Combining Rewriting and Incremental Materialisation Maintenance for Datalog Programs with Equality

Boris Motik, Yavor Nenov, Robert Piro and Ian Horrocks
Department of Computer Science, Oxford University
Oxford, United Kingdom
firstname.lastname@cs.ox.ac.uk

Abstract

Materialisation precomputes all consequences of a set of facts and a datalog program so that queries can be evaluated directly (i.e., independently from the program). Rewriting optimises materialisation for datalog programs with equality by replacing all equal constants with a single representative; and incremental maintenance algorithms can efficiently update a materialisation for small changes in the input facts. Both techniques are critical to practical applicability of datalog systems; however, we are unaware of an approach that combines rewriting and incremental maintenance. In this paper we present the first such combination, and we show empirically that it can speed up updates by several orders of magnitude compared to using either rewriting or incremental maintenance in isolation.

1 Introduction

Datalog [Abiteboul et al., 1995] is a declarative, rule-based language that can describe (possibly recursive) data dependencies. It is widely used in applications as diverse as enterprise data management [Aref, 2010] and query answering over ontologies in the OWL 2 RL profile [Motik et al., 2009] extended with SWRL rules [Horrocks et al., 2004].

Querying the set $\Pi^\approx(E)$ of consequences of a set of explicit facts $E$ and a datalog program $\Pi$ is a key service in datalog systems. It can be supported by precomputing and storing $\Pi^\approx(E)$ so that queries can be evaluated directly, without further reference to $\Pi$. Set $\Pi^\approx(E)$ and the process of computing it are called the materialisation of $E$ w.r.t. $\Pi$. This technique is used in the state of the art systems such as Owlgres [Stockier and Smith, 2008], WebPIE [Urbani et al., 2012], Oracle’s RDF store [Wu et al., 2008], GraphDB (formerly OWLIM) [Bishop et al., 2011], and RDFox [Motik et al., 2014].

Although datalog traditionally employs the unique name assumption (UNA), in some applications uniqueness of identifiers cannot be guaranteed. For example, due to the distribution and the independence of data sources, in the Semantic Web different identifiers are often used to refer to the same domain object. Handling such use cases requires an extension of datalog without UNA, in which one can infer equalities between constants using a special equality predicate $\approx$ that can occur in facts and rule heads. The semantics of $\approx$ can be captured explicitly using rules that axiomatise $\approx$ as a congruence relation; however, this is known to be inefficient when equality is used extensively. Therefore, systems commonly use rewriting [Baader and Nipkow, 1998; Nieuwenhuis and Rubio, 2001]—an optimisation where equal constants are replaced with a canonical representative, and only facts containing such representatives are stored. The benefits of rewriting have been well-documented in practice [Wu et al., 2008; Urbani et al., 2012; Bishop et al., 2011; Motik et al., 2015b].

Moreover, datalog applications often need to handle continuous updates to the set of explicit facts $E$. Rematerialisation (i.e., computing the materialisation from scratch) is often very costly, so incremental maintenance algorithms are often used in practice. Adding facts to $E$ is trivial as one can simply continue from where the initial materialisation has finished; hence, given a materialisation $\Pi^\approx(E)$ of $E$ w.r.t. $\Pi$ and a set of facts $E^-$, the main challenge for an incremental algorithm is to efficiently compute $\Pi^\approx(E \setminus E^-)$. Several such algorithms have already been proposed. Truth maintenance systems [Doyle, 1979; de Kleer, 1986; Goasdoué et al., 2013] track dependencies between facts to efficiently determine whether a fact has a derivation from $E \setminus E^-$, so only facts for which no such derivations exist are deleted. Such approaches, however, store large amounts of auxiliary information and are thus often unsuitable for data-intensive applications. Counting [Nicolas and Yazdani, 1983; Gupta et al., 1993; Urbani et al., 2013; Goasdoué et al., 2013] stores with each fact $F \in \Pi^\approx(E)$ the number of times $F$ has been derived during initial materialisation, and this number is used to determine when to delete $F$; however, in its basic form counting works only with nonrecursive rules, and a proposed extension to recursive rules requires multiple counts per fact [Dewar et al., 1992], which can be costly. The Delete/Rederive (DRed) algorithm [Gupta et al., 1993] handles recursive rules with no storage overhead: to delete $E^-$ from $E$, the algorithm first overdeletes all consequences of $E^-$ in $\Pi^\approx(E)$ and then redervises all facts provable from $E \setminus E^-$. The Backward/Forward (B/F) algorithm combines backward and forward chaining in a way that outperforms DRed on inputs where facts have many alternative derivations—a common scenario in Semantic Web applications [Motik et al., 2015c].

Combining rewriting and incremental maintenance is difficult due to complex interactions between the two tech-
niques: removing $E^a$ from $E$ may entail retracting equali-
ties, which may (partially) invalidate the rewriting and require
the restoration of rewritten facts (see Section 3). To the best
of our knowledge, such a combination has not been consid-
ered in the literature, and practical systems either use rewriting
with rematerialisation, or axiomatis e equality and use incre-
mental maintenance; in either case they give up a tech-
nique known to be critical for performance. In this paper we
present the B/F algorithm, which combines rewriting with
B/F: given a set of facts $E^a$, our algorithm efficiently updates
the materialisation of $E$ w.r.t. II computed using the rewriting
approach by Motik et al. (2015b). Extensions of datalog with
equality are nowadays used mainly for querying RDF data
extended with OWL 2 RL ontologies and SWRL rules, so we
formalise our algorithm in the framework of RDF; however,
our approach can easily be adapted to general datalog.

We have implemented B/F in the open-source RDFox
system1 and have evaluated it on several real-world and syn-
thetic datasets. Our results show that the algorithm indeed
combines the best of both worlds, as it is often several orders
of magnitude faster than either rematerialisation with rewith-
ing, or B/F with axiomatised equality.

2 Preliminaries

Datalog. A term is a constant (a, b, A, R, etc.) or a variable
($x$, $y$, z, etc.). An (RDF) atom has the form $(t_1, t_2, t_3)$, where
$t_1, t_2, t_3$ are terms; an (RDF) fact (also called a triple) is a
variable-free RDF atom; and a dataset is a finite set of facts.
A (datalog) rule $r$ is an implication of the form (1), where $H, B_1, \ldots, B_n$ are atoms and each variable occurring in $H$
also occurs in some $B_i$; $h(r) := H$ is the head atom of $r$; each $B_i$ is a body atom of $r$; and $b(r)$ is the set of all body atoms of $r$. A (datalog) program is a finite set of rules.

$$H \leftarrow B_1 \land \cdots \land B_n$$

A substitution is a partial mapping of variables to terms. For a
term, atom, rule, or a set of these, Voc($a$) is the set of all
constants in $a$, and $\alpha$ is the result of applying a substitution
$\sigma$ to $a$. The materialisation $\Pi^\infty(E)$ of a dataset $E$ w.r.t.
a program II is the smallest superset of $E$ containing $h(r)\sigma$ for
each rule $r \in II$ and substitution $\sigma$ with $b(r)\sigma \subseteq \Pi^\infty(E)$.

Equality. The constant owl:sameAs (abbreviated $\approx$) can
be used to encode equality between constants. For example,
fact (P.Smith, $\approx$, Peter_Smith) states that P.Smith
and Peter_Smith are one and the same object. Facts of the form
$\langle s, \approx, t \rangle$ are called equalities and, for readability, are
abbreviated as $s \approx t$; note that $\approx \subseteq \text{Voc}(s \approx t)$. Program II consists of rules ($\approx_1, \ldots, \approx_4$) axiomatics $\approx$ as a congruence relation.
If a program II or a dataset $E$ contain $\approx$, systems then
answer queries in the materialisation of $E$ w.r.t. II $\cup \Pi^\infty$.

Rewriting is a well-known optimisation of this approach. For
$\pi$ a mapping of constants to constants and $\alpha$ a constant, fact,
rule, dataset, or substitution, $\pi(\alpha)$ is the result of replacing
each constant $c$ in $\alpha$ with $\pi(c)$; such $\alpha$ is normal w.r.t. $\pi$
if $\pi(\alpha) = \alpha$; and $\pi(\alpha)$ is the representative of $\alpha$ in $\pi$. For
a constant $c$, let $c^\pi := \{d | \pi(d) = c\}$. For a dataset, let $U^\pi := \{(s, p, o) | \langle \pi(s), \pi(p), \pi(o) \rangle \in U\}$; and, for a fact, let $F^\pi := \{F\}^\pi$. We assume that all constant are totally
ordered such that $\approx$ is the smallest constant; then, for $S$
nonempty set of constants, min $S$ (resp. max $S$) is the smallest
(resp. greatest) element of $S$. Let $U$ be a dataset and let
$E_c(U) := \{c\} \cup \{d | c \approx d \in U\}$; then, the rewriting of $U$ is
the pair $(\pi, I)$ such that

1. $\pi(c) = \min E_c(U)$ for each constant $c$, and
2. $I = \pi(U)$.

Note that $\pi(\approx) = \approx$, that the rewriting is unique for $U$, and
that $\Pi^\infty(U) = U$ implies $I^\pi = U$. The r-materialisation of
a dataset $E$ w.r.t. a program II is the rewriting $(\pi, I)$ of the
dataset $J = (\Pi \cup \Pi^\infty)^\approx(E)$.

3 Updating R-Materialisation Incrementally

Let $E$ and $E^a$ be datasets, let $E' = E \setminus E^a$, and let II be a
program. Moreover, let $J (resp. J')$ be the materialisation of
$E$ (resp. $E'$) w.r.t. II $\cup \Pi^\infty$, and let $(\pi, I) (resp. (\pi', I'))$ be the
r-materialisation of $E$ (resp. $E'$) w.r.t. II. Given $(\pi, I)$
and $(\pi', I')$, the B/F algorithm computes $(\pi', I')$ efficiently
by combining the B/F algorithm by Motik et al. (2015c) for
incremental maintenance in datalog without equality with the
r-materialisation algorithm by Motik et al. (2015b).

We discuss the intuition in Section 3.1 and some optimisations in
Section 3.2, and we formalise the algorithm in Section 3.3.

3.1 Intuition

Main Difficulty. An update may lead to the deletion of equal-
ities, which may require adding facts to $I$. The following
example program II and dataset $E$ exhibit such behaviour.

$$II = \{y_1 \approx y_2 \leftarrow \langle y_1, R, x \rangle \land \langle y_2, R, x \rangle,$$

$$y_1 \approx y_2 \leftarrow \langle x, R, y_1 \rangle \land \langle x, R, y_2 \rangle \}$$

$$E = \{\{a, R, b\}, \{c, R, d\}, \{a, R, d\}\}$$

$$I = \{\{a, R, b\}, a \approx a, R \approx R, b \approx b, \approx \approx \approx\}$$

$$\pi = \{a \mapsto a, b \mapsto b, c \mapsto a, d \mapsto b, R \mapsto R, \approx \mapsto \approx\}$$

$$E^- = \{\{a, R, d\}\}$$

$$I' = \{\{a, R, b\}, a \approx a, R \approx R, b \approx b, \approx \approx \approx, \{c, R, d\}, c \approx c, d \approx d\}$$

$$\pi' = \{a \mapsto a, b \mapsto b, c \mapsto c, d \mapsto d, R \mapsto R, \approx \mapsto \approx\}$$

Relation $R$ is bijective in II, so $a \approx c \in J$ as both $a$ and $c$
have outgoing $R$-edges to $d$, and $b \approx d \in J$ as both $b$ and $d$
have incoming $R$-edges from $a$. By rewriting, we represent
each fact $\langle a, R, b \rangle$ from $J$ using a single fact $\langle a, R, b \rangle$, and
analogously for facts involving $\approx$; thus, instead of 14 facts,
we store just five facts. Assume now that we remove $E^a$ from $E$.
In $J$ and $J'$ we ascribe no particular meaning to $\approx$, so the monotonicity of datalog ensures $J \subseteq J'$; thus, the
B/F algorithm just needs to delete facts that no longer hold.
However, \( a \approx c \notin J' \) and \( b \approx d \notin J' \), so we must update \( \pi \) and extend \( I \) with the facts from \( J' \) that are not represented via \( \pi' \). Thus, in our example, \( J' \) actually contains \( I \).

**Solution Overview.** \( \text{B/F}^\infty \) consists of Algorithms 1–7 that follow the same basic idea as \( \text{B/F} \); to highlight the differences, lines that exist in \( \text{B/F} \) in a modified form are marked with ‘\(^*\), and new lines and algorithms are marked with ‘\( \ast \)’.

We initially mark all facts in \( \pi(E^-) \) as ‘doubtful’—that is, we indicate that their truth might change. Next, for each ‘doubtful’ fact \( F \), we determine whether \( F \) is provable from \( E' \) and, if not, we identify the immediate consequences of \( F \) (i.e., the facts in \( J \) that can be derived using \( F \)) and mark them as ‘doubtful’; we know exactly which facts have changed after processing all ‘doubtful’ facts. To check the provability of \( F \), we use backward chaining to identify the facts in \( I \) that can prove \( F \), and we use forward chaining to actually prove \( F \). The latter process also identifies the necessary changes to \( \pi \) and \( I \), which we apply to \( (\pi, I) \) in a final step. We next describe the components of \( \text{B/F}^\infty \) in more detail.

**Procedure** saturate() is given a dataset \( C \subseteq I \) of checked facts, and it computes the set \( L \) containing each fact \( F \) derivable from \( E' \) such that each fact in a derivation of \( F \) is contained in \( C^\infty \); thus, \( C \) identifies the part of \( J' \) to recompute. Rather than storing \( L \) directly, we adapt the \( r \)-materialisation algorithm by Motik et al. (2015b) and represent \( L \) by its rewriting \((\gamma, P \setminus \hat{P})\); the role of the two sets \( P \) and \( \hat{P} \) is discussed shortly. Lines 36–40 compute the facts in \( L \) derivable immediately from \( E' \): we iterate over each \( F \in C \) and each \( G \in \text{F}^\infty \); since we represent \( L \) by its rewriting, we add \( \gamma(G) \) to \( P \). The roles of set \( Y \) and lines 37–39 will be discussed shortly. Lines 41–50 compute the facts in \( L \) derivable using rules: we consider each fact \( F \in P \setminus \hat{P} \) (lines 41–42), each rule \( r \), and each match \( \sigma \) of \( F \) to a body atom of \( r \) (line 48), we evaluate the remaining body atoms of \( r \) (line 49), and we derive \( \gamma(h(r)\tau) \) for each match \( \tau \) (line 50). This basic idea is slightly more complicated by rewriting: if \( F = a \approx b \), we modify \( \gamma \) so that one constant becomes the representative of the other one (line 45). As a consequence, facts can become ‘outdated’ w.r.t. \( \gamma \), so we keep track of such facts using \( \hat{P} \): if \( F \) is ‘outdated’, we add \( F \) to \( \hat{P} \) and \( \gamma(F) \) to \( P \) (line 44); due to the latter, \( P \setminus \hat{P} \) eventually contains all ‘up to date’ facts. Finally, we apply the reflexivity rules \((\approx_i)\) to \( F \) (line 47).

Procedure saturate() is repeatedly called in \( \text{B/F}^\infty \). Set \( C \), however, never shrinks between successive calls, so set \( L \) never shrinks either; hence, at each call we can just continue the computation instead of starting ‘from scratch’. A minor problem arises if we derive a fact \( F \) with \( F \notin C^\infty \) and so we do not add \( \gamma(F) \) to \( P \), but \( C \) is later extended so that \( F \in C^\infty \) holds. We handle this by maintaining a set \( Y \) of ‘delayed’ facts; in line 59 we add \( F \) to \( Y \) if \( F \notin C^\infty \); and in line 40 we identify each ‘delayed’ fact \( G \in C^\infty \cap Y \) and add \( \gamma(G) \) to \( P \).

**Procedure** rewrite(\( a, b \)) implements rewriting: we update \( \gamma \) (line 52), apply the replacement rules \((\approx_1)\)–\((\approx_3)\) to already processed facts containing ‘outdated’ constants (line 54), ensure that \( \Gamma \) is normal w.r.t. \( \gamma \) (line 56), and reapply the normalised rules (lines 57–58). Motik et al. (2015b) discuss in detail the issues related to rule updating and reevaluation.

**Procedure** checkProvability() takes a fact \( F \in I \) and ensures that, for each \( G \in \text{F}^\infty \), we have \( G \in J' \) iff \( \gamma(G) \in P \setminus \hat{P} \)—that is, we know the correct status of each fact that \( F \) represents. To this end, we add \( F \) to \( C \) (line 22) and thus ensure that \( (\gamma, P \setminus \hat{P}) \) correctly represents \( L \) (line 23). Each fact is added to \( C \) only once, which guarantees termination of the recursion. We then use backward chaining to examine facts occurring in proofs of \( F \) and recursively check their provability; we stop at any point during that process if all facts in \( \text{F}^\infty \) become provable (lines 24, 28, 31, and 35). Lines 25–24 handle the reflexivity rules \((\approx_4)\): to check provability of \( c \approx c \), we recursively check the provability each fact containing \( c \). Lines 29–31 handle replacement rules \((\approx_1)\)–\((\approx_3)\): we recursively check the provability of \( c \approx c \) for each constant \( c \) occurring in \( F \). Finally, lines 32–35 handle the rules in \((\pi(I))\): we consider each rule \( r \in \pi(I) \) whose head matches \( F \) and each substitution \( \tau \) that matches the body of \( r \) in \( I \), and we recursively check the provability of \( b(r)\tau \).

**Procedure** \( \text{BF}^\infty() \) computes the set \( D \subseteq I \) of ‘doubtful’ facts. After initialising \( D \) to \( \pi(E^-) \) (lines 3–4), we consider each fact \( F \in D \) (lines 5–16) and determine whether some \( G \in \text{F}^\infty \) is no longer provable (line 6); if so, we add to \( D \) all facts that might be affected by the deletion of \( G \). Lines 9–11 handle rules \((\approx_1)\)–\((\approx_3)\); line 12 handles rules \((\approx_4)\); and lines 13–15 handle rule \((\pi(I))\): we identify each rule \( r \in \pi(I) \) where \( F \) matches a body atom of \( r \), we evaluate the remaining body atoms of \( r \) in \( I \), and we add \( h(r)\tau \) to \( D \) for each \( \tau \) such that \( b(r)\tau \notin I \). Once \( D \) is processed, \((\gamma, P \setminus \hat{P})\) reflects the changes to \((\pi, I)\), which we exploit in Algorithm 2.

### 3.2 Optimisations

**Reflexivity.** Facts of the form \( F = c \approx c \) can be expensive for backward chaining: due to reflexivity rules \((\approx_4)\), in lines 25–28 we may end up recursively proving each fact \( G \) that mentions \( c \). However, \( F \) holds trivially if \( F' \) contains a fact mentioning \( c \), in which case we can consider \( F \) proven and avoid any recursion. This is implemented in lines 37–39.

**Avoiding Redundant Derivations.** Assume that \( I \) contains a rule \( y_1 \approx y_2 \leftarrow (x, R, y_1) \land (x, R, y_2) \), and consider a call to saturate() in which facts \( (a, R, b) \) and \( (a, R, d) \) both end up in \( P \). Unless we are careful, in line 50 we might consider substitution \( \tau_1 = \{ x \mapsto a, y_1 \mapsto b, y_2 \mapsto d \} \) twice: once when we match \( (a, R, b) \) to \( (x, R, y_1) \), and once when we match \( (a, R, d) \) to \( (x, R, y_2) \). Such redundant derivations can substantially degrade performance.

To solve this problem, set \( V \) keeps track of the processed subset of \( P \): after we extract a fact \( F \) from \( P \), in line 42 we transfer \( F \) to \( V \); moreover, in line 49 we evaluate rule bodies in \( V \setminus \hat{P} \) instead of \( P \setminus \hat{P} \). Now if \( (a, R, b) \) is processed before \( (a, R, d) \), at that point we have \( (a, R, d) \notin V \), so \( \tau_1 \) is not returned as a match in line 49; the situation when \( (a, R, d) \) is processed first is analogous. This, however, does not eliminate all repetition: \( \tau_2 = \{ x \mapsto a, y_1 \mapsto b, y_2 \mapsto b \} \) is still considered when \( (a, R, b) \) is matched to either of the two body atoms in the rule. Therefore, we annotate (see Section 3.3) the body atoms of rules so that, whenever \( F \) is matched to some body atom \( B_i \), no atom \( B_j \) preceding \( B_i \) in the body of \( r \) can be matched to \( F \). In our example, \( \tau_2 \) is thus
B/F\^{\omega} avoids redundant derivations in similar vein: set O tracks the processed subset of D; in lines 10 and 14 we match the relevant rules in I \setminus O; and in line 16 we add a fact to O once it has been processed.

\textbf{Disproved Facts.} For each \( F \in I \) with \( F^w \cap J' = \emptyset \), no fact in \( F^w \) participates in a proof of any fact in \( J' \). Thus, in line 7 we collect all such facts in a set \( S \) of disproved facts, and in lines 26, 29, and 33 we exclude \( S \) from backward chaining.

\textbf{Singletons.} If we encounter \( F = c \approx e \) in line 9 or 29 where \( c \) represents only itself (i.e., \( |c^\pi| = 1 \)), then we know that no fact in \( F^w \) can derive a new fact using rules \(( \approx_1)-(\approx_3)\), and so we can avoid considering rules \(( \approx_1)-(\approx_3)\).

\subsection{3.3 Formalisation}

We borrow the notation by Motik \textit{et al.} (2015c) to formalise B/F\^{\omega}. We recapitulate some definitions, present the pseudo-code, and formally state the algorithm’s properties.

Given a dataset \( X \) and a fact \( F \), operation \( X.\text{add}(F) \) adds \( F \) to \( X \), and operation \( X.\text{delete}(F) \) removes \( F \) from \( X \); both return \( t \) if \( X \) was changed. For iteration, operation \( X.\text{next} \) returns the next fact from \( X \), or \( e \) if no such fact exists.

An \textit{annotated query} has the form \( Q = B^\pi_1 \land \cdots \land B^\pi_k \), where each \( B_i \) is a atom and \textit{annotation} \( \bowtie_i \) is either empty or equal to \( \neq \). Given datasets \( X \) and \( Y \) and a substitution \( \sigma \), operation \( X.\text{eval}(Q,Y,\sigma) \) returns a set containing each smallest substitution \( \tau \) such that \( \sigma \subseteq \tau \) and, for \( 1 \leq i \leq k \), (i) \( B_i.\tau \in X \) if \( \bowtie_i \) is empty or (ii) \( B_i.\tau \in Y \) if \( \bowtie_i \) is \( \neq \).

We often write \( \{Z \} \) instead of \( X \), meaning that \( Q \) is evaluated in the difference of sets \( Z \) and \( X \).

Given a fact \( F \), operation \( \Pi.\text{matchHead}(F) \) returns all tuples \( \langle r, Q, \sigma \rangle \) with \( r \in \Pi \) a rule of the form (1), \( \sigma \) a substitution such that \( H.\sigma = F \), and \( Q = B_1 \land \cdots \land B_n \). Moreover, operation \( \Pi.\text{matchBody}(F) \) returns all tuples \( \langle r, Q, \sigma \rangle \) with \( r \in \Pi \) a rule of the form (1), \( \sigma \) a substitution such that \( B_{i+1} = F \) for some \( 1 \leq i \leq n \), an \( Q \) is defined as

\[ Q = B^\pi_1 \land \cdots \land B^\pi_i \land B_{i+1} \land \cdots \land B_n. \]  

Finally, given a mapping \( \gamma \) of constants to constants, and constants \( d \) and \( c \), operation \( \gamma.\text{mergeInto}(d, c, \sigma) \) modifies \( \gamma \) so that \( \gamma(e) = c \) holds for each constant \( e \) with \( \gamma(e) = d \).

\textit{B/F}^\omega consists of Algorithms 1–7. Theorem 1 shows that the algorithm is correct and that, just like the seminal algorithm [Abiteboul \textit{et al.}, 1995], it does not repeat derivations; the proof is given in a technical report [Motik \textit{et al.}, 2015a].

\section{4 Evaluation}

We have implemented and evaluated the B/F\^{\omega} algorithm in the open-source RDF data management system RDFOX. The system and the test data are all available online.\footnote{https://krr-nas.cs.ox.ac.uk/2015/ICJAI/RDFOX/index.html}

\textbf{Objectives.} Updates can be handled either incrementally or by rematerialisation, and equality can be handled either by rewriting or by axiomatisation, giving rise to four possible approaches to updates. Our first objective was to compare all of them to determine their relative strengths and weaknesses. As \( E^- \) increases in size, incremental update becomes harder, but rematerialisation becomes easier. Thus, our second objective was to investigate the relationship between the update size and the performance of the respective approaches.

\textbf{Datasets.} Equality is often used in OWL ontologies on the Semantic Web, so we based our evaluation on several well-known synthetic and ‘real’ RDF datasets.

Each dataset comprises an OWL ontology and a set of explicit facts. UOBM [Ma \textit{et al.}, 2006] extends LUBM [Guo \textit{et al.}, 2005], and we used the data generated for 100 universities; we did not use LUBM because it does not use \( \approx \). Claros contains information about cultural artefacts.\footnote{http://www.clarosnet.org/XDB/ASP/clarosHome/} DBpedia consists of structured information extracted from Wikipedia.\footnote{http://dbpedia.org/} UniProt is a knowledge base about protein sequences;\footnote{http://www.uniprot.org} we selected a subset of the original (very large) set of facts. Finally, OpenCyc is an extensive, manually curated upper ontology.\footnote{http://www.cyc.com/platform/opencyc}

Following Zhou \textit{et al.} (2013), we converted the ontologies into lower (L) and upper bound (U) programs: the former is the OWL 2 RL subset of the ontology transformed into datalog as described by Grosof \textit{et al.} (2003), and the latter captures all consequences of the ontology using an unsound approximation. Upper bound programs are interesting as they tend to be ‘hard’. We also manually extended the lower bound (LE) of Claros with ‘hard’ rules (e.g., we defined related documents as pairs of documents that refer to the same topic).

\textbf{Update Sets.} For each dataset, we randomly selected several subsets \( E^- \) of \( E \). We considered small updates of 100 and 5k facts on all datasets. Moreover, for each dataset we identified the ‘equilibrium’ point \( n \) at which B/F\^{\omega} and Remat\^{\omega} take roughly the same time. If \( n \) was large, we generated subsets \( E^- \) with sizes equal to 25\%, 50\%, 75\%, and 100\% of \( n \); otherwise, we divided \( n \) in an ad hoc way.

\textbf{Test Setting.} We used a Dell server with two 2.60GHz Intel Xeon E5-2670 CPUs and 256 GB of RAM running Fedora release 20, kernel version 3.17.7-200.fc20.x86_64.

\textbf{Test Results.} Table 1 summarises our test results. For each dataset, we show the numbers of explicit facts (\( |E| \)) and rules (\( ||I|| \)), the number of facts in the initial r-materialisation (\( |I^w| \)), and the time \( (T^w) \) and the number of derivations \( (D^w) \) used to compute it via rewriting; moreover, we show the latter three numbers for the initial materialisation computed using axiomatised equality (\( |I^A| \), \( T^A \), and \( D^A \)). For each set \( E^- \), we show the numbers \( \Delta|I^w| \) and \( \Delta|I^A| \) of deleted facts with rewriting and axiomatisation, respectively, as well as the times \( (T) \) and the number of derivations \( (D) \) for each of the four update approaches. All times are in seconds. We could not complete all axiomatisation tests with Claros-LE as each run took about two hours. Due to the upper bound transformation, the r-materialisation of UOBM-100-U contains a constant \( c \) with \( |c^\pi| = 3930 \); thus, when \( \approx \) is axiomatised, deriving just all equalities involving \( c^\pi \) requires 3930\(^3 = 60 \) billion derivations, which causes the initial materialisation to last longer than four hours. The number of derivations \( D \) in
Theorem 1. Let \((\pi, I)\) be the \(r\)-materialisation of a dataset \(E\) w.r.t. a program \(\Pi\), and let \(E^-\) be a dataset.

1. Algorithm 1 terminates, at which point \((\pi, I)\) contains the \(r\)-materialisation of \(E \setminus E^-\) w.r.t. \(\Pi\).

2. Each combination of a rule \(r\) and a substitution \(\tau\) is considered at most once in line 50 or line 58, but not both.

3. Each combination of a rule \(r\) and a substitution \(\tau\) is considered at most once in line 15.
B/F\(\infty\) is the sum of the number of times a fact is determined as ‘doubtful’ (lines 11, 12, and 15), checked in backward chaining (lines 27, 30, and 34), or derived in forward chaining (line 30); we use this number to estimate reasoning difficulty independently from implementation details.

**Discussion.** For updates of 100 facts, B/F\(\infty\) outperforms all other approaches, often by orders of magnitude, and in most cases it does so even for much larger updates.

Even when \(|A^1| - |I^1|\) is ‘small’ (i.e., when not many equalities are derived), B/F\(\infty\) outperforms B/F\(A\). This seems to be mainly because B/F\(A\) ascribes no special meaning to \(\Pi\), and so it does not use the optimisation from lines 37–39; thus, when trying to prove \(c \approx c\), B/F\(A\) performs backward chaining via rules \((\approx c)\) and so it potentially examines each fact containing \(c\). On Claros-L, although \(|A^1| |I^1|\) is of similar sizes, \(I^1\) contains one constant \(c\) with \(|c^1| = 306\), which gives rise to 306\(^3\) derivations; this explains the difference in the performance of B/F\(\infty\) and B/F\(A\).

\(\Pi\) outperforms B/F\(\infty\) in cases similar to those described by Motik et al. (2015c). For example, in UOBM, relation \(\text{hasSameHomeTownWith}\) is symmetric and transitive, which creates cliques of connected constants; B/F always recomputes each changed clique, thus repeating most of the ‘hard’ work. Equality connects constants in cliques, which poses similar problems for B/F\(\infty\). For example, due to the constant \(c\) with \(|c^1| = 3930\), deleting 5k facts in UOBM-100-U results in only 961k (about 1.2% of \(|I^1|\)) facts being added to set \(C\) in line 22, but these facts contribute to 73% of the derivations from the initial \(r\)-materialisation; thus, B/F\(\infty\) repeats Algorithm 5 a substantial portion of the initial work.

On OpenCyc-L, Remat\(\infty\) already outperforms B/F\(\infty\) on updates of 1k tuples, which was surprising since the former makes more derivations than the latter. Our investigation revealed that OpenCyc-L contains about 200 rules of the form \((x, type, y) \leftarrow (x, Ri, y)\) that never fire during forward chaining; however, to check provability of \((a, type, C)\), Algorithm 4 considers in line 32 each time each of the 200 rules. After removing all such ‘idle’ rules manually, B/F\(\infty\) and Remat\(\infty\) could update 1k tuples in roughly the same time. Further analysis revealed that the slowdown in B/F\(\infty\) occurs mainly in line 40: the condition is checked for 13.3M facts \(F\), and these give rise to 139M facts in \(F^2\), each requiring an index lookup; the latter number is similar to the number of derivations in materialisation, which explains the slowdown. We believe one can check this condition more efficiently by using additional book-keeping.

### Table 1: Experimental results

![Table 1: Experimental results](image-url)
5 Conclusion

This paper describes what we believe to be the first approach to incremental maintenance of datalog materialisation when the latter is computed using rewriting—a common optimisation used when programs contain equality. Our algorithm proved to be very effective, particularly on small updates.

In our future work, we shall aim to address the issues we identified in Section 4. For example, to optimise the check in line 40, we shall investigate ways of keeping track of how explicit facts are merged so that we can implement the test by iterating over the appropriate subset of $E$ rather than over $\overline{F}$. Moreover, we believe we can considerably improve the efficiency of both the initial materialisation and the incremental updates by using specialised algorithms for rules that produce large cliques; hence, we shall identify common classes of ‘hard’ rules and then develop such specialised algorithms.

Acknowledgments

This work was funded by the EPSRC projects MaSI, Score!, and DBOnto, and the FP7 project Optique.

References


