived (propagated). Both algorithms perform the two steps by evaluating a *transformed* version of the original rules (resulting in a bigger program). In contrast, the method presented in this paper uses the original unchanged program.

Staudt and Jarke developed in [32] a purely declarative version of DRed. Their algorithm, however, transforms the original program into even more rules than DRed itself. Also, the transformed program includes negation even if the original program does not.

Recently, Volz, Staab, and Motik extended [36] Staudt and Jarke's version of DRed to handle rule changes and applied the resulting method to reasoning on the *semantic web*. For example, the Volz, Staab, and Motik version of DRed transforms 12 RDF semantics Datalog rules into a maintenance program of 60 rules [36]. Their method leads to complete recomputation for any change in base triples in the case of a single (ternary) predicate axiomatization of RDF(S) which is a significant disadvantage especially in the area of semantic web where this

kind of axiomatization is very common. In comparison, the algorithm presented in this paper makes do with the original 12 rules, no modification is necessary to handle general rule updates, and always only the relevant part of a predicate's extension is recomputed, including the case of a single predicate axiomatization of RDF(S).

4 Preliminaries

Throughout this paper a Datalog [33] rule language is assumed, rules are assumed to be range restricted, and the usual notation, e.g. $h \leftarrow b_1, b_2, \ldots, b_n$, is used (we use a dot "." as the rule separator where necessary). That is, it is assumed that the logical language has no function symbols other than constants and it is assumed that it includes a finite number of constants but at least one constant and two formulas: $\top$ and $\bot$ which respectively evaluate to true and false in all interpretations, and all rules are range restricted, i.e. a variable occurring in the rule head occurs in the rule body too. Rules with $\top$ as the body, e.g. $h \leftarrow \top$, are called base facts. A base atom is the head of a base fact. A program is a finite set of rules. In addition, the usual definition of the immediate consequence operator T_P and of its ordinal powers is assumed. See for example [8] for details. Let $r = h \leftarrow b_1, \ldots, b_n$ be a rule. Then $\text{head}(r) = h$, $\text{body}(r) = \{b_1, \ldots, b_n\}$, and $\text{atoms}(r) = \text{body}(r) \cup \{\text{head}(r)\}$. $\mathbb{N}$ denotes the set of natural numbers including zero, $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$.

Let a and b be two ground atoms, e.g. $a, b \in HB$, where HB is the Herbrand base. We say that b directly depends on a in T_P^ω (or with respect to P) if there is a a rule instance r of a rule in P such that $\text{atoms}(r) \subseteq T_P^\omega$, $a \in \text{body}(r)$, and $a = \text{head}(r)$. We define the (indirectly) depends relation as the transitive closure of the directly depends relation. We say that an atom a directly depends on a rule r in T_P^ω (or with respect to P) if there is a ground rule instance s of r such that $\text{atoms}(s) \subseteq T_P^\omega$ and $\text{head}(s) = a$. We say that an atom a depends on a rule r in T_P^ω (or with respect to P) if it directly depends on r in T_P^ω or if it depends on an atom $b \in T_P^\omega$ that directly depends on r in T_P^ω. Note that if P is a *recursive* Datalog program then an atom may depend on itself in T_P^ω.

Lemma 1. *Let P be a definite range restricted Datalog program and $a \in HB$ a ground atom. Then $a \in T_P^\omega$ iff a depends on a rule in P.*

Proof. By induction on the power of T_P, resp. the length of the sequence of rules documenting the "depends on" relation.

Algorithm 1.1: Classical semi-naive forward chaining

```
Name
  semi-naive-forward-chaining(P)
Input
  P – a definite range restricted program
```

```
Output
  T_P^ω — the least fixpoint of T_P
Variables
  Δ — a set of atoms
Initialization
  F := ∅; Δ := T_P(∅)
begin
  while Δ ≠ ∅
  begin
    F := F ∪ Δ
    Δ := T_P(F, Δ) \ F
  end
  return F
end
```

Note that in Algorithm 1.1, "T_P" denotes both the immediate consequence operator and the corresponding semi-naive mapping $T_P(F, \Delta) : \mathcal{P}(HB) \times \mathcal{P}(HB) \rightarrow \mathcal{P}(HB)$.

Let us first examine the fixpoint of the T_P operator with respect to program updates.

5 An Analysis of the Fixpoint

Three sets of atoms can be distinguished in a fixpoint T_P^ω with respect to a set of rules $D \subset P$ to remove: the set of atoms that do not lose any derivation after D is removed, the set of atoms that lose a derivation after D is removed, and the set of atoms that lose a derivation after D is removed but are still derivable from P. The following introduces a notation for these three sets which is used throughout the rest of the paper.

Notation 1 *Let P be a definite range restricted Datalog program and $D \subseteq P$ a set of rules to remove. Then*

- $U = \{a \in T_P^\omega \mid (\exists d \in D)$ *a depends on d with respect to* $P\}$,
- $K = T_P^\omega \setminus U$
- $O = U \cap T_{P \setminus D}^\omega$

U is the set of [u]nsure atoms, i.e. the set of atoms lose at least one derivation as a result of removing D. K is the set of atoms to [k]eep, i.e. the set of atoms that do not lose any derivation after the removal of D. O is the set of [o]therwise "supported" atoms, i.e. the set of atoms from U that have derivations which do not include any rule from D.

T_P^ω will also be referred to as the old fixpoint *and $T_{P \setminus D}^\omega$ as the* new fixpoint.

The following lemma and Figure 1 illustrate relationships between U, K, and Os.

Lemma 2. *The following holds:*

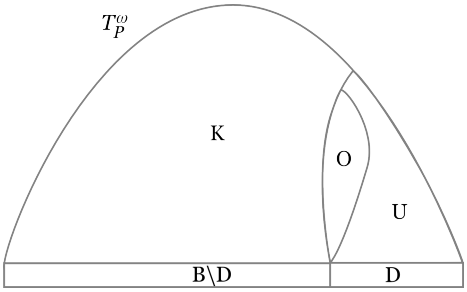

Fig. 1: Illustration of how K, U, and O relate in the case that D is a set of base facts. B is the set of all base facts in P. With removal of arbitrary rules, there may be more "U" sets each having its own "O" subset, see Figure 2.

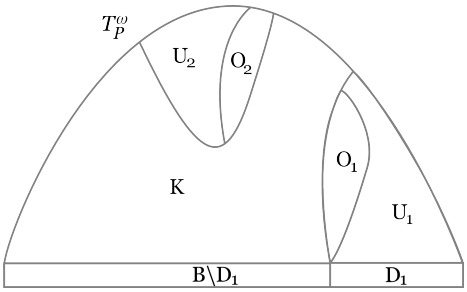

Fig. 2: Illustration of how K, U, and O relate in the general case. B is the set of all base facts in P. D_1 is a set of base facts to remove, D_2 is a set of rules to remove, $D = D_1 \cup D_2$. U_1 is the set of atoms that depend on a fact from D_1. U_2 is the set of atoms that depend on a rule from D_2. $U = U_1 \cup U_2$.

1. $U \cup K = T_P^\omega$,
2. $U \cap K = \emptyset$,
3. $O \cap K = \emptyset$.

Proof. Follows immediately from definition.

Lemma 3. *The following holds:*

1. $K \subseteq T_{P\setminus D}^\omega$,
2. $T_{P\setminus D}^\omega = K \cup O$,
3. $T_{P\setminus D}(K) \subseteq T_{P\setminus D}^\omega$.

Proof. Point 1: Let $a \in K = T_P^\omega \setminus U$. Assume by contradiction that $a \notin T_{P\setminus D}^\omega$ then a does not depend on a rule in $P \setminus D$ (Lemma 1) which is in contradiction with $a \in T_P^\omega$ because $(P \setminus D) \subseteq P$ and Lemma 1.

Point 2: $K \cup O = (T_P^\omega \setminus U) \cup (U \cap T_{P\setminus D}^\omega) \supseteq T_{P\setminus D}^\omega$, from Notation 1 and because $T_{P\setminus D}^\omega \subseteq T_P^\omega$ for definite programs. $K \cup O \subseteq T_{P\setminus D}^\omega$ by Point 1 and because $O \subseteq T_{P\setminus D}^\omega$ by definition.

Point 3 follows from Point 1 and the fact that $T^{\omega}_{P\setminus D}$ is the least fixpoint of $T_{P\setminus D}$.

Lemma 3 shows a way to compute the new fixpoint by determining the sets K and O. The core of the problem lies in determining the set O, i.e. those atoms that lose a derivation after removing D but are derivable with respect to $P \setminus D$ nevertheless. The following two propositions show that the new fixpoint can be computed by forward chaining on the set K.

Proposition 1. $T^{\omega}_{P\setminus D} = T^{\omega}_{P\setminus D}(K)$.

Proof. $T_{P\setminus D}(\emptyset) \subseteq T_{P\setminus D}(K)$ by monotonicity of T_P. Therefore also $T^{n}_{P\setminus D} \subseteq T^{n}_{P\setminus D}(K)$, for all $n \in \mathbb{N}_1$, by monotonicity of T_P.

$\subseteq$: Let $a \in T^{\omega}_{P\setminus D} = \bigcup\{T^{\beta}_{P\setminus D} \mid \beta < \omega\}$. Then there is a β such that $a \in T^{\beta}_{P\setminus D} \subseteq T^{\beta}_{P\setminus D}(K) \in \{T^{\beta}_{P\setminus D}(K) \mid \beta < \omega\}$. Therefore $a \in T^{\omega}_{P\setminus D}(K)$. In summary $T^{\omega}_{P\setminus D} \subseteq T^{\omega}_{P\setminus D}(K)$.

$\supseteq$: Let $a \in T^{\omega}_{P\setminus D}(K)$. There is an α such that $a \in T^{\alpha}_{P\setminus D}(K)$. By Lemma 3, it holds that $K \subseteq T^{\omega}_{P\setminus D}$. Therefore there is a β such that $K \subseteq T^{\beta}_{P\setminus D}$. Thus, $T^{\alpha}_{P\setminus D}(K) \subseteq T^{\beta+\alpha}_{P\setminus D}$ by monotonicity of T_P. Together, $a \in T^{\beta+\alpha}_{P\setminus D} \subseteq T^{\omega}_{P\setminus D}$. And in summary, $T^{\omega}_{P\setminus D}(K) \subseteq T^{\omega}_{P\setminus D}$.

An immediate corollary of Proposition 1 is the following proposition which states that all atoms in O are eventually derived by forward chaining on K.

Proposition 2. *If $a \in O$ then there is an $n \in \mathbb{N}_1$ such that $a \in T^{n}_{P\setminus D}(K)$.*

Proof. Let $a \in O$. $O = U \cap T^{\omega}_{P\setminus D}$, therefore $a \in T^{\omega}_{P\setminus D}(K)$ by Proposition 1. P (and therefore also $P \setminus D$) is a definite range restricted Datalog program. Therefore the fixpoint is reached in a finite number of steps and therefore there is an $n \in \mathbb{N}_1$ such that $a \in T^{n}_{P\setminus D}(K)$.

Let us now show how the set U can easily be determined by a slightly modified semi-naive forward chaining algorithm.

6 Overestimation of Deletions

The set U of atoms possibly affected by a removal can be computed by a simple modification of semi-naive forward chaining. In fact, only a different initialization is necessary. The main advantage over existing approaches is that the computation fully exploits the already computed fixpoint as it is shown later.

Algorithm 1.2: Overestimation of atoms to delete.

```
Name
  dependent-atoms(P, T^ω_P, D)
```

```
Input
  P − a definite range restricted program
  D − a set of rules, D ⊆ P
Output
  F − a set of atoms from T_P^ω that depend on a rule from D in
      T_P^ω, i.e. the set U of Notation 1
Variables
  Δ − a set of atoms
Initialization
  F := ∅; Δ := T_D(T_P^ω)
begin
  while Δ ≠ ∅
  begin
    F := F ∪ Δ
    Δ := T_P(T_P^ω, Δ) \ F
  end
  return F
end
```

One difference in the initialization in comparison to classical semi-naive forward chaining is that Δ is initialized with immediate consequences of the rules D to be removed with respect to the old fixpoint (notice the D in $T_D(T_P^\omega)$). The second difference is that the semi-naive T_P mapping is applied to the fixpoint instead of the set F. Using the fixpoint as the input to the mapping means that the computation is faster than the original forward chaining, see the following example, while the Δ ensures that any derived atom depends on an atom from the set of atoms with which Δ was initialized and thus on a rule from D.

Example 1. Let $P = \{a \leftarrow \top.\quad b \leftarrow a.\quad c \leftarrow a, b\}$. The atom c is derived only in the second iteration of classical (semi-naive) forward chaining from scratch while it is derived already in the first iteration of Algorithm 1.2 with the input T_P^ω, $D = \{a \leftarrow \top\}$. Note that the difference increases with the length of the minimal derivation of an atom.

Lemma 4. *Algorithm 1.2 is correct in the sense that it computes exactly the atoms from T_P^ω that depend on a rule from D in T_P^ω.*

Proof. Let i be the iteration number and let F_i and Δ_i be the F and Δ computed in the i-th iteration (i.e. after i executions of the while body). All uses of the "depends" relationship are with respect to P in this proof.

We will show by induction on i that $a \in F_i$ iff $a \in T_P^\omega$ depends on a rule from D and there is a sequence of at most $\leq i$ rules as evidence.

$i = 0$: Trivial because $F_0 = \emptyset$.

$i = 1$: $F_1 = F_0 \cup \Delta_0 = \emptyset \cup T_D(T_P^\omega)$. That is if $a \in F_1$ then $a \in T_D(T_P^\omega)$ and there is an instance of a rule in D the head of which is a and thus a depends on it. Conversely, if $a \in T_P^\omega$ depends on a rule from D and it is evidenced by a single rule then the rule must be an instance of a rule in D, hence $a \in T_D(T_P^\omega) = F_1$.

$i \rightsquigarrow i + 1$.

$\Rightarrow$: Let $a \in F_{i+1}$. $F_{i+1} = F_i \cup \Delta_i$. If $a \in F_i$ then the statement holds by induction hypothesis. If $a \in \Delta_i$ then $a \in T_P(T_P^\omega, \Delta_{i-1}) \setminus F_{i-1}$. Therefore there is an instance r of a rule in P such that $\text{head}(r) = a$ and there is an atom $b \in \text{body}(r)$ that is also in $\Delta_{i-1} \subseteq F_i$ and thus, by induction hypothesis, b depends on a rule from D and there is a sequence of at most i rules that shows it. Adding r to the end of the sequence creates one of length at most $i+1$ that shows that a depends on a rule from D.

$\Leftarrow$: Let $a \in T_P^\omega$ such that it depends on a rule from D and there is a sequence of rule instances $r_1, \ldots, r_i, r_{i+1}$ that shows it. Then by induction hypothesis $\text{head}(r_i) \in F_i$. Therefore there is a $j < i$ such that $\text{head}(r_i) \in \Delta_j$ (because $F_i = \bigcup_{k=0}^{i} \Delta_k$). Therefore $a = \text{head}(r_{i+1}) \in T_P(T_P^\omega, \Delta_j)$, and thus either $a \in \Delta_{j+1}$ or $a \in F_j$, in either case $a \in F_{j+1} \subseteq F_{i+1}$.

The algorithm terminates because the Herbrand base is finite in the Datalog case, the set of ground atoms F increases in each step, and each atom is added to F at most once.

7 The Incremental View Maintenance Method

A incremental view maintenance algorithm to compute the new fixpoint based on the above observations can be summarized as follows:

1. Determine U (and therefore K).
2. Compute $T_{P \setminus D}^\omega(K)$

See Algorithm 1.3 for a complete specification.

Algorithm 1.3: Incremental view maintenance

```
Name
  FP-update(P, T_P^ω, D)
Input
  P − a range restricted definite Datalog program
  D − a subset of P, the set of rules to remove
  T_P^ω − the least fixpoint of T_P
Output
  T_{P\D}^ω − the new least fixpoint
Variables
  U, K, Δ, F − sets of atoms
Initialization
  U := dependent-atoms(P, T_P^ω, D)
  K := T_P^ω \ U
  Δ := T_{P\D}(K) \ K
  F := K
begin
  while Δ ≠ ∅
  begin
    F := F ∪ Δ
```

```
20      Δ := T_{P\D}(F, Δ) \ F
    end
    return F
23  end
```

Proposition 3. *Algorithm 1.3 terminates and computes* $T^{\omega}_{P\setminus D}$.

Proof. The algorithm correctly computes the set K, by definition of K and because the dependent atoms algorithm is correct (Lemma 4).

The algorithm computes $T^{i}_{P\setminus D}(K)$ where i is the $i-th$ iteration of the while loop in the algorithm. The algorithm computes $T^{\omega}_{P}(K)$ by a similar argument as for the correctness of classical semi-naive forward chaining. The Herbrand base is finite for a Datalog language with a finite number of constants and the set of ground atoms F increases in each iteration and each atom is added at most once. Hence the algorithm terminates.

The rest follows immediately from Proposition 1.

Let Δ_i be the Δ in the i-th iteration of the algorithm. Notice that $(\bigcup_{i=0}^{n} \Delta_i) = O$, where n is the number of the iteration in which fixpoint is reached.

Algorithm 1.3 uses the original unmodified program to compute the new fixpoint from the old one. It amounts to two consecutive semi-naive forward chainings each time with a different initialization.

Notice that the first Δ is computed naively in the initialization, the rest of the computation continues semi-naively. The naive step can be avoided if additional information such as the number of rule instances per atom is kept. This is however out of the scope of this paper as the formal treatment requires multiset semantics and distinguishing rule instances that are evidence of a proof from those that are not, i.e. the concept of well-foundedness from the field of reason maintenance [12,13,24].

Also note that this strategy of computing the new fixpoint relies on the heuristic that the set U is small in comparison with the set K. This obviously does not have to be always the case; a trivial example is when $D = P$.

7.1 Stratifiable normal programs

Algorithm 1.3 can be extended to stratifiable[8] normal programs. The analysis from Section 5 does not hold for normal Datalog programs; for example point 2 of Lemma 3 depends on monotonicity of T_P with respect to P: $T_{P'}(F) \subseteq T_P(F)$ for $P' \subseteq P$ which does not hold for normal programs. However, it is easy to see that the method can still be applied to stratifiable normal Datalog programs stratum by stratum.

7.2 Rule updates

It is worth stressing that Algorithm 1.3 is applicable to both base fact *and* general rule updates. In fact, if a rule, which is not a base fact, is to be removed

then all atoms that directly depend on it are determined in a single step of the dependent−atoms algorithm. Indeed, this is the main purpose of computing a view and keeping it materialized.

Atoms that indirectly depend on a rule to be removed can then be computed by the usual semi-naive forward chaining algorithm. This situation is depicted in Figure 2 by the sets with index 2 (D_2 is not depicted there; it consists of rules the rule instances of which "are on the border of U_2 and K").

8 Complexity

It is well-known [25] that T_P^ω can be computed in $O(n^k)$ time for a range restricted Datalog program P, where n is the number of terms (i.e. constants because all atoms in B are ground) in B and k is the maximum over all rules in P of the number of variables in that rule. To compute T_P^ω in the given time, one can generate all instances of all rules in P using the constants in B and then apply the well-known algorithm for the deductive closure of a set of ground Horn clauses [11], which runs in linear time (note that the satisfiability problem is NP-complete for the general class of *all* propositions). The worst case time complexity of Algorithm 1.2 can thus be estimated also as $O(n^k)$ because, in the worst case, computing the set of dependent atoms is equivalent to computing them by semi-naive forward chaining from scratch (see the next section for details). The worst-case time complexity of Algorithm 1.3 is the same as the worst-case time complexity of running semi-naive forward chaining twice from scratch, i.e. also $O(n^k)$.

While the worst-case time complexity of Algorithm 1.3 is likely the same as that of DRed and Staudt and Jarke's declarative DRed (and thus of the Volz, Staab, and Motik algorithm too), the maintenance program of declarative DRed creates longer rules and thus the k for a problem instance and declarative DRed is bigger than the k for the same problem instance and Algorithm 1.3. Declarative DRed also creates a maintenance program that is a Datalog program with (stratified) negation even for a Datalog program P without negation and thus it always requires stratum by stratum evaluation. Declarative DRed algorithms require evaluation of a maintenance program that is substantially bigger than the original program and for this reason too it can be reasonably expected that Algorithm 1.3 will be faster than these algorithms in practice as it requires evaluation of merely the original program.

Let us now compare compare Algorithm 1.3 with the existing methods in more detail in the next section.

9 An Example Evaluation

In the Encyclopedia of database systems [23], there is a comparison of the DRed and the PF algorithm [10] on a classic reachability example. Let us reproduce the example here and let us evaluate Algorithm 1.3 on it as well.

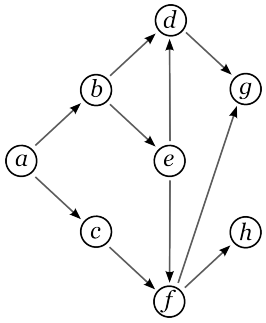

Fig. 3: A reproduction of a diagram found in [10]. A graph depicting the (extensional) edge predicate. There is a directed edge from x to y iff edge(x, y) holds. The edge (e, f) is to be removed.

Example 2. Let P be a datalog program that consists of the following two rules and of a set of base facts represented as a graph in Figure 3.

r_1: reach$(S, D) \leftarrow$ edge(S, D)

r_2: reach$(S, D) \leftarrow$ reach(S, I), edge(I, D)

Note that S can stand for "source", D for "destination", and I for "intermediate."

Algorithm 1.3 first determines the set of possible deletions U by calling Algorithm 1.2 and then it performs semi-naive forward-chaining on the set K (the complement of U in the fixpoint) and thus computes, using the new updated program, the new fixpoint. Figure 4 shows the old fixpoint of program P with the sets U, K, and O marked. The number of a row corresponds to the forward chaining iteration in which the atoms in the row are first derived. Atoms are ordered alphabetically in each row.

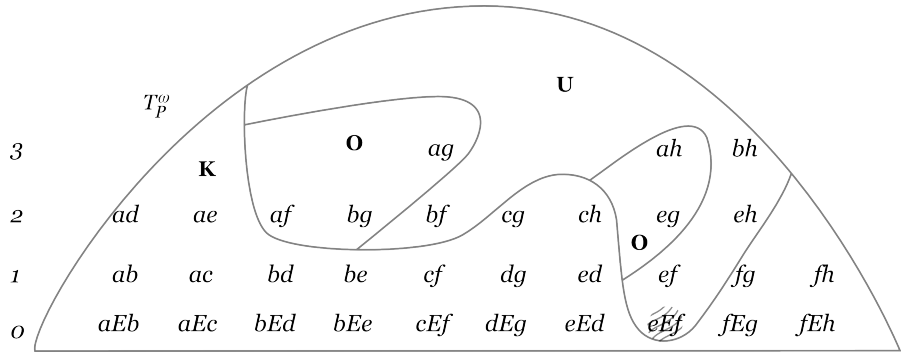

Fig. 4: A diagram showing the relationship between sets U, K, and O as computed by Algorithm 1.3 for Example 2. xy stands for reach(x, y), xEy stands for edge(x, y). Atom eEf (i.e. edge(e, f)) is being removed.

Table 1 shows a run of Algorithm 1.3 on the example, see also Figure 4.

Table 1: A run of Algorithm 1.3 for Example 2. xy stands for reach(x, y), xEy stands for edge(x, y). Atom eEf (i.e. edge(e, f)) is being removed.

Step 1	Overestimation (Algorithm 1.2)	
	F	Δ
Initialization	$\emptyset$	eEf
	eEf	ef
	eEf ef	bf af eg eh
	eEf ef bf af eg eh	bg bh ag ah
	eEf ef bf af eg eh bg bh ag ah	$\emptyset$
Step 2	The rest of Algorithm 1.3	
Initialization	$U = \{eEf\ ef\ bf\ af\ eg\ eh\ bg\ bh\ ag\ ah\}$ $\quad K = T_P^\omega \setminus U$ $\Delta = \{af\ bg\ eg\ ag\}$ $\quad F = K$	
	$F = K \cup \{af\ bg\ eg\ ag\}$	$\Delta = \{ah\}$
	$F = K \cup \{af\ bg\ eg\ ag\ ah\}$	$\Delta = \emptyset$

Algorithm 1.3 directly computes the new fixpoint. It needs three rule evaluations to compute the overestimation. In comparison, DRed needs four rule evaluations to compute the same result. As noted earlier, the difference becomes more significant with increasing lengths of derivations of atoms in the fixpoint. A similar comparison holds for the PF algorithm too because the overestimation phase of the PF algorithm tends to be less efficient than the one of the DRed algorithm (eager filtration may lead to redundant work).

The set of tuples deleted from the reach relation is of course the same for all three algorithms: $U \setminus (K \cup \{af\ bg\ eg\ ag\ ah\}) = \{ef\ eh\ bf\ bh\}$.

Algorithm 1.3 fully takes advantage of the already computed fixpoint when computing the set U of possible deletions (in contrast to both DRed and PF). The overestimation algorithm dependent−atoms performs the best when all derived atoms directly depend on a base atom. Then the overestimation is computed in a single iteration. The worst case occurs when for each rule instance r it holds that body$(r) \subseteq T_P^i$ and body$(r) \cap T_P^{i-1} = \emptyset$, i.e. for each rule instance all its body atoms are derived in the same forward chaining iteration when computing from scratch. In such a case, the dependent−atoms algorithm runs only as fast as normal forward chaining from scratch.

The decisive difference is that Algorithm 1.3 works with the original program P while DRed and PF use a transformed program P'. It is almost inevitable that P'-steps simulate P-steps and thus repeat work that was already done when computing the fixpoint for P.

10 Conclusion

We have presented a novel method for incremental maintenance of materialized recursive Datalog programs. Our method handles changes in both facts and rules, works by evaluating the original and the target programs while fully using the already computed fixpoint in the overestimation phase, and recomputes only the affected part of a predicate's extension. The method is applicable to important Datalog-fragments of Semantic Web languages. The method cannot be directly used in existing deductive databases due to the need for a modification of the classical semi-naive forward chaining. The modification is however only small and thus implementation is likely to be simple. While aggregation is not handled by our current method, we are confident that the extension is possible.

Acknowledgements.

We would like to thank Norbert Eisinger for valuable insights and discussions about the presented ideas. The research leading to these results is part of the project "KiWi - Knowledge in a Wiki" and has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 211932.

References

1. Alchourron, C.E., Gardenfors, P., Makinson, D.: On the logic of theory change: Contraction functions and their associated revision functions. Theoria 48 (1982)
2. Alchourron, C.E., Gardenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. J. Symbolic Logic (1985)
3. Berners-lee, T., Hollenbach, J., Lu, K., Presbrey, J., Schraefel, M.: Tabulator redux: Browsing and writing linked data. In: Proc. WWW2008 Workshop on Linked Data on the Web. vol. 369 (2008)
4. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American 284(5), 28–37 (May 2001)
5. Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. LNCS 2342 (2002)
6. Broekstra, J.: Storage, Querying and Inferencing for Semantic Web Languages. Ph.D. thesis, Vrije Universiteit (2005)
7. Broekstra, J., Kampman, A.: Inferencing and truth maintenance in RDF schema – exploring a naive practical approach. Workshop on Practical and Scalable Semantic Systems (PSSS) (2003)
8. Bry, F., Linse, B., Furche, T., Ley, C., Eiter, T., Eisinger, N., Gottlob, G., Pichler, R., Wei, F.: Foundations of rule-based query answering. Springer LNCS 4636 Reasoning Web, Third International Summer School 2007 (2007)
9. De Kleer, J.: Choices without backtracking. In: Proceedings of AAAI-84 (1984)
10. Dietrich, S.W.: Maintenance of Recursive Views. In: Encyclopedia of Database Systems, pp. 1674–1679. Springer Verlag (2009)
11. Dowling, W., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. The Journal of Logic Programming 1(3) (1984)

12. Doyle, J.: Truth maintenance systems for problem solving. Tech. Rep. AI-TR-419, Dep. of Electrical Engineering and Computer Science of MIT (1978)
13. Doyle, J.: The ins and outs of reason maintenance. Proc. IJCAI'83 (1983)
14. Doyle, J.: Reason maintenance and belief revision – Foundations vs. Coherence theories. Cambridge University Press (1992)
15. Guessoum, A., Lloyd, J.: Updating knowledge bases. New Generation Computing 8(1), 71–89 (June 1990)
16. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. Data Engineering Bulletin 18(2), 3–18 (1995)
17. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. SIGMOD Rec. 22, 157–166 (June 1993)
18. Harrison, J.V., Dietrich, S.W.: Maintenance of materialized views in a deductive database: An update propagation approach. In: Workshop on Deductive Databases, JICSLP. pp. 56–65 (1992)
19. Hayes, P.: RDF semantics. Tech. rep., W3C (2004)
20. Hayes, P.J.: The frame problem and related problems in artificial intelligence. Tech. rep., Stanford, CA, USA (1971)
21. Klyne, G., Carroll, J.J.: Resource description framework (RDF): Concepts and abstract syntax. Tech. rep., W3C (2004)
22. Koons, R.: Defeasible reasoning. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy (Spring 2005)
23. Liu, L., Özsu, M.T. (eds.): Encyclopedia of Database Systems. Springer (2009)
24. Martins, J.P., Shapiro, S.C.: A model for belief revision. Artificial Intelligence 35 (1988)
25. McAllester, D.: On the complexity analysis of static analyses. J. ACM 49(4), 512–537 (2002)
26. McAllester, D.A.: Truth maintenance. AAAI90 (1990)
27. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 web ontology language – profiles. Tech. rep., W3C (2009)
28. Nebel, B.: Reasoning and revision in hybrid representation systems. Springer-Verlag New York, Inc., New York, NY, USA (1990)
29. Schaffert, S., Eder, J., Grünwald, S., Kurz, T., Radulescu, M., Sint, R., Stroka, S.: KiWi–a platform for semantic social software. In: Proceedings of the 4th Workshop on Semantic Wikis, ESWC (2009)
30. Schaffert, S., Bry, F., Baumeister, J., Kiesel, M.: Semantic wikis. IEEE Software 25 (2008)
31. Sintek, M., Decker, S.: Triple – a query, inference, and transformation language for the semantic web. In: Horrocks, I., Hendler, J. (eds.) The Semantic Web – ISWC 2002, LNCS, vol. 2342, pp. 364–378. Springer (2002)
32. Staudt, M., Jarke, M.: Incremental Maintenance of Externally Materialized Views. In: Proc. 22th Int. Conf. VLDB. San Francisco, CA, USA (1996)
33. Ullman, J.D.: Principles of database and knowledge-base systems. Computer Science Press (1989)
34. Val, A.D.: On the relation between the coherence and foundations theories of belief revision. In: Proc. 12th Nat. Conf. on AI. AAAI (1994)
35. Volz, R.: Web Ontology Reasoning in Logic Databases. Ph.D. thesis, Universitaet Fridericiana zu Karlsruhe (TH) (2004)
36. Volz, R., Staab, S., Motik, B.: Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. In: Journal on Data Semantics II, LNCS, vol. 3360. Springer, Berlin, Heidelberg (2005)
37. W3C: OWL 2 web ontology language. Tech. rep., W3C (2009)