Optimised Maintenance of Datalog Materialisations

Pan Hu, Boris Motik, Ian Horrocks
Department of Computer Science, University of Oxford
Oxford, United Kingdom
firstname.lastname@cs.ox.ac.uk

Abstract

To efficiently answer queries, datalog systems often materialise all consequences of a datalog program, so the materialisation must be updated whenever the input facts change. Several solutions to the materialisation update problem have been proposed. The Delete/Rederive (DRed) and the Backward/Forward (B/F) algorithms solve this problem for general datalog, but both contain steps that evaluate rules ‘backwards’ by matching their heads to a fact and evaluating the partially instantiated rule bodies as queries. We show that this can be a considerable source of overhead even on very small updates. In contrast, the Counting algorithm does not evaluate the rules ‘backwards’, but it can handle only non-recursive rules. We present two hybrid approaches that combine DRed and B/F with Counting so as to reduce or even eliminate ‘backward’ rule evaluation while still handling arbitrary datalog programs. We show empirically that our hybrid algorithms are usually significantly faster than existing approaches, sometimes by orders of magnitude.

1 Introduction

Datalog (Abiteboul, Hull, and Vianu 1995) is a rule language that is widely used in modern information systems. Datalog rules can declaratively specify tasks in data analysis applications (Luteberget, Johansen, and Steffen 2016; Piro et al. 2016), allowing application developers to focus on the objective of the analysis—that is, on specifying what needs to be computed rather than how to compute it (Markl 2014). Datalog can also capture OWL 2 RL (Horrocks et al. 2004). It is implemented in systems such as WebPIE (Urbani et al. 2012), VLog (Urbani, Jacobs, and Krötzsch 2016), Oracle’s RDF Store (Wu et al. 2008), OWLIM (Bishop et al. 2011), and RDFox (Nenov et al. 2015), and it is extensively used in practice.

When performance is critical, datalog systems usually precompute the materialisation (i.e., the set of all consequences of a program and the explicit facts) in a preprocessing step so that all queries can later be evaluated directly over the materialisation. Recomputing the materialisation from scratch whenever the explicit facts change can be expensive. Systems thus typically use an incremental maintenance algorithm, which aims to avoid repeating most of the work. Fact insertion can be effectively handled using the seminaive algorithm (Abiteboul, Hull, and Vianu 1995), but deletion is much more involved since one has to check whether deleted facts have derivations that persist after the update. The Delete/Rederive (DRed) algorithm (Gupta, Mumick, and Subrahmanian 1993; Staudt and Jarke 1996), the Backward/Forward (B/F) algorithm (Motik et al. 2015), and the Counting algorithm (Gupta, Mumick, and Subrahmanian 1993) are well-known solutions to this problem.

The DRed algorithm handles deletion by first overdeleting all facts that depend on the removed explicit facts, and then rederiving the facts that still hold after overdeletion. The rederivation stage further involves rederiving all overdeleted facts that have alternative derivations, and then recomputing the consequences of the rederived facts until a fixpoint is reached. The algorithm and its variants have been extensively used in practice (Urbani et al. 2013; Ren and Pan 2011). In contrast to DRed, the B/F algorithm searches for alternative derivations immediately (rather than after overdeletion) using a combination of backward and forward chaining. This makes deletion exact and avoids the potential inefficiency of overdeletion. In practice, B/F often, but not always, outperforms DRed (Motik et al. 2015).

Both DRed and B/F search for derivations of deleted facts by evaluating rules ‘backwards’: for each rule whose head matches the fact being deleted, they evaluate the partially instantiated rule body as a query; each query answer thus corresponds to a derivation. This has two consequences. First, one can examine rule instances that fire both before and after the update, which is redundant. Second, evaluating rules ‘backwards’ can be inherently more difficult than matching the rules during initial materialisation: our experiments show that this step can, in some cases, prevent effective materialisation maintenance even for very small updates.

In contrast, the Counting algorithm (Gupta, Mumick, and Subrahmanian 1993) does not evaluate rules ‘backwards’, but instead tracks the number of distinct derivations of each fact: a counter is incremented when a new derivation for the fact is found, and it is decremented when a derivation no longer holds. A fact can thus be deleted when its counter drops to zero, without the potentially costly ‘backward’ rule evaluation. The algorithm can also be made optimal in the...
sense that it considers precisely the rule instances that no longer fire after the update and the rule instances that only fire after the update. The main drawback of Counting is that, unlike DRed and B/F, it is applicable only to nonrecursive rules (Nicolas and Yazdanian 1983). Recursion is a key feature of datalog, allowing one to express common properties such as transitivity. Thus, despite its favourable properties, the Counting algorithm does not provide us with a general solution to the materialisation maintenance problem.

Towards the goal of developing efficient general-purpose maintenance algorithms, in this paper we present two hybrid approaches that combine DRed and B/F with Counting. The former tracks the nonrecursive and the recursive derivations separately, which allows the algorithm to eliminate all ‘backward’ rule evaluation and also limit overdeletion. The latter tracks nonrecursive derivations only, which eliminates ‘backward’ rule evaluation for nonrecursive rules; however, recursive rules can still be evaluated ‘backwards’ to eagerly identify alternative derivations. Both combinations can handle recursive rules, and they exhibit ‘pay-as-you-go’ behaviour in the sense that they become equivalent to Counting on nonrecursive rules. Apart from the modest cost of maintaining counters, our algorithms never involve more computation steps than their unoptimised counterparts. Thus, our algorithms combine the best aspects of DRed, B/F, and Counting: without incurring a significant cost, they eliminate or reduce ‘backward’ rule evaluation, are optimal for nonrecursive rules, and can also handle recursive rules.

We have implemented our hybrid algorithms and have compared them with the original DRed and B/F algorithms on several synthetic and real-life benchmarks. Our experiments show that the cost of counter maintenance is negligible, and that our hybrid algorithms typically outperform existing solutions, sometimes by orders of magnitude. Our test system and datasets are available online.\footnote{http://krr-nas.cs.ox.ac.uk/2017/counting/}

2 Preliminaries

We now introduce datalog with stratified negation. We fix countable, disjoint sets of constants and variables. A term is a constant or a variable. A vector of terms is written \( \vec{t} \), and we often treat it as a set. A (positive) atom has the form \( P(t_1, \ldots, t_k) \) where \( P \) is a \( k \)-ary predicate and each \( t_i \), \( 1 \leq i \leq k \), is a term. A term or an atom is ground if it does not contain variables. A fact is a ground atom, and a dataset is a finite set of facts. A rule has the form

\[
B_1 \land \cdots \land B_m \land \neg \text{not } B_{m+1} \land \cdots \land \neg \text{not } B_n \rightarrow H,
\]

where \( m \geq 0, n \geq 0, \) and \( B_i \) and \( H \) are atoms. The head \( h(r) \) of \( r \) is the atom \( H \), the positive body \( b^+(r) \) of \( r \) is the set of atoms \( B_1, \ldots, B_m \), and the negative body \( b^-(r) \) of \( r \) is the set of atoms \( B_{m+1}, \ldots, B_n \). Rule \( r \) must be safe: each variable occurring in \( r \) must occur in a positive body atom.

A substitution \( \sigma \) is a mapping of finitely many variables to constants. For \( \alpha \) a term, literal, rule, conjunction, or a vector or set thereof, \( \alpha \sigma \) is the result of replacing each occurrence of a variable \( x \) in \( \alpha \) with \( \sigma(x) \) (if the latter is defined).

A stratification \( \lambda \) of a program \( \Pi \) maps each predicate of \( \Pi \) to a positive integer such that, for each rule \( r \in \Pi \) with \( h(r) = P(\vec{t}) \), (i) \( \lambda(P) \geq \lambda(R) \) for each atom \( R(\vec{s}) \in b^+(r) \), and (ii) \( \lambda(P) > \lambda(R) \) for each atom \( R(\vec{s}) \in b^-(r) \). Program \( \Pi \) is stratifiable if a stratification \( \lambda \) of \( \Pi \) exists. A rule \( r \) with \( h(r) = P(\vec{t}) \) is recursive w.r.t. \( \lambda \) if an atom \( R(\vec{s}) \in b^+(r) \) exists such that \( \lambda(P) = \lambda(R) \); otherwise, \( r \) is nonrecursive w.r.t. \( \lambda \). For each positive integer \( s \), program \( \Pi_s = \{ r \in \Pi \mid \lambda(h(r)) = s \} \) is the stratum \( s \) of \( \Pi \), and programs \( \Pi_0 \) and \( \Pi_{\infty} \) are the recursive and the nonrecursive subsets, respectively, of \( \Pi \). Finally, \( \Omega^* \) is the set of all facts that belong to stratum \( s \)—that is, \( \Omega^* = \{ P(\vec{t}) \mid \lambda(P) = s \} \).

Rule \( r' \) is an instance of a rule \( r \) if a substitution \( \sigma \) exists mapping all variables of \( r \) to constants such that \( r' = r \sigma \).

For \( I \) a dataset, the set \( \text{inst}_r[I] \) of instances of \( r \) obtained by applying a rule \( r \) to \( I \), and the set \( \Pi[I] \) of facts obtained by applying a program \( \Pi \) to \( I \) are defined as follows.

\[
\text{inst}_r[I] = \{ r\sigma \mid b^+(r)\sigma \subseteq I \text{ and } b^-(r)\sigma \cap I = \emptyset \} \tag{1}
\]

\[
\Pi[I] = \bigcup_{r \in \Pi} \{ h(r') \mid r' \in \text{inst}_r[I] \} \tag{2}
\]

We often say that each instance in \( \text{inst}_r[I] \) fires on \( I \). We are now ready to define the semantics of stratified datalog. Given a dataset \( E \) of explicit facts and a stratification \( \lambda \) of \( \Pi \) with maximum stratum index number \( S \), we define the following sequence of datasets: let \( I^0_s = E \); let \( I^s_i = I^{s-1}_s \) for index \( s \) with \( 1 \leq s \leq S \); let \( I^S_i = I^S_{i-1} \cup \Pi^S_{i-1} \) for each integer \( i > 0 \); and let \( I^\infty_i = \bigcup_{j \geq 0} I^j_i \). Set \( I^\infty_s \) is called the materialisation of \( \Pi \) w.r.t. \( E \) and \( \lambda \). It is well known that \( I^\infty_s \) does not depend on \( \lambda \), so we usually write it as \( \text{mat}(\Pi, E) \). In this paper, we consider the problem of maintaining \( \text{mat}(\Pi, E) \): given \( \text{mat}(\Pi, E) \) and datasets \( E^- \) and \( E^+ \), our algorithm computes \( \text{mat}(\Pi, (E \setminus E^-) \cup E^+) \) incrementally while minimising the amount of work.

3 Motivation and Intuition

As motivation for our work, we next discuss how evaluating rules ‘backwards’ can be a significant source of inefficiency during materialisation maintenance. We base our discussion on the DRed algorithm for simplicity, but our conclusions apply to the B/F algorithm as well.

3.1 The DRed Algorithm

To make our discussion precise, we first present the DRed algorithm (Gupta, Mumick, and Subrahmanian 1993; Staudt and Jarke 1996). Let \( E \) be a program with a stratification \( \lambda \), let \( E^- \) be a set of explicit facts, and assume that the materialisation \( I = \text{mat}(\Pi, E) \) of \( \Pi \) w.r.t. \( E \) has been computed. Moreover, assume that \( E \) should be updated by deleting \( E^- \) and inserting \( E^+ \). The DRed algorithm efficiently modifies the ‘old’ materialisation \( I \) to the ‘new’ materialisation \( I' = \text{mat}(\Pi, (E \setminus E^-) \cup E^+) \) by deleting some facts and adding others; we call such facts affected by the update.

Due to the update, some rule instances that fire on \( I \) will no longer fire on \( I' \), and some rule instances that do not fire on \( I \) will fire on \( I' \); we also call such rule instances affected by the update. A key problem in materialisation maintenance
is to identify the affected rule instances. Clearly, the body of each affected rule instance must contain an affected fact. Based on this observation, the affected rule instances can be efficiently identified by the following generalisation of the operators \(\text{inst}_I\) and \(\Pi[I]\) from Section 2. In particular, let \(I^P, I^N, P,\) and \(N\) be datasets such that \(P \subseteq I^P\) and \(N \cap I^N = 0\); then, let

\[
\text{inst}_{\{I^P, I^N : P, N\}} = \{ r\sigma | b^+(r)\sigma \subseteq I^P \text{ and } b^-(r)\sigma \cap I^N = 0, \text{ and } b^+(r)\sigma \cap P \neq \emptyset \text{ or } b^-(r)\sigma \cap N \neq \emptyset \}
\]

and let

\[
\Pi[I^P, I^N : P, N] = \bigcup_{r \in \Pi} \{ h(r') | r' \in \text{inst}_{\{I^P, I^N : P, N\}} \}.
\]

Intuitively, the positive and the negative rule atoms are evaluated in \(I^P\) and \(I^N\); sets \(P\) and \(N\) identify the affected positive and negative facts; \(\text{inst}_{\{I^P, I^N : P, N\}}\) are the affected instances of \(r\); and \(\Pi[I^P, I^N : P, N]\) are the affected consequences of \(\Pi\). We define \(\Pi[I^P, I^N]\) and \(\Pi[I^P, I^N]\) analogously to above, but without the condition \(b^+(r)\sigma \cap P \neq \emptyset \) or \(b^-(r)\sigma \cap N \neq \emptyset \). We omit for readability whenever \(I^P = I^N\), and furthermore we omit \(N\) when \(N = \emptyset\). Sets \(\Pi[I^P, I^N]\) and \(\Pi[I^P, I^N : P, N]\) can be computed efficiently in practice by evaluating the body of each rule \(r \in \Pi\) as a conjunctive query and instantiating the head as needed.

Algorithm 1 formalises DRed. The algorithm processes each stratum \(s\) and accumulates the necessary changes to \(I\) in the set \(D\) of overdeleted and the set \(A\) of added facts. The materialisation is updated in line 6, so, prior to that, \(I\) and \((I \setminus D) \cup A\) are the ‘old’ and the ‘new’ materialisation, respectively. The computation proceeds in three phases.

In the overdeletion phase, \(D\) is extended with all facts that depend on a deleted fact. In line 8 the algorithm identifies the facts that are explicitly deleted \((E^- \cap O^+)\) or are affected by deletions in the previous strata \((\Pi[I^P : D \setminus A, A \setminus D])\), and then in lines 9–13 it computes their consequences. It uses a form of the semiautive strategy, which ensures that each rule instance is considered only once during overdeletion.

In the one-step rederivation phase, \(R\) is computed as the set of facts that have been overdeleted, but that hold nonetheless. To this end, in line 4 the algorithm considers each fact \(F\) in \(D \cap O^+\), and it adds \(F\) to \(R\) if \(F\) is explicit or it is rederived by a rule instance. The latter involves evaluating rules ‘backwards’: the algorithm identifies each rule \(r \in \Pi\) whose head can be matched to \(F\), and it evaluates over the ‘new’ materialisation the body of \(r\) as a query with the head variables bound; fact \(F\) holds if the query returns at least one answer. As we discuss shortly, this step can be a major source of inefficiency in practice, and the main contribution of this paper is eliminating ‘backward’ rule evaluation and thus significantly improving the performance.

In the insertion step, in line 15 the algorithm combines the one-step rederived facts \((R)\) with the explicitly added facts \((E^+ \cup O^+)\) and the facts added due to the changes in the previous strata \((\Pi[I^P : (I \setminus D) \cup A : A \setminus D, D \setminus A])\), and then in lines 16–20 it computes all of their consequences and adds them to \(A\). Again, the semiautive strategy ensures that each rule instance is considered only once during insertion.

### 3.2 Problems with Evaluating Rules ‘Backwards’

The one-step rederivation in line 4 of Algorithm 1 evaluates rules ‘backwards’. In this section we present two examples that demonstrate how this can be a major source of inefficiency. Both examples are derived from datasets we used in our empirical evaluation that we present in Section 6; hence, these problems actually arise in practice.

Our discussion depends on several details. In particular, we assume that all facts are indexed so that all facts matching any given atom (possibly containing constants) can be identified efficiently. Moreover, we assume that conjunctive queries corresponding to rule bodies are evaluated left-to-right: for each match of the first conjunct, we partially instantiate the rest of the body and match it recursively. Finally, we assume that query atoms are reordered prior to evaluation to obtain an efficient evaluation plan.

**Example 1.** Let \(\Pi\) and \(E\) be the program and the dataset as specified in (4) and (5), respectively.

\[
R(x, y_1) \land R(x, y_2) \rightarrow S(y_1, y_2)
\]

\[
E = \{ R(a_i, c_i), R(a_i, c_i) | 1 \leq i \leq n \}
\]

The materialisation mat(\(\Pi\), \(E\)) consists of \(E\) extended with facts \(S(b, b), S(b, c_i), S(c_i, b),\) and \(S(c_i, c_i)\) for \(1 \leq i \leq n\).

During materialisation, the body of rule (4) can be evaluated efficiently left-to-right: we match \(R(x, y)\) to either \(R(a_i, b)\) or \(R(a_i, c_i)\); this instantiates \(R(x, y)\) as \(R(a_i, y)\), and we use the index to find the matching facts \(R(a_i, b)\) and \(R(a_i, c_i)\). Thus, \(R(x, y)\) has \(2n\) matches, each of which contributes to two matches of \(R(x, y)\), so the overall cost of rule matching is \(O(n)\). The rule body is symmetric, so reordering the body atoms has no effect.

Now assume that we delete all \(R(a_i, c_i)\) with \(1 \leq i \leq n\). DRed then overdeletes all \(S(b, c_i), S(c_i, b)\) and \(S(c_i, c_i)\)

---

1873
facts in lines 8–13, and this can be done efficiently as in
the previous paragraph. Next, in one-step rederivation, the
algorithm will match these facts to the head of the rule (4)
and obtain queries \( R(x, b) \land R(x, c_i) \), \( R(x, c_i) \land R(x, b) \),
and \( R(x, c_i) \land R(x, c_j) \). All but the last of these queries con-
tain atom \( R(x, b) \) and, no matter how we reorder the body
atoms of (4), we have \( n \) queries where \( R(x, b) \) is evaluated
first. Each of these \( n \) queries identifies \( n \) candidate matches
\( R(a_i, b) \) using the index only to find out that the second atom
cannot be matched. Thus, \( R(x, b) \) is matched to \( n^2 \) facts in
total, so the cost of one-step rederivation is \( O(n^2) \)—one de-
gree higher than for materialisation.

Example 1 shows that evaluating a rule ‘backwards’ can
be inherently more difficult than evaluating it during ma-
terialisation, thus giving rise to a dominating source of ineffi-
ciency. In fact, evaluating a rule with \( m \) body atoms ‘back-
wards’ can be seen as answering a query with \( m + 1 \) atoms,
where the head of the rule is an extra query atom; since
the number of atoms determines the complexity of query eval-
uation, this extra atom increases the algorithm’s complexity.

Our next example shows that this problem is exacerbated
if the space of admissible plans for queries corresponding
to rule bodies is further restricted. This is common in sys-
tems that provide built-in functions. In particular, to facilitate
manipulation of concrete values such as strings or integers,
datalog systems often allow rule bodies to contain built-in atoms
of the form \( (t := \text{exp}) \), where \( t \) is a term and \( \text{exp} \) is
an expression constructed using constants, variables, func-
tions, and operators as usual. For example, a built-in atom
can have the form \( (z := z_1 + z_2) \), and it assigns to \( z \) the sum
of \( z_1 \) and \( z_2 \). The set of supported functions varies among im-
plementations, but a common feature is that all values in \( \text{exp} \)
must be bound by prior atoms before the built-in atom can
be evaluated. As we show next, this can be problematic.

Example 2. Let program \( \Pi \) consist of rules (6) and (7). If
we read \( B(s, t, n) \) as saying that there is an edge from node
\( s \) to node \( t \) of length \( n \), then the program entails \( D(s, n) \) if
there exists a path of length \( n \) from node \( a \) to node \( s \).

\[
B(a, y, z) \rightarrow D(y, z) \quad (6)
\]
\[
D(x, z_1) \land B(x, y, z_2) \land (z := z_1 + z_2) \rightarrow D(y, z) \quad (7)
\]
Let \( E \) be the dataset as specified below.

\[
E = \{ B(a, b_1, 1), B(a, c_i, 1), B(b_i, d_j, 1) \mid 1 \leq i, j \leq n \}
\]

During materialisation, rule (6) first derives \( D(b_1, 1) \) and
dulet \( D(c_1, 1) \) with \( 1 \leq i \leq n \), so the cost of this step is \( O(n) \).

Next, atom \( D(x, z_1) \) in rule (7) is matched to \( n \) facts \( D(c_1, 1) \)
without deriving anything. Atom \( D(x, z_1) \) is also matched
to \( D(b_1, 1) \) once, so atom \( B(x, y, z_2) \) is instantiated to
\( B(b_1, y, z_2) \) and matched to \( n \) facts \( B(b_1, d_j, 1) \), deriving
\( n \) facts \( D(d_j, 2) \). Thus, the cost of rule matching is \( O(n) \).

Now assume that \( B(a, b_1, 1) \) is deleted. Then, \( D(b_1, 1) \)
and all \( D(d_j, 2) \) can be efficiently overdeleted as in the pre-
vious paragraph, but trying to prove them is much more dif-
ficult. Matching each \( D(d_j, 2) \) to the head of (6) produces
a query \( B(a, d_j, 2) \), which does not produce a rule instance.
Moreover, matching \( D(d_j, 2) \) to the head of (7) produces a
query \( D(x, z_1) \land B(x, d_j, z_2) \land (2 := z_1 + z_2) \). Now, as we
discussed earlier, \( z_1 \) and \( z_2 \) must both be bound before we
can evaluate the built-in atom \( (2 := z_1 + z_2) \). If we evaluate
\( B(x, d_j, z_2) \) first, then we try \( n \) facts \( B(b_i, d_j, 1) \) with
\( 1 \leq i \leq n \); for each of them, atom \( D(x, z_1) \) is instantiated
as \( D(b_i, z_1) \) and is not matched in the surviving facts. In
contrast, if we evaluate \( D(x, z_1) \) first, then we try \( n \) facts
\( D(c_i, 1) \); for each of them, atom \( B(x, d_j, z_2) \) is instantiated
as \( B(c_i, y, z_2) \) and is not matched. Thus, regardless of how
we reorder the body of (7), the first atom considers a total of
\( n^2 \) facts, so the cost of one-step rederivation is \( O(n^2) \).

To overcome this, one might rewrite the built-in atom as
\( (z := z_1 + z_2) \) or \( (2 := z_1 + z_2) \) so that it can be evaluated
immediately after \( z \) and either \( z_1 \) or \( z_2 \) are bound. Either
way, one-step rederivation still takes \( O(n^2) \) steps on our ex-
ample. Also, built-in expressions are often not invertible.

4 Combining DRed with Counting

We now address the efficiencies we outlined in Section 3.
Towards this goal, in Section 4.1 we first present the intu-
itions, and then in Section 4.2 we formalise our solution.

4.1 Intuition

As we already mentioned in Section 1, the Counting algo-
rithm (Gupta, Mumick, and Subrahmanian 1993) does not
evaluate rules ‘backwards’; instead, it tracks the number of
derivations of each fact. The main drawback of Counting
is that it cannot handle recursive rules. We now illustrate
the intuition behind our DRed algorithm, which combines
DRed with Counting in a way that eliminates ‘backward’
rule evaluation, while still supporting recursive rules.

The DRed algorithm associates with each fact two coun-
ters that track the derivations via the nonrecursive and
the recursive rules separately. The counters are decremented
(resp. incremented) when the associated fact is derived in
overdeletion (resp. insertion), which allows for two impor-
tant optimisations. First, as in the Counting algorithm, the
nonrecursive counter always reflects the number of deriv-
ations from facts in earlier strata; hence, a fact with a nonzero
nonrecursive counter should never be overdeleted because
it clearly remains true after the update. This optimisation
captures the behaviour of Counting on nonrecursive rules
and it also helps limit overdeletion. Second, if we never
overdelete facts with nonzero nonrecursive counters, the
only way for a fact to still hold after overdeletion is if
its recursive counter is nonzero; hence, we can replace
‘backward’ rule evaluation by a simple check of the recursive
counter. Note, however, that the recursive counters can be
checked only after overdeletion finishes. This optimisation
extends the idea of Counting to recursive rules to completely
avoid ‘backward’ rule evaluation. The following example il-
ustrates these ideas and compares them to DRed.

Example 3. Let \( \Pi \) be the program containing rule (8).

\[
A(x) \land B(x, y) \rightarrow A(y) \quad (8)
\]
Moreover, let \( E \) be defined as follows:

\[
E = \{ A(a), A(b), A(d), B(a, c), B(b, c), B(c, d), B(d, e) \}
\]
The materialisation mat(Π, E) extends E with A(c) and A(e). Figure 1 shows the dependencies between derivations using arrows. For clarity, we do not show the B-facts.

Now assume that A(a) is deleted. The standard DRed algorithm first overdeletes A(a), A(c), A(d), and A(e); it rederives A(d) since the fact is in E \ E^−; it rederives A(c) by evaluating rule (8) ‘backwards’; and it derives A(d) and A(e) from the rederived facts.

Next consider applying the DRed^c to the same update. For each fact, Figure 1 shows a pair consisting of the nonrecursive and the recursive counter before the update (row T1), after overdeletion (row T2), and after the update (row T3). Note that the presence of a fact in E is akin to a nonrecursive derivation, so facts A(a), A(b), and A(d) have nonrecursive derivation counts of one before the update. Now A(c) is derived from A(a) and A(b) using the recursive rule (8), so the recursive counter for A(c) is two. Analogously, A(d) and A(e) have just one recursive derivation each. During overdeletion, A(a) is first removed from E, so the nonrecursive counter of A(a) is decremented to zero and the fact is deleted. Since A(a) derives A(c) via rule (8), the recursive counter of A(c) is decremented; since the nonrecursive counter of A(c) is zero, the fact is overdeleted. Since A(c) derives A(d) via rule (8), the recursive counter of A(d) is decremented. Now the nonrecursive counter of A(d) is nonzero, so we know that A(d) holds after the update; hence, the fact is not overdeleted, and the overdeletion phase stops. Thus, while DRed overdeletes four facts, DRed^c overdeletes only A(a) and A(c), and does not ‘touch’ A(e).

Next, DRed^c proceeds to one-step rederivation. The recursive counter of A(c) is nonzero, which means that the fact has a recursive derivation (from A(b) in this case) that is not affected. Thus, DRed^c rederives A(c) without any ‘backward’ rule evaluation.

Finally, DRed^c applies insertion. Since A(c) derives A(d) via (8), the recursive counter of A(d) is incremented. Fact A(d), however, was not overdeleted, so insertion stops.

By avoiding ‘backward’ rule evaluation, DRed^c removes the dominating source of inefficiency on Examples 1 and 2. In fact, on the nonrecursive program from Example 1, the recursive counter is never used and DRed^c performs the same inferences as the Counting algorithm.

### 4.2 Formalisation

We now formalise our DRed^c algorithm. Our definitions use the standard notion of multisets—a generalisation of sets where each element is associated with a positive integer called the multiplicity specifying the number of the element’s occurrences in the multiset. Moreover, ⊕ is the multiset union operator, which adds the elements’ multiplicities.

If an operand of ⊕ is a set, it is treated as a multiset where all elements have multiplicity one. Finally, we extend the notion of rule matching to correctly reflect the number of times a fact is derived: for I^p, I^p, P, and N datasets with P ⊆ I^p and N ∩ I^p = ∅, we define Π[I^p, I^p : P, N] as the multiset containing a distinct occurrence of h[r^x] for each rule r ∈ Π and its instance r^x ∈ inst[r^x],[I^p, I^p : P, N]. This multiset can be computed analogously to Π[I^p, I^p : P, N].

Just like DRed, the DRed^c takes as input a program Π, a stratification λ, a set of explicit facts E and its materialisation I = mat(Π, E), and the sets of facts E^− and E^+ to remove from and add to E. Additionally, the algorithm also takes as input maps Cnr and Cc that associate each fact F with its nonrecursive and recursive counters Cnr[F] and Cc[F], respectively. These maps should correctly reflect the relevant numbers of derivations. Formally, Cnr and Cc must be compatible with Π, λ, and E, which is the case if Cnr[F] = Cc[F] = 0 for each fact F ∉ I, and for each fact F ∈ I and s the stratum index such that F ∈ O^s (i.e., s is the index of the stratum that F belongs to),

- Cnr[F] is the multiplicity of F in E ⊕ Π[^n^r[I]], and
- Cc[F] is the multiplicity of F in Π[^c[I]].

For simplicity, we assume that Cnr and Cc are defined on all facts, and that Cnr[F] = Cc[F] = 0 holds for F ∉ I; thus, we can simply increment the counters for each newly derived fact in procedure INSERT. In practice, however, one can maintain counters only for the derived facts and initialise the counters to zero for the freshly derived facts.

DRed^c is formalised in Algorithm 2. Its structure is similar to DRed, with the following main differences: instead of evaluating rules ‘backwards’, one-step rederivation simply checks the recursive counters (line 24); a fact is overdeleted only if the nonrecursive derivation counter is zero (line 34); and the derivation counters are decremented in overdeletion (lines 29–32 and 36–37) and incremented in insertion (lines 41–44 and 49–50). The algorithm also accumulates changes to the materialisation in sets D and A by iteratively processing the strata of λ in three phases.

In the overdeletion phase, DRed^c first considers explicitly deleted facts or facts affected by the changes in earlier strata (lines 29–32). This is analogous to line 8 of DRed, but DRed^c must distinguish Π^c from Π^r so it can decrement the appropriate counters. Next, DRed^c identifies the set ΔD of facts that have not yet been deleted and whose nonrecursive counter is zero (line 34): a fact with a nonzero nonrecursive counter will always be part of the ‘new’ materialisation. Note that recursive derivations can be cyclic, so we cannot use the recursive counter to further constrain overdeletion at this point. Then, in lines 35–38 the algorithm propagates consequences of ΔD just like Algorithm 1, with additionally decrementing the recursive counters in line 37.
Algorithm 2 $\text{DRED}^c(\Pi, \lambda, E, I, E^-, E^+, C_m, C_t)$

1. $D := A = \emptyset, \quad E' := (E^- \cap E) \setminus E^+, \quad E^+ := E^+ \setminus E$
2. for each stratum index $s$ with $1 \leq s \leq S$ do
3. 24. $R := \{ F \in D \cap O^+ | C_t[F] > 0 \}$
4. 25. INSERT
5. 26. $E := (E \setminus E^-) \cup E^+$, $I := (I \setminus D) \cup A$
6. 27. procedure $\text{OVERDELETE}$
7. 28. $N_D := \emptyset$
8. 29. for $F \in (E^- \cap O^-) \cap \Pi_m$ do
9. 30. $N_D := N_D \cup \{F\}, \quad C_m[F] := C_m[F] - 1$
10. 31. for $F \in \Pi_r[I \setminus D \setminus A, A \setminus D]$ do
11. 32. $N_D := N_D \cup \{F\}, \quad C_t[F] := C_t[F] - 1$
12. 33. loop
13. 34. $\Delta_D := \{ F \in N_D \setminus D | C_m[F] = 0 \}$
14. 35. if $\Delta_D = \emptyset$ then break
15. 36. for $F \in \Pi_r[I \setminus (D \setminus A), I \cup A : \Delta_D]$ do
16. 37. $N_D := N_D \cup \{F\}, \quad C_t[F] := C_t[F] + 1$
17. 38. $D := D \cup \Delta_D$
18. 39. procedure $\text{INSERT}$
19. 40. $N_A := R$
20. 41. for $F \in (E^+ \cap O^-) \cap \Pi_m[I \setminus D \setminus A : A \setminus D]$ do
21. 42. $N_A := N_A \cup \{F\}, \quad C_m[F] := C_m[F] + 1$
22. 43. for $F \in \Pi_r[I \setminus (D \setminus A), (I \setminus D) \cup A : \Delta_A]$ do
23. 44. $N_A := N_A \cup \{F\}, \quad C_t[F] := C_t[F] + 1$
24. 45. loop
25. 46. $\Delta_A := N_A \setminus ((I \setminus D) \cup A)$
26. 47. if $\Delta_A = \emptyset$ then break
27. 48. $A := A \cup \Delta_A$
28. 49. for $F \in \Pi_r[I \setminus (D \setminus A), I \setminus D : \Delta_A]$ do
29. 50. $N_A := N_A \cup \{F\}, \quad C_t[F] := C_t[F] + 1$

In the one-step rederivation phase, instead of evaluating rules ‘backwards’, DRED$^c$ just checks the recursive counter of each fact $F \in D \cap O^-$ (line 24); if $C_r[F] \neq 0$, then some derivations of $F$ were not ‘touched’ by overdeletion so $F$ holds in the ‘new’ materialisation. Conversely, if $C_r[F] = 0$, then $F \in D$ guarantees that $C_m[F] = 0$ holds as well, so $F$ is not one-step rederivable by a rule in $\Pi$.

The insertion phase of DRED$^c$ just uses the semi-naïve evaluation while incrementing the counters appropriately.

Without recursive rules, DRED$^c$ becomes equivalent to Counting, and it is optimal in the sense that only affected rule instances are considered during the update. Moreover, the computational complexities of both DRED$^c$ and DRED are the same as for the semi-naïve materialisation algorithm: ExpTime in combined and PTime in data complexity (Dantsin et al. 2001). Finally, DRED$^c$ never performs more inferences than DRED and is thus more efficient. Theorem 1 shows that our algorithm is correct, and its proof is given in an extended technical report (Hu, Motik, and Horrocks 2017).

**Theorem 1.** Algorithm 2 correctly updates $I = \text{mat}(\Pi, E)$ to $I' = \text{mat}(\Pi, E')$ for $E' = (E - E^-) \cup E^+$, and it updates $C_m$ and $C_t$ so they are compatible with $\Pi, \lambda$, and $E'$.

### 5 Combining B/F with Counting

The B/F algorithm by Motik et al. (2015) uses a combination of backward and forward chaining that makes the deletion phase exact. More specifically, when a fact $F \in \Delta_B$ is considered during deletion, the algorithm uses a combination of backward and forward chaining to look for alternative derivations of $F$, and it deletes $F$ only if no such derivation can be found. Backward chaining allows B/F to be much more efficient than DRed on many datasets, and this is particularly the case if a program contains many recursive rules. Thus, we cannot hope to remove all ‘backward’ rule evaluation without eliminating the algorithm’s main advantage.

Still, there is room for improvement: backward chaining involves ‘backward’ evaluation of both nonrecursive and recursive rules, and we can use nonrecursive counters to eliminate the former. Algorithm 3 formalises B/F$^c$—our combination of the B/F algorithm by Motik et al. (2015) with Counting. The main difference to the original B/F algorithm is that B/F$^c$ associates with each fact a nonrecursive counter that is maintained in lines 59–60 and 89–90, and, instead of evaluating nonrecursive rules ‘backwards’ to explore alternative derivations of a fact, it just checks in line 79 whether the nonrecursive counter is nonzero. We know that a fact holds if its nonrecursive counter is nonzero; otherwise, we apply backward chaining to recursive rules only. We next describe the algorithm’s steps in more detail.

Procedure $\text{DELETE:UNPROVED}$ plays an analogous role to the overdeletion step of DRed and DRED$^c$. The procedure maintains the nonrecursive counter for each fact in the same way as DRED$^c$, and the main difference is that a fact $F$ is deleted (i.e., added to $\Delta_B$) in line 66 only if no alternative derivation can be found using a combination of backward and forward chaining implemented in functions $\text{CHECK}$ and $\text{SATURATE}$. If an alternative derivation is found, $F$ is added to the set $P$ of proved facts.

A call to $\text{CHECK}(F)$ searches for the alternative derivations of $F$ using backward chaining. The function maintains the set $C$ of checked facts, which ensures that each $F$ is checked only once (line 71 and 78). The procedure first calls $\text{SATURATE}(F)$ to determine whether $F$ follows from the facts considered thus far; we discuss this step in more detail shortly. If $F$ is not proved, the procedure then examines in lines 73–76 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. This involves evaluating rules ‘backwards’ and, as we already discussed in Section 5, this is the main advantage of the B/F algorithm over DRed on a number of complex inputs. The function terminates once $F$ is successfully proved (line 76).

Set $P$ accumulates facts that are checked and successfully proved, and it is computed in function $\text{SATURATE}$ using forward chaining. Given a fact $F$ that is being checked, it first verifies whether $F$ has a nonrecursive derivation. In the original B/F algorithm, this is done by evaluating the nonrecursive rules ‘backwards’ in the same way as in line 4 of DRed. In contrast, B/F$^c$ avoids this by simply checking whether the nonrecursive counter is nonzero (line 79); if that is the case, then $F$ is known to have nonrecursive derivations and it is added to $P$ via lines 80 and 82. If $F$ is proved, the proce-
dure propagates its consequences (line 80–85). In particular, the procedure ensures that each consequence \( F' \) of \( P \), the facts in the ‘new’ materialisation in the previous strata, and the recursive rules is added to \( P \) if \( F' \in C \), or is added to the set \( Y \) of delayed facts if \( F' \notin C \). Intuitively, set \( Y \) contains facts that are proved but that have not been checked yet. If a fact in \( Y \) is checked at a later point, it is proved in line 79 without having to apply the rules again.

Since the deletion step of B/F\(^c\) is ‘exact’ in the sense that it deletes precisely those facts that no longer hold after the update, rederivation is not needed. Thus, DELETE UNPROVED is directly followed by INSERT, which is the same as in DRed and DRed\(^c\), with the only difference that B/F\(^c\) maintains only the nonrecursive counters.

Algorithm 3 is correct in the same way as B/F since checking whether a fact has a nonzero nonrecursive counter is equivalent to checking whether a derivation of the fact exists by evaluating nonrecursive rules ‘backwards’.

### 6 Evaluation

We have implemented the unoptimised and optimised variants of DRed and B/F and have compared them empirically.

**Benchmarks** We used the following benchmarks for our evaluation: UOBM (Ma et al. 2006) is a synthetic benchmark that extends the well known LUBM (Guo, Pan, and Heflin 2005) benchmark; Reactome (Croft et al. 2013) models biological pathways of molecules in cells; Uniprot (Bateman et al. 2015) describes protein sequences and their functional information; ChemBL (Gaulton et al. 2011) represents functional and chemical properties of bioactive compounds; and Claros describes archeological artefacts. Each benchmark consists of a set of facts and an OWL 2 DL ontology, which we transformed into datalog programs of different levels of complexity and recursiveness. More specifically, the upper bound (U) programs were obtained using the complete but unsound transformation by Zhou et al. (2013), and they entail all consequences of the original ontology but may also derive additional facts. The recursive (R) programs were obtained using the sound but incomplete transformation by Kaminski, Nenov, and Grau (2016), and they tend to be highly recursive. For Claros, the lower bound extended (LE) program was obtained by manually introducing several ‘hard’ rules, and it was already used by Motik et al. (2015) to compare DRed with B/F. Finally, to estimate the effect of built-in literals on materialisation maintenance, we developed a new synthetic benchmark SSPE (Single-Source Path Enumeration). Its dataset consists of a randomly generated directed acyclic graph of 100 k nodes and 1 M edges, and its program traverses paths from a single source analogously to rules (6)–(7). All the tested programs are recursive, although the percentage of the recursive rules varies. Table 1 shows the numbers of facts (\(|E|\)), strata (\(S\)), the nonrecursive rules (\(|\Pi^r|\)), and the recursive ones (\(|\Pi^r|\)) for each benchmark.

**Test Setup** We conducted all experiments on a Dell PowerEdge R720 server with 256GB RAM and two Intel Xeon E5-2670 2.6GHz processors, running Fedora 24, kernel version 4.8.12-200.fc24.x86_64. All algorithms handle insertions using the seminaive evaluation. The only overhead is in counter maintenance, which we measured during initial materialisation (which also uses seminaive evaluation). Hence, the main focus of our tests was on comparing the performance of our algorithms on ‘small’ and ‘large’ deletions. In both cases, we first materialised the relevant program on the explicit facts, and then we performed the following tests.

To test small deletions, we measured the performance on
Table 1: Average running times for deleting 1000 facts (seconds)

| Dataset  | |E| | S | |Π| | DRedc | DRed | B/Fc | B/F |
|----------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| UOBM-U   | 254.8 M         | 5               | 135             | 144             | 179.24          | 1,185.87        | 1.18            | 31.96           |                 |
| UOBM-R   | 254.8 M         | 6               | 164             | 2,215           | 0.29            | 0.56            | 0.26            | 0.24            |                 |
| Reactome-U| 12.5 M         | 9               | 814             | 28              | 0.06            | 1.03            | 0.05            | 1.00            |                 |
| Reactome-R| 12.5 M         | 1               | 0               | 21,385          | 26.57           | 62.46           | 0.89            | 0.90            |                 |
| Uniprot-R| 123.1 M         | 5               | 9,312           | 2,706           | 4.31            | 8.71            | 4.13            | 4.36            |                 |
| ChemBL-R | 289.2 M         | 3               | 1,766           | 499             | 7.91            | 12.22           | 1.27            | 1.26            |                 |

Table 2: Running times for handling large deletions (seconds)

| Dataset  | |E| | S | |Π| | DRedc | DRed | B/Fc | B/F |
|----------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| UOBM-U   | 50%             | 1.34 k          | 3.66 k          | 1.64 k          | 3.11 k          | 1.36 k          | 1.60 k          | 1.61 k          |                 |
| UOBM-R   | 36%             | 3.28 k          | 5.75 k          | 2.76 k          | 2.87 k          | 4.14 k          | 4.16 k          | 4.19 k          |                 |
| Reactome-U| 68%             | 30.70           | 6.32 k          | 39.16           | 6.33 k          | 30.90           | 31.23           | 31.32           |                 |
| Reactome-R| 31%             | 1.07 k          | 1.78 k          | 0.92 k          | 0.93 k          | 0.91 k          | 0.91 k          | 0.92 k          |                 |
| Uniprot-R| 47%             | 1.57 k          | 3.47 k          | 1.92 k          | 2.86 k          | 1.98 k          | 1.99 k          | 2.00 k          |                 |
| ChemBL-R | 69%             | 4.74 k          | 7.22 k          | 3.25 k          | 4.18 k          | 2.56 k          | 2.57 k          | 2.59 k          |                 |
| Claros-LE| 8%              | 5.01 k          | 17.81 k         | 3.49 k          | 16.74 k         | 3.36 k          | 3.49 k          | 3.60 k          |                 |
| SSPE     | 2%              | 74.36           | 7.24 k          | 7.90 k          | 7.75 k          | 68.28           | 70.18           | 71.81           |                 |

ten randomly selected subsets $E^{-} \subseteq E$ of 1,000 facts. In all apart from Claros-LE, the running times did not depend significantly on the selected subset of $E$, so in Table 1 we report the average times across all ten runs. On Claros-LE, however, the running times varied significantly, so we report in the table the best, the worst, and the average times.

To test large deletions, we identified the largest subset $E^{-} \subseteq E$ on which either DRedc or B/Fc takes roughly the same time as computing the ‘new’ materialisation from scratch. We measured the performance of all algorithms on $E^{-}$, as well as the performance of rematerialisation with no counters (Remat), just the nonrecursive counter (Remat-1C), and both counters (Remat-2C). This test allows us to assess the scalability of our algorithms. Table 2 reports the running times and the percentages of the deleted facts.

**Discussion**

DRedc outperformed DRed on all inputs for small deletions. In particular, on SSPE, the average running time for DRed drops from 28 minutes to just over 10 seconds. On Reactome-U, the improvement is by several orders of magnitude, albeit unoptimised DRed is already quite efficient. The improvement is also significant in many other cases, including UOBM-U, Reactome-R, and ChemBL-R. In fact, Reactome-U and SSPE exhibit data and rule patterns outlined in Examples 1 and 2, which clearly demonstrates the benefits of eliminating ‘backward’ rule evaluation. Moreover, the program of Claros-LE contains a symmetric and transitive predicate $relatedPlaces$, so the materialisation contains several large cliques of constants connected by this predicate (Motik et al. 2015). When a fact $relatedPlaces(a, b)$ is overdeleted, the DRed algorithm overdeletes all $relatedPlaces(c, d)$ where $c$ and $d$ belong to the same clique as $a$ and $b$, which requires a cubic number of derivations. However, DRedc can sometimes (but not always) prove that $relatedPlaces(a, b)$ holds using the nonrecursive counter; as one can see, this can considerably improve the performance by avoiding costly overdeletion.

B/Fc also outperformed B/F for small deletions in many cases: B/Fc was more than 20 times faster for UOBM-U, Reactome-U, and the ‘best’ case of Claros-LE, which is in line with our observation that ‘backwards’ rule evaluation can be quite costly. In contrast, on the highly recursive datasets (i.e., all R-datasets and SSPE), the performance of B/Fc and B/F is roughly the same: the main source of difficulty is due to the recursive rules, whose evaluation is unaffected by the optimisations proposed in this paper.

B/Fc outperformed DRedc on all datasets but SSPE. This is so because B/Fc eagerly identifies alternative derivations of facts, which is often easy, and is beneficial since it can considerably reduce overdeletion. However, as we discussed earlier in Section 3, backward rule evaluation can be a dominant source of inefficiency (e.g., on SSPE). In such cases, DRedc is faster than B/Fc since DRedc completely eliminates backward rule evaluation, whereas B/Fc only avoids backward evaluation on nonrecursive rules.

The tests for large deletions show that our algorithms can efficiently delete large subsets of the explicit facts on all but two benchmarks: Claros-LE and SSPE. Claros-LE is difficult due to the presence of cliques as explained earlier, and SSPE is difficult because deleting a small percentage of the explicit facts leads to the deletion of about half of the inferred facts. Nevertheless, DRedc always considerably outperforms DRed; the difference is particularly significant on Reactome-U and SSPE, where DRedc is several orders of magnitude faster than DRed.
magnitude faster. Similarly, B/F consistently outperforms B/F on all cases apart from SSPE, where the latter is due to the overhead of maintaining the counters.

Finally, the rematerialisation times show that counter maintenance incurs only modest overheads: Remat-2C was in the worst case only several percent slower than Remat.

7 Conclusion

We have presented two novel algorithms for the maintenance of datalog materialisations, obtained by combining the well-known DRed and B/F algorithms with Counting. Our evaluation shows that our algorithms are generally more efficient than the original ones, often by orders of magnitude. Our algorithms can handle both small and large updates efficiently, and have thus been shown to be ready for practical use. In future, we shall develop a modular approach to materialisation and its maintenance that tackles the difficult cases such as Claros-LE using reasoning modules that can be ‘plugged into’ the seminaive evaluation to handle difficult rule combinations using custom algorithms.

Acknowledgments

This work was supported by the EPSRC projects MaSI, DBOnto, and ED.

References


