A Potpourri of Reason Maintenance Methods
Incremental View Maintenance Reconsidered

Jakub Kotowski, François Bry, and Norbert Eisinger

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

Abstract. We present novel methods 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 methods presented in this article overcome these drawbacks and derive additional information useful also for explanation, at the price of an adaptation of the semi-naïve forward chaining.

1 Introduction and Motivation

The mostly read-only Web has become a predominantly read/write social Semantic Web – information is more volatile and systems that can handle changes efficiently grow important. Semantic Web applications make use of the Resource Description Framework (RDF) and the Web Ontology Language (OWL) a large part of which can be axiomatized using Datalog rules. Social web applications (e.g. wikis) are teeming with user activity that changes facts which in turn poses high demands on a reasoner. 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 database technique for improving the speed of query evaluation and it has been argued [6] feasible on the Semantic Web too. Efficient incremental maintenance of materialized views is highly desirable as rules and base facts change. Let us define the problem formally.

Let P be a definite range restricted logic program and $D \subseteq P$ a subset of P. The *reason 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 with respect to P. The reason maintenance problem has a trivial inefficient solution which is computing the fixpoint $T^\omega_{P \setminus D}$ directly from $P \setminus D$, disregarding T^ω_P. This paper studies how to solve the reason maintenance problem *incrementally* by leveraging information available from the fixpoint computed before base facts or rules change.

First, we introduce so called support graphs – a framework that allows for a unified description of the presented methods. Then we define three extended

immediate consequence operators and formulate their properties. In Section 4, we then provide reason maintenance methods based on these operators.

2 Preliminaries

Throughout this paper a Datalog [27] rule language is assumed with range restricted rules (i.e. a variable occurring in the rule head occurs in the rule body too). The formulas $\top$ and $\bot$ respectively evaluate to true and false in all interpretations. 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. The usual definition of the immediate consequence operator T_P and of its ordinal powers is assumed, see e.g. [19]. 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 $\text{atoms}(r) = \text{body}(r) \cup \{\text{head}(r)\}$. Let HB be the Herbrand base, HU the Herbrand universe, and P a program. $HU_P \subseteq HU$ is the set of ground terms that can be constructed with symbols occurring in P. $HB_P \subseteq HB$ comprises all ground atoms that can be constructed with symbols occurring in P. Note that HU_P and HB_P are finite for a *Datalog* program P.

A *support* with respect to a definite program P is a pair $s = (r, \sigma)$, where $r \in P$ and σ is a substitution such that $\text{dom}(\sigma) \subseteq \text{var}(r)$. Let $r = A \leftarrow B_1, \ldots, B_n$. Then $\text{head}(s) = A\sigma$ is the *head* of s, and the set $\text{body}(s) = \{B_1\sigma, \ldots, B_n\sigma\}$ is called the *body* of s. If $n = 0$ then s is called a *base* support, otherwise it is called a *derived* support. Furthermore, $\text{heads}(S) = \{\text{head}(s) \mid s \in S\}$ for a set of supports S. A support is ground if its head and its body are ground. We say that s supports $\text{head}(s)$.

The number of supports of an atom is infinite in general. The number of ground supports with respect to HB_P is finite for a Datalog program P because the codomains of their substitutions are subsets of HU_P which is finite. The notion of a support is different from the notion of a rule instance. Consider the program $P = \{r_1 = p(a,a) \leftarrow \top. \quad r_2 = q(x) \leftarrow p(x,x). \quad r_3 = q(y) \leftarrow p(a,y)\}$. The atom $q(a)$ is derivable both via rule r_2 and r_3 from the base fact r_1. In both cases the resulting rule instance is $q(a) \leftarrow p(a,a)$. However there are two different ground supports $(r_2, \{x \mapsto a\})$ and $(r_3, \{y \mapsto a\})$ which allows for distinguishing the two different derivations of $q(a)$.

Several operators on multisets are introduced later in this paper. There are a number of different definitions of multisets [26,5]. We adopt the definitions from [8] of extended multisets that allow infinite multiplicities. Infinity completes the total ordering $\leq$ on $\mathbb{N}_0$ so that each subset of $\mathbb{N}_0^\infty$ has the least upper bound (and also the greatest lower bound) in $\mathbb{N}_0^\infty$. Let D be a non-empty set. A *multiset* on D is a pair $A = (D, \mu)$, where $\mu : S \to \mathbb{N}_0^\infty$, *multiplicity*, is a function from D, the domain set, to the set $\mathbb{N}_0^\infty$. The empty multiset is denoted[1] $\emptyset_m^D$. The root [4] of a multiset $A = (D, \mu)$ is the set $\text{root}(A) = \{d \in D \mid \mu(d) > 0\}$. A multiset can be infinite in two different ways: its root is infinite or it contains an element with infinite multiplicity. Let $A = (D, \mu_A), B = (D, \mu_B)$ be multisets

[1] The domain set superscript may be omitted where it is clear from context.

and $a \in D$. Let $a \in^i_m (D, \mu)$ iff $a \in D$ and $\mu(a) = i$ and $a \in_m (D, \mu)$ iff there is an $i > 0$ such that $a \in^i_m (D, \mu)$. Submultiset ($A \subseteq_m B$), proper submultiset ($A \subset_m B$), multiset union ($A \cup_m B$), multiset intersection ($A \cap_m B$), multiset sum ($A \uplus B$), and multiset removal ($A \ominus B$) are defined as expected, cf. [8]. We denote multisets by square brackets "[ ]" and sets by braces "{ }", e.g. $A = [a, a, b, c] = (\{a, b, c\}, \{(a, 2), (b, 1), (c, 1)\})$, $\text{root}(A) = \{a, b, c\}$.

Analogically to the set-builder (also set comprehension) notation $\{x \mid P(x)\}$, the multiset-builder notation $[x \mid P(x)]$ is used to specify multisets. $P(x)$ is called the *(multiset-)builder condition* and $[x \mid P(x)]$ is the multiset of individuals that satisfy the condition $P(x)$. It is assumed that whenever the $\in_m$ relationship is used in the multiset-builder condition the multiplicities are transferred to the resulting multiset. Also, in cases like $[x \mid \exists y P(x, y)]$, the multiplicity of x in the resulting multiset is $|\{y \mid P(x, y)\}|$. For example, let $A = \{a\}$ and $N = [n, n, n]$. Then $N_1 = [\{x, y\} \mid x \in A, y \in_m N] = [\{a, n\}, \{a, n\}, \{a, n\}]$, $N_2 = [\{x, y\} \mid x \in A, y \in \text{root}(N)] = [\{a, n\}]$, $N_3 = \big\{\{x, y\} \mid x \in A, y \in_m N\big\} = \{\{a, n\}\}$. Also, $n \in_m N$, $n \in^3_m N$, $\{a, n\} \in^3_m N_1$, $\{a, n\} \in^1_m N_2$, $\{a, n\} \in N_3$.

The *multiset powerset*, $\mathbb{P}(S)$, of a set S is the set of all multisets on S, i.e. $\mathbb{P}(S) = \{(S, \mu) \mid \mu : S \to \mathbb{N}^\infty_0\}$, $\mathbb{P}(\emptyset) = \emptyset$. Let $S = \{a, b, c\}$. Then $\mathbb{P}(S) = \{(S, \mu) \mid \mu : S \to \mathbb{N}^\infty_0\} = \{\emptyset_m, [a], [a, a], \ldots, [b], [b, b], \ldots, [c], [c, c], \ldots, [a, b], [a, a, b], \ldots, [a, b, b], \ldots, [a, b, c], [a, a, b, c], \ldots, [a, b, b, c], \ldots, [a, b, c, c], \ldots, [a, a, b, b, c], \ldots\}$. The submultiset relation $\subseteq_m$ is a partial order on $\mathbb{P}(S)$. $(\mathbb{P}(S), \subseteq_m)$ is a complete lattice, see Proposition 23 in Casasnovas et al. [8], which is important as it allows us to apply Lloyd's classical fixpoint theory for operators on complete lattices [19] to our multiset operators. *If an operator X is continuous on a complete lattice then naïve forward chaining with X computes its least fixpoint* [19]. *This result can be used in proving correctness of semi-naïve forward chaining with a semi-naïve mapping corresponding to X.*

2.1 Support Graphs

A support graph is a data structure inspired by data-dependency networks of classical reason maintenance [11]. In contrast to data-dependency networks, support graphs can accommodate a notion of derivation which is close to the notion of a proof from assumptions in classical mathematical logic. This allows for a unified description of reasoning and reason maintenance methods presented here. The proposed definition of a derivation has the advantage that the number of derivations with respect to range restricted Datalog programs is always finite which is in contrast to established counting reason maintenance methods: for example in [23], the number of derivation trees (a notion defined in [20]) can be infinite. P is assumed to be a definite range restricted Datalog program in the rest of the paper unless explicitly stated otherwise. Support graphs are introduced only informally here, see [18] for details.

A *support graph* with respect to P is a bipartite directed graph where nodes are ground atom nodes and ground supports connected by edges according to the structure of the supports (see examples that follow). Atom nodes are labelled by atoms, i.e. an atom may be represented by multiple nodes in a support

graph. A support graph is *compact* if no two atom nodes have the same label. A support graph induced by a set of supports S, denoted $\mathrm{SG}(S)$, is the compact support graph corresponding to the supports in S. In diagrams, atom nodes are represented as dots and supports as ovals. Base supports are emphasised by the symbol $\top$ and no arrow ($\top$ is neither a node nor an edge in the s.g.), see Figure 1. A support graph G is *homomorphically embedded* in a support graph H if there is a graph homomorphism from G to H that respects labellings, see Figure 2. A *path* in G is an alternating sequence of adjacent supports and atom nodes. If there is a path from a node x to a node y in G, we say that x *supports* y in G and y *depends on* x in G. If the path has length 1 we may add the adverb "directly." If the path has length > 1 we may add the adverb "indirectly." A node in a support graph depends on a rule if it is a support that includes the rule or it depends on a support in the graph that includes the rule. A support graph is *well-founded* if it is acyclic and each atom node has an incoming edge. Note that all nodes with no incoming edge of a well-founded support graph are base supports. Being acyclic, a well-founded support graph has at least one node with no outgoing edge. A *derivation* of an atom a in a s.g. G is a minimal (w.r.t. $\subseteq$) well-founded s.g. D that is homomorphically embedded in G and has a node labelled a as its only sink node. The depth of D is the number of supports on the longest path in D that ends with the only sink node. A *support* $s \in S$ is *well-founded in* G if there exists a derivation of $\mathrm{head}(s)$ in G that includes s. A node x in G *depends strongly* on a node y in G if y is in every derivation of x in G. Consider Figure 2. Graphs D, E, F, and G are all derivations of d in H. H is not a derivation because e.g. its proper subgraph D is well-founded. The graph in Figure 1 is not well-founded for two reasons: it is cyclic and its atom $q(b, b)$ has no incoming edge. Let G' be a supergraph of G with an additional support $s_4 = (r_4, \emptyset)$. Then G' is not well-founded, but s_3 is well-founded in G' because G' without s_2 is a derivation of $head(s_3)$ in G'. The only support that is well-founded in G is s_1.

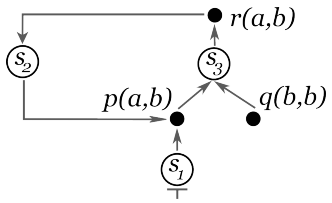
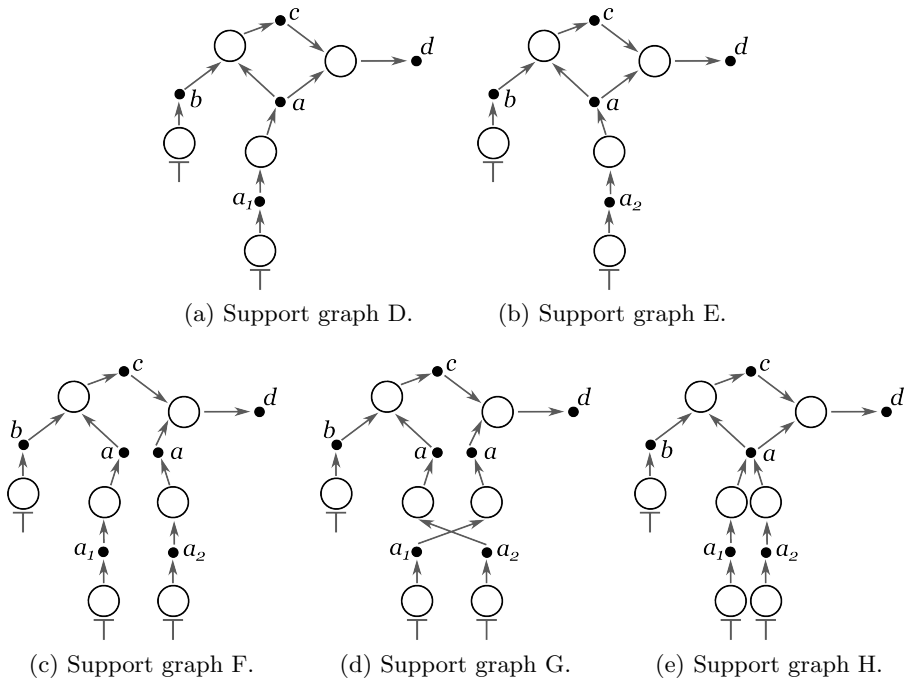


Fig. 1: The compact support graph $\mathrm{SG}(S)$ for $P = \{r_1 = p(a, w) \leftarrow \top.\quad r_2 = p(u, v) \leftarrow r(u, v).\quad r_3 = r(x, z) \leftarrow p(x, y), q(y, z).\quad r_4 = q(b, b) \leftarrow \top\}$ and $S = \{s_1 = (r_1, \{w \mapsto b\}), s_2 = (r_2, \{u \mapsto a, v \mapsto b\}), s_3 = (r_3, \{x \mapsto a, y \mapsto b, z \mapsto b\})\}$, where u, v, w, x, y are variables and a, b are constants.

The number of derivations of a from P is finite up to renaming of atom nodes. Intuitively, a support graph consists only of ground supports and there is



(a) Support graph D. (b) Support graph E.

(c) Support graph F. (d) Support graph G. (e) Support graph H.

Fig. 2: Compact support graphs D and E and non-compact support graphs F and G are homomorphically embedded in the compact support graph H. All are with respect to the propositional definite program $P = \{d \leftarrow a, c.\quad c \leftarrow a, b.\quad a \leftarrow a_1.\quad a \leftarrow a_2.\quad a_1 \leftarrow \top.\quad a_2 \leftarrow \top.\quad b \leftarrow \top\}$.

only a finite number of them with respect to a range restricted Datalog definite program P and thus there is only a finite number of derivations w.r.t. P.

3 Extended Immediate Consequence Operators

The support Herbrand base sHB is the set of all ground supports. sHB_P is the set of all ground supports with respect to P.

Definition 1. *The support keeping immediate consequence operator skT_P for P is the mapping $skT_P : \mathcal{P}(sHB_P) \rightarrow \mathcal{P}(sHB_P)$ defined as $skT_P(F) = \{s \in sHB_P \mid s = (r, \sigma), r = H \leftarrow B_1, \ldots, B_n \in P, \mathrm{dom}(\sigma) = \mathrm{var}(r), \mathrm{body}(s) \subseteq \mathrm{heads}(F)\}$.*

The skT_P operator is continuous and it is easy to see that $T_P^i = \mathrm{heads}(skT_P^i)$, for all $i \in \mathbb{N}_0$, and $T_P^\omega = \mathrm{heads}(skT_P^\omega)$. A modified classical naïve forward chaining algorithm where T_P is replaced with skT_P computes skT_P^ω. A semi-naïve mapping $skT_P(F, \Delta) : \mathcal{P}(sHB_P) \times \mathcal{P}(HB_P) \rightarrow \mathcal{P}(sHB_P)$ can be defined by adding the condition $(\exists 1 \leq j \leq n) B_j\sigma \in \Delta$ to Definition 1.

Algorithm 1.1: Semi-naïve support keeping forward chaining

```
Input: P  Initialization: F := ∅; Δ_F := skT_P(∅); Δ := skT_P(∅)
while Δ ≠ ∅
  F := F ∪ Δ_F
  Δ_F := skT_P(F, heads(Δ))                    {new supports}
  Δ := Δ_F \ {s ∈ Δ_F | head(s) ∈ heads(F)}{excl. redundancies,cycles}
return F
```

Well-foundednes is important because of the following fact: a ground atom a is in T_P^ω iff a has a support w.r.t. P that is well-founded in $\mathrm{SG}(skT_P^\omega)$. Additionally, each support of an atom $g \in T_P^\omega$ not well-founded in $G = \mathrm{SG}(skT_P^\omega)$ *depends strongly* on g in G.

Definition 2. *The* support counting immediate consequence operator scT_P *for* P *is a mapping* $scT_P : \mathbb{P}(HB_P) \to \mathbb{P}(HB_P)$ *defined as* $scT_P(S) = [H\sigma \in HB_P \mid \exists\, s = (r, \sigma), r = H \leftarrow B_1, \ldots, B_n \in P, \mathrm{body}(s) \subseteq \mathrm{root}(S), \mathrm{dom}(\sigma) = \mathrm{var}(r) \quad]$.

The "count" is represented by the multiplicities of ground atoms in the multiset. It can be shown by induction on i that $T_P^i = \mathrm{root}(scT_P^i)$, for all $i \in \mathbb{N}_0$ and also $T_P^\omega = \mathrm{root}(scT_P^\omega)$. Again, scT_P is continuous and a corresponding semi-naïve mapping can be defined by adding the condition $(\exists 1 \leq j \leq n) B_j\sigma \in \Delta$ to Definition 2.

Algorithm 1.2: Semi-naïve support counting forward chaining

```
Input: P  Initialization F := ∅_m; Δ := T_P(∅); Δ_F := scT_P(∅_m)
while Δ ≠ ∅
  F := F ⊎ Δ_F
  Δ_F := scT_P(F, Δ)          {new atoms}
  Δ := root(Δ_F) \ root(F)  {exclude redundancies, cycles}
return F
```

An *extended atom* is a pair (a, D), where $a \in HB_P$ and $D \subseteq HB_P$. D is called a *dependency* of a. The *derivation counting Herbrand base* (w.r.t. P) is the set $dHB_P = HB_P \times \mathcal{P}(HB_P)$. It follows that dHB is finite when HB is finite.

Definition 3. *The* derivation counting immediate consequence operator dcT_P *for* P *is the mapping:* dcT_P: $\mathbb{P}(dHB_P) \to \mathbb{P}(dHB_P)$ *defined as* $dcT_P(S) = [\ (H\sigma, D) \in dHB_P \mid (\exists s = (r, \sigma)), \mathrm{dom}(\sigma) = \mathrm{var}(r), r = H \leftarrow B_1, \ldots, B_n \in P; (\forall 1 \leq i \leq n)\ (\exists D_i)\ (B_i\sigma, D_i) \in_m S; H\sigma \notin D_i, H\sigma \neq B_i\sigma, D = \bigcup_{i=1}^n D_i \cup \{B_i\sigma\}\ ]$.

dcT_P is continuous. A corresponding semi-naïve mapping can be defined by adding the condition $(\exists 1 \leq j \leq n)(B_j\sigma, D_j) \in_m \Delta$ to Definition 3. The derivation count is represented by the multiplicities of extended atoms in the multiset.

Algorithm 1.3: Semi-naïve derivation counting forward chaining

```
Input: P  Initialization: F := ∅_m ; Δ := dcT_P(∅_m)
while Δ ≠ ∅_m
  F := F ⊎ Δ
  Δ := dcT_P(F, Δ) {cycles are handled by dcT_P itself}
return F
```

Proposition 1. *$(a, D) \in^x_m dcT^\omega_P$ iff there are x derivations of a with respect to P that use all and only atoms in D.*

Corollary 1. *The number of derivations of a with respect to P is* $\sum_{\exists D (a,D) \in^x_m dcT^\omega_P} x$

Proposition 2. *Let P be a definite Datalog program. Then scT^ω_P and dcT^ω_P are multisets with finite multiplicities.*

It is easy to see that skT_P, scT_P, and dcT_P operators all derive at least the information derived by the T_P operator. Therefore Herbrand interpretations induced by the extended fixpoints can straightforwardly be defined so that they are the same as those induced by T^ω_P.

The operators allow for a formal and declarative specification how to derive additional information that can be used for explanation and more efficient reason maintenance. Supports, support counts, and derivation counts provide information about derivability of atoms.

It is well-known [21] that T^ω_P can be computed in $O(n^k)$ time for a range restricted Datalog program P, where n is the number of constants in base facts and k is the maximum over all rules in P of the number of variables in that rule. It is easy to see that the worst case time complexity of Algorithms 1.1 and 1.2 is also $O(n^k)$. For Algorithm 1.3 it is $O(n^{2k})$ (the $O(n^k)$ computation may have to be repeated up to $O(n^k)$ times).

4 Reason Maintenance

4.1 Support Counting

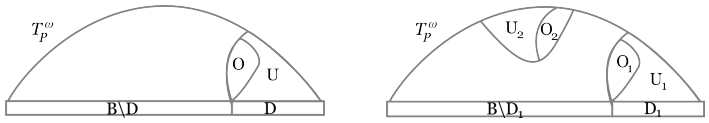


Fig. 3: Illustration how the sets U, K, and O relate. B is the set of base facts in P. The left diagram assumes that D consists of base facts. The right diagram is more general: sets U_2 and O_2 correspond to rules in D that are not base facts.

Let us review a reason maintenance technique without support graphs [1] that can be improved upon by support counting. Let $D \subseteq P$ be a set of rules

to remove. Let $U \subset T_P^\omega$ be the set of ("*u*nsure") atoms that depend on a rule from D in $\mathrm{SG}(skT_P^\omega)$, See Figure 3. Let $K = T_P^\omega \setminus U$ ("atoms to *k*eep"), and $O = U \cap T_{P\setminus D}^\omega$ ("*o*therwise supported atoms"). It holds that $T_{P\setminus D}^\omega = K \cup O$, i.e. the new fixpoint can be computed by determining U and K and then by semi-naïve forward chaining on K, i.e. $T_{P\setminus D}^\omega = T_P^\omega(K)$. The semi-naïve forward chaining however includes a naïve initialization step: $\Delta := T_{P\setminus D}(K) \setminus K$. This Δ includes the atoms in O that are derivable in one step from K and $P \setminus D$ and it can be determined directly if support counts are kept.

Lemma 1. *Let $SU \subseteq skT_P^\omega$ be the supports that depend on a rule from D. Let $a \in_m^x scT_P^\omega$. Then $a \in O \cap T_{P\setminus D}(K)$ iff $x > |\{t \in SU \mid \mathrm{head}(t) = a\}|$.*

The lemma intuitively says that a is in $O \cap T_{P\setminus D}$ iff it has a support in skT_P^ω that does not depend on a rule from D. Such a support is an evidence of derivability of a from $P \setminus D$ because it necessarily is well-founded: it can be shown that 1) an atom is derivable iff it has a well-founded support, and 2) a not well-founded support strongly depends on the support's head. From 2) it follows that each not well-founded support of a must be in SU which means that any extra support of a that is not in SU is well-founded which by 1) means that a is derivable. Algorithm 1.4 uses this result to replace the naïve initialization step of [1] with a comparison of support counts.

Algorithm 1.4: scT_P-based reason maintenance without support graphs

```
Input: scT_P^ω, D
Initialization: T_P^ω := root(scT_P^ω),
 SU:={s ∈ skT_P^ω | s deps.on an r ∈ D}, K:=[a | a ∈_m scT_P^ω and a ∉ heads(SU)],
 Δ_F := [ a ∈ heads(SU) | a ∈_m^x scT_P^ω and x > |{t ∈ SU | head(t) = a}| ],
 Δ := root(Δ_F), F := K
while Δ ≠ ∅
  F := F ⊎ Δ_F
  Δ_F := scT_{P\D}(F, Δ)          {repetitions add to F}
  Δ := root(Δ_F) \ root(F)        {forbid cyclic derivations}
return F
```

Support Counting for Non-recursive Datalog Programs The $\mathrm{SG}(skT_P^\omega)$ graph has no cycles for non-recursive P. This allows for a specialized incremental reason maintenance algorithm that processes only atoms that are eventually removed or at most one step further, see Algorithm 1.5. Algorithm 1.5 only processes atoms in O, Algorithm 1.4 (and the version in [1]) all atoms in U.

Algorithm 1.5: Reason maintenance with support counts and no cycles.

```
Input P, D, scT_P^ω
Initialization F := ∅_m, Δ_F := scT_D(scT_P^ω),
  Δ := {a | (∃x ∈ ℕ_1)a ∈_m^x scT_P^ω and a ∈_m^x Δ_F}, Δ_Σ := Δ
```

```
while Δ ≠ ∅
  F := F ⊎ Δ_F
  Δ_F := scT_P(scT_P^ω, Δ) {atoms losing a support}
  Δ := {a | (∃x ∈ ℕ_1)a ∈_m^x scT_P^ω and a ∈_m^x F ⊎ Δ_F} \ Δ_Σ
  Δ_Σ := Δ_Σ ∪ Δ        {atoms that lost all supports}
return scT_P^ω ⊖ F
```

In the initialization, Algorithm 1.5 first determines atoms that directly depend on a rule to remove ($scT_D(scT_P^\omega)$) and out of these it takes atoms (Δ) that lose all supports and therefore are not in the new fixpoint and it propagates their deletion. The while cycle works similarly, with the difference that any affected atom directly depends on an atom that lost all supports in the previous iteration (Δ_F is computed using the scT_P mapping and Δ, notice the set P).

4.2 Derivation Counting

Reason maintenance becomes a simple task in the case of *base fact* updates when a dcT_P fixpoint is available. Such a fixpoint provides information about derivations of any atom in the form of extended atoms. Any atom derived from a base fact has a corresponding extended atom in the dcT_P fixpoint that has the base fact in its dependency. To compute the new fixpoint for a base fact removal, it is only necessary to remove all extended atoms from the fixpoint that have the base fact in its dependency.

Algorithm 1.6: Reason maintenance for base fact updates using derivation counts

```
Input P, dcT_P^ω, D
Inv := [(a, S) | (a, S) ∈_m dcT_P^ω and ((S = ∅) or (∃d ∈ heads(D)) d ∈ S)]
return dcT_P^ω ⊖ Inv {remove invalidated atoms}
```

Derivation counting does not keep track of rules that are used to derive an atom. Reason maintenance for removing rules that are not base facts therefore requires either tracking rules as well or a more sophisticated approach. The dcT_P operator and the current fixpoint can be used to determine all extended atoms that were derived using a rule to remove: $dcT_D(dcT_P^\omega)$. All derivations that use any atom from this multiset are invalidated. The corresponding extended atoms can be computed by semi-naïve dcT_P forward-chaining analogously to determining the set SU in [1].

Note that counting supports is inherently more efficient than counting derivations as it can be done by extending standard semi-naïve forward chaining to keep count of generated rule instances per atom. In contrast, counting derivations is in general akin to generating all derivations. From this perspective, counting supports of an atom amounts to determining the in-degree of the atom node in the respective compact support graph while counting derivations amounts to generating all specific subtrees (derivations) of the support graph.

5 Related Work

The reason maintenance problem is essentially a problem of changing knowledge. As such, it is related to literature ranging from epistemology, to logic, to databases. One of the overarching concepts is defeasible reasoning [17] which includes two subfields related to this work: belief revision and reason maintenance.

Belief revision (also called *AGM theory*) [2,3] makes no distinction between base and derived facts and it aims at revising a theory so that only a minimal change occurs in its deductive closure. In contrast, 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* [11,22,10] refers to knowledge base update techniques which keep a record of derivations in form of a "data dependency network." Support graphs can be seen as a novel extension of data dependency networks that can represent (non-compact) derivations. One of the reason maintenance techniques has been specialized to RDF(S) reasoning and implemented by Broekstra et al. [6]. Belief revision and reason maintenance are closely related and are compared in the literature [12].

The reason maintenance problem has been studied in the field of *deductive databases* under the name *incremental view maintenance* on and off since around 1980, see for example a survey article [14] by Gupta and Mumick, authors of the probably most popular DRed algorithm [15]. The DRed algorithm is similar to the one described in [1] but does not directly handle rule updates and requires program transformations that increase its size. The PF algorithm [16] is similar to DRed. Both work on the same principle of deriving an overestimation of deleted atoms and then finding alternative derivations for them, cf [9] for a comparison. In contrast to DRed and PF, the methods presented in this paper use the original unchanged program. Staudt and Jarke developed in [25] a purely declarative version of DRed. Their algorithm transforms the original program into even more rules than DRed or PF and can add negation even if the original program is definite. Volz, Staab, and Motik extended [28] Staudt and Jarke's version of DRed to handle rule changes and applied the resulting method to reasoning on the Semantic Web. Their version of DRed transforms 12 RDF semantics Datalog rules into a maintenance program of 60 rules [28]. Their method leads to a 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 on the Semantic Web where this kind of axiomatization is very common. In comparison, the methods presented here make do with the original 12 rules, always only the relevant part of a predicate's extension is recomputed, and no modification is necessary to handle general rule updates in the case of the scT_P-based methods.

Duplicate semantics [23] of Datalog programs counts so called *derivation trees* defined by Maher in [20]. Maher's definition allows for repetitions which is excluded in our definition of derivation by the minimality condition. As a result, it is possible that an atom has an infinite number of Maher's derivation trees while the number of derivations is always finite for Datalog programs in our case. The main problem, detecting cycles in support graphs of recursive Datalog

programs, is the same in both cases and is studied also in [24]. In this respect, scT_P and dcT_P provide novel duplicate semantics. Moreover, the scT_P-based multiset semantics can be computed more efficiently than the original duplicate semantics and to best of our knowledge it has not yet been described in the literature. A similar counting approach for non-recursive Datalog programs is described in [13] – it is less efficient than our method because it counts derivation trees as opposed to counting just the in-degree of atom nodes.

6 Conclusion

We have presented novel methods for incremental maintenance of materialized Datalog views. Our methods handle changes in both facts and rules, works by evaluating the original and the target programs and recomputes only the affected part of a predicate's extension. The methods are applicable to important Datalog-fragments of Semantic Web languages. An extension to stratifiable normal Datalog programs is straightforward and while aggregation is not handled by our current methods, we are confident that the extension is possible.

It should be stressed that one reason maintenance method may not fit all needs especially in a system containing as diverse kinds of information as a semantic wiki. For example, semantic wikis employ RDF(S) and OWL ontologies and often use many different ontologies to represent different kinds of data: some important to users some not. As Weaver and Hendler showed [29], it is often easy to split RDF(S) data in parts for which the fixpoint can be computed in parallel. A similar strategy can be taken for reasoning and reason maintenance in a semantic wiki. For a part of the data not directly visible to users only the T_P^ω fixpoint may be computed, for other parts one of the skT_P^ω, scT_P^ω, or dcT_P^ω fixpoints may be computed which provide additional information with each atom that can be used for *explanation*. In all cases, one of the reason maintenance methods that takes advantage of the particular fixpoint can be used.

Acknowledgements. 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. Reasoning as Axioms Change: Incremental View Maintenance Reconsidered. Proc. RR 2011 (2011)
2. Alchourron, C.E., Gardenfors, P., Makinson, D.: On the logic of theory change: Contraction functions and their associated revision functions. Theoria 48 (1982)
3. Alchourron, C.E., Gardenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. J. Symbolic Logic (1985)
4. Blizard, W.: Dedekind multisets and function shells. Theoretical Computer Science 110(1), 79–98 (March 1993)
5. Blizard, W.D.: The development of multiset theory. The Review of Modern Logic 1(4), 319–352 (1991)

6. 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)
7. 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)
8. Casasnovas, J., Mayor, G.: Discrete t-norms and operations on extended multisets. Fuzzy Sets and Systems 159(10), 1165–1177 (May 2008)
9. Dietrich, S.W.: Maintenance of Recursive Views. In: Encyclopedia of Database Systems, pp. 1674–1679. Springer Verlag (2009)
10. Doyle, J.: Truth maintenance systems for problem solving. Tech. Rep. AI-TR-419, Dep. of Electrical Engineering and Computer Science of MIT (1978)
11. Doyle, J.: The ins and outs of reason maintenance. Proc. IJCAI'83 (1983)
12. Doyle, J.: Reason maintenance and belief revision – Foundations vs. Coherence theories. Cambridge University Press (1992)
13. Gupta, A., Katiyar, D., Mumick, I.S.: Counting solutions to the View Maintenance Problem. In: In Workshop on Deductive Databases, JICSLP. pp. 185–194 (1992)
14. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. Data Engineering Bulletin 18(2), 3–18 (1995)
15. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. SIGMOD Rec. 22, 157–166 (June 1993)
16. 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)
17. Koons, R.: Defeasible reasoning. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy (Spring 2005)
18. Kotowski, J.: Constructive Reasoning for Semantic Wikis. Ph.D. thesis (2011)
19. Lloyd, J.: Foundations of Logic Programming. Berlin: Springer-Verlag (1987)
20. Maher, M., Ramakrishnan, R.: Déjà vu in fixpoints of logic programs. In: North American Conference on Logic Programming (1989)
21. McAllester, D.: On the complexity analysis of static analyses. J. ACM 49(4), 512–537 (2002)
22. McAllester, D.A.: Truth maintenance. AAAI90 (1990)
23. Mumick, I.S.: Query Optimization in Deductive and Relational Databases. Ph.D. thesis, Stanford University (1991)
24. Mumick, I.S., Shmueli, O.: Finiteness Properties of Database Queries. Fourth Australian Database Conference (1993)
25. Staudt, M., Jarke, M.: Incremental Maintenance of Externally Materialized Views. In: Proc. 22th Int. Conf. VLDB. San Francisco, CA, USA (1996)
26. Syropoulos, A.: Mathematics of multisets. In: Multiset Processing, LNCS, vol. 2235. Springer Berlin / Heidelberg, Berlin, Heidelberg (December 2001)
27. Ullman, J.D.: Principles of database and knowledge-base systems. Computer Science Press (1989)
28. 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)
29. Weaver, J., Hendler, J.: Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In: Proc. ISWC2009 (October 2009)