The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)

Modular Materialisation of Datalog Programs

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

Abstract

The seminaïve algorithm can be used to materialise all consequences of a datalog program, and it also forms the basis for algorithms that incrementally update a materialisation as the input facts change. Certain (combinations of) rules, however, can be handled much more efficiently using custom algorithms. To integrate such algorithms into a general reasoning approach that can handle arbitrary rules, we propose a modular framework for computing and maintaining a materialisation. We split a datalog program into modules that can be handled using specialised algorithms, and we handle the remaining rules using the seminaïve algorithm. We also present two algorithms for computing the transitive and the symmetric–transitive closure of a relation that can be used within our framework. Finally, we show empirically that our framework can handle arbitrary datalog programs while outperforming existing approaches, often by orders of magnitude.

1 Introduction

Datalog (Abiteboul, Hull, and Vianu 1995) is a prominent rule language whose popularity is mainly due to its ability to express recursive definitions such as transitive closure. Datalog captures OWL 2 RL (Motik et al. 2009) ontologies with SWRL rules (Horrocks et al. 2004), so it supports query answering on the Semantic Web. It has been implemented in many systems, including but not limited to WebPIE (Urbani et al. 2012), VLog (Urbani, Jacobs, and Krötzsch 2016), Oracle's RDF Store (Wu et al. 2008), OWLIM (Bishop et al. 2011a), and RDFox (Nenov et al. 2015).

Datalog reasoning is often realised by precomputing and storing all consequences of a datalog program and a set of facts; this process and its output are both called *materialisation*. A materialisation must be updated when the input facts change, but doing so 'from scratch' can be inefficient if changes are small. To minimise the overall work, *incremental maintenance algorithms* have been developed. These include the well-known *Delete/Rederive* (DRed) (Gupta, Mumick, and Subrahmanian 1993; Staudt and Jarke 1996) and *Counting* (Gupta, Mumick, and Subrahmanian 1993) algorithms, and the more recent *Backward/Forward* (B/F) (Motik et al. 2015), DRed^c, and B/F^c (Hu, Motik, and Horrocks 2018b) algorithms.

Materialisation and all aforementioned incremental algorithms compute rule consequences using *seminaïve evaluation* (Abiteboul, Hull, and Vianu 1995). The main benefit of this approach is that each applicable inference is performed exactly once. However, all consequences of certain rules or rule combinations can actually be computed without considering every applicable inference. For example, consider applying a program that axiomatises a relation R as symmetric and transitive to input facts that describe a connected graph consisting of n vertices. In Section 3 we show that computing all consequences using seminaïve evaluation involves $O(n^3)$ rule applications, whereas a custom algorithm can achieve the same goal using only $O(n^2)$ steps. Since incremental maintenance algorithms are based on the seminaïve algorithm, they can suffer from similar deficiencies.

Approaches that can maintain the closure of specific datalog programs have already been considered in the literature. For example, maintaining transitive closure of a graph has been studied extensively (Ibaraki and Katoh 1983; La Poutre and van Leeuwen 1987; King 1999; Demetrescu and Italiano 2000). Subercaze et al. (2016) presented an algorithm for the materialisation of the transitive and symmetric properties in RDFS-Plus. Dong, Su, and Topor (1995) showed that insertions into a transitively closed relation can be maintained by evaluating four nonrecursive first-order queries. However, these approaches can only handle datalog programs for which they have been specifically developed—that is, the programs are not allowed to contain any additional rules. The presence of other rules introduces additional complexity since updates computed by specialised algorithms must be propagated to the remaining rules and vice versa. Moreover, many of these approaches cannot handle deletion of input facts, which is a key problem in incremental reasoning. Thus, it is currently not clear whether and how customised algorithms can be used in general-purpose datalog systems that must handle arbitrary datalog rules and support incremental additions and deletions.

To address these issues, in this paper we present a modular framework for materialisation computation and incremental materialisation maintenance that can integrate specialised reasoning algorithms with the seminaïve evaluation. The framework partitions the rules of a datalog program into disjoint subsets called *modules*. For each module, four pluggable functions are used to compute certain consequences

of the module's rules; there are no restrictions on how these functions are implemented, as long as their outputs satisfy certain conditions. Moreover, if no specialised algorithm for a module is available, the four functions can be implemented using seminaïve evaluation. Thus, our framework can efficiently handle certain combinations of rules, but it can also handle arbitrary rules while avoiding repeated inferences.

We then examine a module that axiomatises the transitive closure, and a module that axiomatises the symmetric–transitive closure. These modules capture node reachability in directed and undirected graphs, respectively, both of which frequently occur in practice and are thus highly relevant. We present the functions necessary to integrate these modules into our framework and show that they satisfy the properties needed for correctness. We also discuss the kinds of input that are likely to benefit from modular reasoning.

We have implemented our algorithms and compared them on several real-life and synthetic datasets. Our experiments illustrate the potential benefits of the proposed solution: our approach often outperforms state-of-the-art algorithms, sometimes by orders of magnitude. Our system and test data are available online.[1] All proofs of our results are given in a technical report (Hu, Motik, and Horrocks 2018a).

2 Preliminaries

We now introduce datalog with stratified negation. A *term* is a constant or a variable. An *atom* has the form $P(t_1, \dots, t_k)$, where P is a k-ary *predicate* with $k \geq 0$, and each t_i, $1 \leq i \leq k$, is a term. A *fact* is a variable-free atom, and a *dataset* is a finite set of facts. A rule r has the form

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

where $0 \leq m \leq n$, and B_i and H are atoms. For r a rule, $\mathsf{h}(r) = H$ is the *head*, $\mathsf{b}^+(r) = \{B_1, \dots, B_m\}$ is the set of *positive body atoms*, and $\mathsf{b}^-(r) = \{B_{m+1}, \dots, B_n\}$ is the set of *negative body atoms*. Each rule r must be *safe*—that is, each variable occurring in r must occur in at least one positive body atom. A *program* is a finite set of rules.

A *stratification* λ of a program Π maps each predicate occurring in Π to a positive integer such that, for each rule $r \in \Pi$ with predicate P in its head, $\lambda(P) \geq \lambda(R)$ (resp. $\lambda(P) > \lambda(R)$) holds for each predicate R occurring in $\mathsf{b}^+(r)$ (resp. $\mathsf{b}^-(r)$). Such r is *recursive* w.r.t. λ if $\lambda(P) = \lambda(R)$ holds for some predicate R occurring in $\mathsf{b}^+(r)$; otherwise, r is *nonrecursive* w.r.t. λ. Program Π is *stratifiable* if a stratification λ of Π exists. For s an integer, the *stratum* s of Π is the program Π^s containing each rule $r \in \Pi$ whose head predicate P satisfies $\lambda(P) = s$. Moreover, let Π^s_{r} and Π^s_{nr} be the recursive and the nonrecursive subsets, respectively, of Π^s. Finally, let $\mathsf{O}^s = \{P(c_1, \dots, c_n) \mid \lambda(P) = s \text{ and } c_i \text{ are constants}\}$.

A *substitution* σ is a mapping of finitely many variables to constants. For α a term, an atom, a rule, or a set thereof, $\alpha\sigma$ is the result of replacing each occurrence of a variable x in α with $\sigma(x)$, provided that the latter is defined.

If r is a rule and σ is a substitution mapping all variables of r to constants, then rule $r\sigma$ is an *instance* of r. For I a dataset, we define the set $\Pi\big[I\big]$ of all facts obtained by applying a program Π to I as

$$\Pi\big[I\big] = \bigcup_{r \in \Pi} \{\mathsf{h}(r\sigma) \mid \mathsf{b}^+(r\sigma) \subseteq I \text{ and } \mathsf{b}^-(r\sigma) \cap I = \emptyset\}.$$

Let E be a dataset (called *explicit facts*) and let λ be a stratification of Π with maximum stratum index S. Then, let $I^0_\infty = E$; for each $s \geq 1$, let $I^s_0 = I^{s-1}_\infty$, let

$$I^s_i = I^s_{i-1} \cup \Pi^s\big[I^s_{i-1}\big] \text{ for } i > 0, \text{ and let } I^s_\infty = \bigcup_{i \geq 0} I^s_i.$$

Set I^S_∞ is the *materialisation* of Π w.r.t. E and λ. It is known that I^S_∞ does not depend on λ, so we write it as $\mathsf{mat}(\Pi, E)$.

Algorithm 1 $\textsc{Mat}(\Pi, \lambda, E)$

```
1: I := ∅
2: for each stratum index s with 1 ≤ s ≤ S do
3:     Δ := (E ∩ O^s) ∪ Π^s_nr[I]
4:     while Δ ≠ ∅ do
5:         I := I ∪ Δ
6:         Δ := Π^s_r[I : Δ] \ I
```

3 Motivation

In this section we show how custom algorithms can handle certain rule combinations much more efficiently than seminaïve evaluation. We consider here only materialisation, but similar observations apply to incremental maintenance algorithms as most of them use variants of seminaïve evaluation.

3.1 Seminaïve Evaluation

The seminaïve algorithm (Abiteboul, Hull, and Vianu 1995) takes as input a set of explicit facts E, a program Π, and a stratification λ of Π, and it computes $\mathsf{mat}(\Pi, E)$. To apply each rule instance at most once, in each round of rule application it identifies the 'newly applicable' rule instances (i.e., instances that depend on a fact derived in the previous round) as shown in Algorithm 1. For each stratum, the algorithm initialises Δ, the set of newly derived facts, by combining the explicit facts in the current stratum ($E \cap \mathsf{O}^s$) with the facts derivable from previous strata via nonrecursive rules ($\Pi^s_{\mathsf{nr}}\big[I\big]$). Then, in lines 4–6 it iteratively computes all consequences of Δ. To this end, in line 6 it uses operator $\Pi\big[I : \Delta\big]$, which extends $\Pi\big[I\big]$ to allow identifying 'newly applicable' rule instances. Specifically, given datasets I and $\Delta \subseteq I$, operator $\Pi\big[I : \Delta\big]$ returns a set containing $\mathsf{h}(r\sigma)$ for each rule $r \in \Pi$ and substitution σ such that $\mathsf{b}^+(r\sigma) \subseteq I$ and $\mathsf{b}^-(r\sigma) \cap I = \emptyset$ hold (i.e., rule instance $r\sigma$ is applicable to I), but also $\mathsf{b}^+(r\sigma) \cap \Delta \neq \emptyset$ holds (i.e., a positive body atom of $r\sigma$ occurs in the set of facts Δ derived in the previous round of rule application). It is not hard to see that the algorithm computes $I = \mathsf{mat}(\Pi, E)$, and that it considers each rule instance $r\sigma$ at most once.

3.2 Problems with the Seminaïve Evaluation

Although seminaïve evaluation does not repeat derivations, it always considers each applicable rule instance. However,

[1] http://krr-nas.cs.ox.ac.uk/2018/modular/

facts are often derived via multiple, distinct rule instances; this is particularly common with recursive rules, but it can also occur with nonrecursive rules only. We are unaware of a general technique that can prevent such derivations. We next present two programs for which materialisation can be computed without considering all applicable rule instances, thus showing how seminaïve evaluation can be suboptimal.

Example 1. *Let* Π *be the program containing rule* (1) *and let* $E = \{R(c_i, c_{i+1}) \mid 0 \leq i \leq n\}$.

$$R(x, y) \wedge R(y, z) \rightarrow R(x, z) \tag{1}$$

Clearly, $I = \mathsf{mat}(\Pi, E) = \{R(c_i, c_j) \mid 0 \leq i < j \leq n\}$*, so each rule instance of the form*

$$R(c_i, c_j) \wedge R(c_j, c_k) \rightarrow R(c_i, c_k) \tag{2}$$

with $1 \leq i < j < k \leq n$ *is applicable to* I*. Algorithm 1 considers all of these* $O(n^3)$ *rule instances.*

We next present an outline of an approach that is still cubic in general, but on this specific input runs in $O(n^2)$ *time. The key is to distinguish the set* X *of 'external' facts given to* Π *as input from the 'internal' facts derived by* Π*. We can transitively close* R *by iteratively considering pairs of facts* $R(u, v) \in X$ *and* $R(v, w)$*. That is, we require the first fact to be in* X*, but place no restriction on the second fact. (We could have equivalently required the second fact to be in* X*.) In our example, we have* $X = E$*, so the algorithm considers only rule instances of the form*

$$R(c_i, c_{i+1}) \wedge R(c_{i+1}, c_k) \rightarrow R(c_i, c_k) \tag{3}$$

for $0 \leq i < k \leq n$*, of which there are* $O(n^2)$ *many. Intuitively, this is analogous to replacing the predicate* R *in all explicit facts with* X*, and using a linear rule*

$$X(x, y) \wedge R(y, z) \rightarrow R(x, z) \tag{4}$$

instead of rule (1)*. In our approach, however, other rules can derive* R*-facts so the set* X *is not fixed; thus, rule* (1) *cannot be simply replaced with* (4)*. Our approach 'simulates' such linearisation, and it can be expected to perform well whenever the other rules derive fewer facts than rule* (1)*.*

Example 2. *Let* Π *consist of rules* (1) *and* (5)*, and let* $E = \{R(c_i, c_{i+1}) \mid 1 \leq i < n\} \cup \{R(c_n, c_1)\}$.

$$R(x, y) \rightarrow R(y, x) \tag{5}$$

Now $I = \mathsf{mat}(\Pi, E) = \{R(c_i, c_j) \mid 1 \leq i, j \leq n\}$*, so each instance of the form* (2) *with* $1 \leq i, j, k \leq n$ *is applicable to* I*. Algorithm 1 considers all of these* $O(n^3)$ *rule instances.*

However, we can view any relation R *as an undirected graph with* n *vertices. To compute the symmetric–transitive closure of* R*, we first compute the connected components of* R*, and, for each connected component* C*, we enumerate all* $u, v \in C$ *and derive* $R(u, v)$*. The first step is linear in the size of* R *and the second step requires* $O(n^2)$ *time, so the algorithm runs in* $O(n^2)$ *time on any* R*.*

4 Framework

In this section we present a general framework for materialisation and incremental reasoning that can avoid the deficiencies outlined in Section 3 for certain rule combinations. Our framework focuses on recursive rules only: nonrecursive rules Π^s_{nr} are evaluated just once in each stratum, which is usually efficient. In contrast, the recursive part Π^s_{r} of each stratum Π^s must be evaluated iteratively, which is a common source of inefficiency. Thus, our framework splits Π^s_{r} into $n(s)$ mutually disjoint, nonempty programs $\Pi^{s,i}_{\mathsf{r}}$, $1 \leq i \leq n(s)$, called *modules*. (We let $n(s) = 0$ if $\Pi^s_{\mathsf{r}} = \emptyset$.) Our notion of modules should not be confused with ontology modules: the latter are subsets of an ontology that are semantically independent from each other in a well-defined way, whereas our modules are just arbitrary program subsets. Each module is handled using 'plugin' functions that compute certain consequences of $\Pi^{s,i}_{\mathsf{r}}$. These functions can be implemented as desired, as long as their results satisfy certain properties that guarantee correctness. We present our framework in two steps: in Section 4.1 we consider materialisation, and in Section 4.2 we focus on incremental reasoning. Then, in Sections 5 and 6 we discuss how to realise these 'plugin' functions for certain common modules.

Before proceeding, we generalise operator $\Pi\big[I \colon \Delta\big]$ as follows. Given datasets I^{p}, I^{n}, Δ^{p}, and Δ^{n} where $\Delta^{\mathsf{p}} \subseteq I^{\mathsf{p}}$ and $\Delta^{\mathsf{n}} \cap I^{\mathsf{n}} = \emptyset$, let

$$\begin{aligned}\Pi\big[I^{\mathsf{p}}, I^{\mathsf{n}} \colon \Delta^{\mathsf{p}}, \Delta^{\mathsf{n}}\big] = \textstyle\bigcup_{r \in \Pi}\{\mathsf{h}(r\sigma) \mid \\ \mathsf{b}^+(r\sigma) \subseteq I^{\mathsf{p}} \text{ and } \mathsf{b}^-(r\sigma) \cap I^{\mathsf{n}} = \emptyset, \text{ and} \\ \mathsf{b}^+(r\sigma) \cap \Delta^{\mathsf{p}} \neq \emptyset \text{ or } \mathsf{b}^-(r\sigma) \cap \Delta^{\mathsf{n}} \neq \emptyset\}.\end{aligned}$$

When the condition in the last line is not required, we simply write $\Pi\big[I^{\mathsf{p}}, I^{\mathsf{n}}\big]$. Moreover, we omit I^{n} when $I^{\mathsf{p}} = I^{\mathsf{n}}$, and we omit Δ^{n} when $\Delta^{\mathsf{n}} = \emptyset$. Intuitively, this operator computes the consequences of Π by evaluating the positive and the negative body atoms in I^{p} and I^{n}, respectively, while ensuring in each derivation that either a positive or a negative body atom is true in Δ^{p} or Δ^{n}, respectively. Our incremental algorithm uses this operator to identify the consequences of Π that are affected by the changes to the facts matching the positive and the negative body atoms of the rules in Π. For example, if the facts in Δ^{p} are added to (resp. removed from) the materialisation, then $\Pi\big[I^{\mathsf{p}} \colon \Delta^{\mathsf{p}}\big]$ contains the consequences of the rule instances that start (resp. cease) to be applicable because a positive body atom matches to a fact in Δ^{p}. The set Δ^{n} is used to analogously capture the consequences of the negative body atoms of the rules in Π.

4.1 Computing the Materialisation

Our modular materialisation algorithm uses a 'plugin' function $\mathsf{Add}^{s,i}$ for each module $\Pi^{s,i}_{\mathsf{r}}$. The function takes as arguments datasets I^{p}, I^{n}, and Δ such that $\Delta \subseteq I^{\mathsf{p}}$, and it closes I^{p} with all consequences of $\Pi^{s,i}_{\mathsf{r}}$ that depend on Δ. Each invocation of these functions must satisfy the following properties in order to guarantee correctness of our algorithm.

Definition 3. *Function* Add captures *a datalog program* Π *on datasets* I^{p}*,* I^{n}*, and* Δ *with* $\Delta \subseteq I^{\mathsf{p}}$ *if the result of* $\mathsf{Add}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$ *is the smallest dataset* J *that satisfies* $\Pi\big[I^{\mathsf{p}} \cup J, I^{\mathsf{n}} \colon \Delta \cup J\big] \subseteq I^{\mathsf{p}} \cup J$.

For brevity, in the rest of the paper we often say just 'Add captures Π' without specifying the datasets whenever the latter are clear from the context. In the absence of a customised algorithm, Add can always be realised using the seminaïve evaluation strategy as follows:

Algorithm 2 MAT-MOD(Π, λ, E)

```
 7: I := ∅
 8: for each stratum index s with 1 ≤ s ≤ S do
 9:     Δ_1 := ··· := Δ_{n(s)} := ∅
10:     Δ := (E ∩ O^s) ∪ Π^s_nr[I]
11:     while Δ ≠ ∅ do
12:         I := I ∪ Δ
13:         for each i with 1 ≤ i ≤ n(s) do
14:             Δ_i := Add^{s,i}[I, I, Δ \ Δ_i]
15:         Δ := Δ_1 ∪ ··· ∪ Δ_{n(s)}
```

- let $\Delta_0 = \Delta$ and $J_0 = \emptyset$,
- for i starting with 0 onwards, if $\Delta_i = \emptyset$, stop and return J_i; otherwise, let $\Delta_{i+1} = \Pi\big[I^{\mathsf{p}} \cup J_i, I^{\mathsf{n}} : \Delta_i\big] \setminus (I^{\mathsf{p}} \cup J_i)$ and $J_{i+1} = J_i \cup \Delta_{i+1}$ and proceed to $i + 1$.

However, a custom implementation of Add will typically not examine all rule instances from the above computation in order to optimise reasoning with certain modules.

Algorithm 2 formalises our modular approach to datalog materialisation. It takes as input a program Π, a stratification λ of Π, and a set of explicit facts E, and it computes $\mathsf{mat}(\Pi, E)$. The algorithm's structure is similar to Algorithm 1. For each stratum of Π, both algorithms first apply the nonrecursive rules, and then they apply the recursive rules iteratively up to a fixpoint. The main difference is that, given a set of facts Δ derived from the previous iteration, Algorithm 2 computes the consequences of Δ for each module independently using $\mathsf{Add}^{s,i}$ (line 14); note that each Δ_i is closed under $\Pi^{s,i}_{\mathsf{r}}$, which is key to the performance of our approach. The algorithm then combines the consequences of all modules (line 15) before proceeding to the next iteration.

If each $\mathsf{Add}^{s,i}$ function is implemented using seminaïve evaluation as described earlier, then the algorithm does not consider a rule instance more than once. This is achieved by passing $\Delta \setminus \Delta_i$ to $\mathsf{Add}^{s,i}$ in line 14: only facts derived by other modules in the previous iteration are considered 'new' for $\mathsf{Add}^{s,i}$, which is possible since the facts in Δ_i have been produced by the ith module in the previous iteration. Theorem 4 captures these properties formally.

Theorem 4. *Algorithm 2 computes I as $\mathsf{mat}(\Pi, E)$ if function $\mathsf{Add}^{s,i}$ captures $\Pi^{s,i}_{\mathsf{r}}$ in each of its calls. Moreover, if all $\mathsf{Add}^{s,i}$ use the seminaïve strategy, each applicable rule instance is considered at most once.*

4.2 Incremental Updates

Our modular incremental materialisation maintenance algorithm is based on the DRed^c algorithm by Hu, Motik, and Horrocks (2018b), which is a variant of the well-known DRed algorithm (Gupta, Mumick, and Subrahmanian 1993). For each fact, DRed^c maintains two counters that track the number of nonrecursive and recursive derivations of the fact. The algorithm proceeds in three steps. During the deletion phase, DRed^c iteratively computes the consequences of the deleted facts, similar to DRed, while adjusting the counters accordingly. However, to optimise overdeletion, deletion propagation stops on facts with a nonzero nonrecursive counter. In the one-step rederivation phase, DRed^c identifies the facts that were overdeleted but can be rederived from the remaining facts in one step by simply checking the recursive counters: if the counter is nonzero, then the corresponding fact is rederived. In the insertion phase, DRed^c computes the consequences of the rederived and the inserted facts using seminaïve evaluation, which we have already discussed.

Our modular incremental algorithm handles nonrecursive rules in the same way as DRed^c. Thus, the nonrecursive counters, which record the number of nonrecursive derivations of each fact, can be maintained globally just as in DRed^c. In contrast, as discussed in Section 3, custom algorithms for recursive modules will usually not consider all applicable rule instances, so counters of recursive derivations cannot be maintained globally. Nevertheless, certain modules can maintain recursive counters internally (e.g., the module based on the seminaïve evaluation can do so).

In addition to function $\mathsf{Add}^{s,i}$ from Section 4.1, our modular incremental reasoning algorithm uses three further functions: $\mathsf{Diff}^{s,i}$, $\mathsf{Del}^{s,i}$, and $\mathsf{Red}^{s,i}$. Definition 5 captures the requirements on $\mathsf{Diff}^{s,i}$. Intuitively, $\mathsf{Diff}^{s,i}[I^{\mathsf{p}}, \Delta^{\mathsf{p}}, \Delta^{\mathsf{n}}]$ identifies the consequences of $\Pi^{s,i}_{\mathsf{r}}$ affected by the addition of the facts in Δ^{p} and removal of the the facts Δ^{n}, respectively, with both sets containing facts from earlier strata.

Definition 5. *Function* Diff captures *a datalog program Π on datasets I^{p}, Δ^{p}, and Δ^{n} where $\Delta^{\mathsf{p}} \subseteq I^{\mathsf{p}}$, $\Delta^{\mathsf{n}} \cap I^{\mathsf{p}} = \emptyset$, and both Δ^{p} and Δ^{n} do not contain predicates occurring in rule heads in Π if $\mathsf{Diff}[I^{\mathsf{p}}, \Delta^{\mathsf{p}}, \Delta^{\mathsf{n}}] = \Pi\big[I^{\mathsf{p}} : \Delta^{\mathsf{p}}, \Delta^{\mathsf{n}}\big]$.*

Function $\mathsf{Del}^{s,i}$ captures overdeletion: if the facts in Δ are deleted, then $\mathsf{Del}^{s,i}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta, C_{\mathsf{nr}}]$ returns the consequences of $\Pi^{s,i}_{\mathsf{r}}$ that must be overdeleted as well. The function can use the nonrecursive counters C_{nr} in order to stop overdeletion as in DRed^c. We do not specify exactly what the functions must return: as we discuss in Section 6, computing the smallest set that needs to be overdeleted might require considering all rule instances as in DRed^c, which would miss the point of modular reasoning. Instead, we specify the required output in terms of a lower bound J_l and an upper bound J_u. Intuitively, J_l and J_u contain facts that would be overdeleted in DRed^c and DRed, respectively.

Definition 6. *Function* Del captures *a datalog program Π on datasets I^{p}, I^{n}, Δ with $\Delta \subseteq I^{\mathsf{p}}$, and a mapping C_{nr} of facts to integers if $J_l \subseteq \mathsf{Del}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta, C_{\mathsf{nr}}] \subseteq J_u$ where*

- *the* lower bound *J_l is the smallest dataset such that, for each $F \in \Pi\big[I^{\mathsf{p}}, I^{\mathsf{n}} : \Delta \cup J_l\big]$, either $F \in \Delta \cup J_l$ or $C_{\mathsf{nr}}(F) > 0$ holds, and*
- *the* upper bound *J_u is the smallest dataset that satisfies $\Pi\big[I^{\mathsf{p}}, I^{\mathsf{n}} : \Delta \cup J_u\big] \subseteq \Delta \cup J_u$.*

Finally, function $\mathsf{Red}^{s,i}$ captures rederivation: if facts in Δ are overdeleted, then $\mathsf{Red}^{s,i}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$ returns all facts in Δ that can be rederived from $I^{\mathsf{p}} \setminus \Delta$ and $\Pi^{s,i}_{\mathsf{r}}$ in *one or more steps*. This is different from DRed and DRed^c, which both perform only one-step rederivation. This change is important in our framework because, as we shall see in Section 6, $\mathsf{Red}^{s,i}$ provides the opportunity for a module to adjust its internal data structures after deletion.

Definition 7. *Function* $\mathsf{Red}^{s,i}$ captures *a datalog program* Π *on datasets* I^{p}, I^{n}, Δ *with* $\Delta \subseteq I^{\mathsf{p}}$ *if the result of* $\mathsf{Red}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$ *is the smallest dataset* J *that satisfies* $\Pi\big[(I^{\mathsf{p}} \setminus \Delta) \cup J, I^{\mathsf{n}}\big] \cap \Delta \subseteq J$.

Algorithm 3 formalises our modular approach to incremental maintenance. The algorithm takes as input a program Π, a stratification λ of Π, a set of explicit facts E, the materialisation $I = \mathsf{mat}(\Pi, E)$, the sets of facts E^- and E^+ to delete from and add to E, and a map C_{nr} that records the number of nonrecursive derivations of each fact. The algorithm updates I to $\mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$. We next describe the two main steps of the algorithm.

In the overdeletion phase, the algorithm first initialises the set of facts to delete Δ as the union of the explicitly deleted facts ($E^- \cap \mathsf{O}^s$) and the facts affected by changes in previous strata (lines 22 and 23). Then, in lines 26–30 the algorithm computes all consequences of Δ. In each iteration, function $\mathsf{Del}^{s,i}$ is called for each module to identify the consequences of $\Pi^{s,i}_{\mathsf{r}}$ that must be overdeleted due to the deletion of Δ (line 28). As in Algorithm 2, the third argument of $\mathsf{Del}^{s,i}$ is $\Delta \setminus \Delta_i$, which guarantees that the function will not be applied to its own consequences.

In the second step, the algorithm first identifies the rederivable facts by calling $\mathsf{Red}^{s,i}$ for each module (lines 33–35). Then, the consequences of the rederived facts, the explicitly added facts ($E^+ \cap \mathsf{O}^s$), and the facts added due to changes in previous strata are computed in the loop of lines 36–40 analogously to Algorithm 2. Although $\mathsf{Red}^{s,i}$ rederives facts in one or more steps as opposed to the one-step rederivation in DRed and DRed^c, this extra effort is not repeated during insertion since $\mathsf{Add}^{s,i}$ is not applied to the consequences of module i. Theorem 8 states that the algorithm is correct.

Theorem 8. *Algorithm 3 updates* I *from* $\mathsf{mat}(\Pi, E)$ *to* $\mathsf{mat}(\Pi, (E \setminus E^-) \cup E^+)$ *if functions* $\mathsf{Add}^{s,i}$, $\mathsf{Del}^{s,i}$, $\mathsf{Diff}^{s,i}$, *and* $\mathsf{Red}^{s,i}$ *capture* $\Pi^{s,i}_{\mathsf{r}}$ *in all of their calls.*

Algorithm 3 DRED^c-MOD$(\Pi, \lambda, E, I, E^-, E^+, C_{\mathsf{nr}})$

16: $D := A := \emptyset, \quad E^- = (E^- \cap E) \setminus E^+, \quad E^+ = E^+ \setminus E$
17: **for each** stratum index s with $1 \leq s \leq S$ **do**
18: OVERDELETE
19: REDERIVE-INSERT
20: $E := (E \setminus E^-) \cup E^+, \quad I := (I \setminus D) \cup A$

21: **procedure** OVERDELETE
22: $\Delta_1 := \cdots := \Delta_{n(s)} := \emptyset$
23: $\Delta := (E^- \cap \mathsf{O}^s) \cup \Pi^s_{\mathsf{nr}}\big[I \colon D \setminus A, A \setminus D\big]$ and update C_{nr}
24: **for each** i with $1 \leq i \leq n(s)$ **do**
25: $\Delta := \Delta \cup \mathsf{Diff}^{s,i}[I, D \setminus A, A \setminus D]$
26: **while** $\Delta \neq \emptyset$ **do**
27: **for each** i with $1 \leq i \leq n(s)$ **do**
28: $\Delta_i := \mathsf{Del}^{s,i}[I \setminus (D \setminus A), I \cup A, \Delta \setminus \Delta_i, C_{\mathsf{nr}}]$
29: $D := D \cup \Delta$
30: $\Delta := \Delta_1 \cup \cdots \cup \Delta_{n(s)}$

31: **procedure** REDERIVE-INSERT
32: $\Delta := (E^+ \cap \mathsf{O}^s) \cup \Pi^s_{\mathsf{nr}}\big[(I \setminus D) \cup A \colon A \setminus D, D \setminus A\big]$ and update C_{nr}
33: **for each** i with $1 \leq i \leq n(s)$ **do**
34: $\Delta_i := \mathsf{Red}^{s,i}[I, (I \setminus D) \cup A, D \setminus A]$
35: $\Delta := \Delta \cup \Delta_i \cup \mathsf{Diff}^{s,i}[(I \setminus D) \cup A, A \setminus D, D \setminus A]$
36: **while** $\Delta \neq \emptyset$ **do**
37: $A := A \cup \Delta$
38: **for each** i with $1 \leq i \leq n(s)$ **do**
39: $\Delta_i := \mathsf{Add}^{s,i}[(I \setminus D) \cup A, (I \setminus D) \cup A, \Delta \setminus \Delta_i]$
40: $\Delta := \Delta_1 \cup \cdots \cup \Delta_{n(s)}$

5 Transitive Closure

We now consider a module consisting of a single rule (1) axiomatising a relation R as transitive. Following the ideas from Example 1, we distinguish the 'internal' facts produced by rule (1) from the 'external' facts produced by other rules. We keep track of the latter in a global set X_R that is initialised to the empty set. A key invariant of our approach is that each fact $R(a_0, a_n)$ is produced by a chain $\{R(a_0, a_1), \ldots, R(a_{n-1}, a_n)\} \subseteq X_R$ of 'external' facts. Thus, we can transitively close R by considering pairs of R-facts where at least one of them is contained in X_R, which can greatly reduce the number of inferences. A similar effect could be achieved by rewriting the input program: we introduce a fresh predicate X_R, and we replace by X_R each occurrence of R in the head of a rule, as well as one of the two occurrences of R in the body of rule (1). Such an approach, however, introduces the facts containing the auxiliary predicate X_R into the materialisation and thus reveals implementation details to the users. Moreover, the rederivation step can be realised very efficiently in our approach.

Based on the above idea, function $\mathsf{Add}^{\mathsf{tc}(R)}$, shown in Algorithm 4, essentially implements seminaïve evaluation for rule (4): the loops in lines 42–43 and 44–47 handle the two delta rules derived from (4). For $\mathsf{Diff}^{\mathsf{tc}(R)}$, note that sets $A \setminus D$ and $D \setminus A$ in lines 25 and 35 of Algorithm 3 always contain facts with predicates that do not occur in $\Pi^{s,i}_{\mathsf{r}}$ in rule heads; thus, since R occurs in the head of rule (1), these sets contain facts whose predicate is different from R, so $\mathsf{Diff}^{\mathsf{tc}(R)}$ can simply return the empty set. Function $\mathsf{Del}^{\mathsf{tc}(R)}$, shown in Algorithm 5, implements seminaïve evaluation for rule (4) analogously to $\mathsf{Add}^{\mathsf{tc}(R)}$. The main difference is that only facts whose nonrecursive counter is zero are overdeleted, which mimics overdeletion in DRed^c. As a result, not all facts processed in lines 50 and 53 are added to J so, to avoid repeatedly considering such facts, the algorithm maintains the set S of 'seen' facts. Finally, function $\mathsf{Red}^{\mathsf{tc}(R)}$, shown in Algorithm 6, identifies for each source vertex u all vertices reachable by the external facts in X_R.

Theorem 9. *In each call in Algorithms 2 and 3, functions* $\mathsf{Add}^{\mathsf{tc}(R)}$, $\mathsf{Del}^{\mathsf{tc}(R)}$, $\mathsf{Diff}^{\mathsf{tc}(R)}$, *and* $\mathsf{Red}^{\mathsf{tc}(R)}$ *capture a datalog program that axiomatises relation* R *as transitive.*

6 Symmetric–Transitive Closure

We now consider a module consisting of two rules, (1) and (5), axiomatising a relation R as transitive and symmetric. As in Example 2, we can view relation R as an undirected graph. To compute the materialisation, we extract the set C_R

Algorithm 4 $\mathsf{Add}^{\mathsf{tc}(R)}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$

41: $J := \emptyset, \quad Q := \Delta, \quad X_R := X_R \cup \Delta$
42: **for each** $R(u, v) \in \Delta$ and **each** $R(v, w) \in I^{\mathsf{p}} \setminus \Delta$ **do**
43: **add** $R(u, w)$ to Q and J
44: **while** $Q \neq \emptyset$ **do**
45: **remove** an arbitrarily chosen fact $R(v, w)$ from Q
46: **for each** $R(u, v) \in X_R$ such that $R(u, w) \notin I^{\mathsf{p}} \cup J$ **do**
47: **add** $R(u, w)$ to Q and J
48: **return** J

Algorithm 5 $\mathsf{Del}^{\mathsf{tc}(R)}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta, C_{\mathsf{nr}}]$

49: $J := \emptyset, \quad Q := S := \Delta, \quad X_R := X_R \setminus \Delta$
50: **for each** $R(u, v) \in \Delta$ and **each** $R(v, w) \in I^{\mathsf{p}} \setminus S$ **do**
51: **add** $R(u, w)$ to Q and S
52: **if** $C_{\mathsf{nr}}(R(u, w)) = 0$ **then add** $R(u, w)$ to J
53: **while** $Q \neq \emptyset$ **do**
54: **remove** an arbitrarily chosen fact $R(v, w)$ from Q
55: **for each** $R(u, v) \in X_R$ such that $R(u, w) \in I^{\mathsf{p}} \setminus S$ **do**
56: **add** $R(u, w)$ to Q and S
57: **if** $C_{\mathsf{nr}}(R(u, w)) = 0$ **then add** $R(u, w)$ to J
58: **return** $J \setminus \Delta$

Algorithm 6 $\mathsf{Red}^{\mathsf{tc}(R)}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$

59: $J := \emptyset$
60: **for each** u such that there exist v with $R(u, v) \in \Delta$ **do**
61: **for each** w reachable from u via R facts in X_R **do**
62: **add** $R(u, w)$ to J
63: **return** $J \cap \Delta$

Algorithm 7 $\mathsf{Add}^{\mathsf{stc}(R)}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$

64: **return** CLOSEEDGES$(\Delta) \setminus I^{\mathsf{p}}$
65: **function** CLOSEEDGES(Δ)
66: $J := \emptyset$
67: **for each** $R(u, v) \in \Delta$ **do**
68: **if** no $U \in C_R$ exists such that $u \in U$ **then**
69: **add** $\{u\}$ to C_R, and $R(u, u)$ to J
70: **if** no $V \in C_R$ exists such that $v \in V$ **then**
71: **add** $\{v\}$ to C_R, and $R(v, v)$ to J
72: **if** u and v belong to distinct $U, V \in C_R$, resp. **then**
73: **remove** U and V from C_R, and **add** $U \cup V$ to C_R
74: **for each** $u' \in U$ and **each** $v' \in V$ **do**
75: **add** $R(u', v')$ and $R(v', u')$ to J
76: **return** J

Algorithm 8 $\mathsf{Del}^{\mathsf{stc}(R)}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta, C_{\mathsf{nr}}]$

77: $J := \emptyset$
78: **for each** $U \in C_R$ where $\exists R(u, v) \in \Delta$ s.t. $\{u, v\} \subseteq U$ **do**
79: **for each** $u' \in U$ and **each** $v' \in U$ **do**
80: **if** $C_{\mathsf{nr}}(R(u', v')) = 0$ **then add** $R(u', v')$ to J
81: **else add** $R(u', v')$ to Y_R
82: **remove** U from C_R
83: **return** $J \setminus \Delta$

Algorithm 9 $\mathsf{Red}^{\mathsf{stc}(R)}[I^{\mathsf{p}}, I^{\mathsf{n}}, \Delta]$

84: $J :=$ CLOSEEDGES$(Y_R) \cap \Delta$
85: $Y_R := \emptyset$
86: **return** J

of connected components—that is, each $U \in C_R$ is a set of mutually connected vertices in the symmetric–transitive closure of R; finally, we derive $R(u, v)$ for all u and v in each component $U \in C_R$. Set C_R is global and is initially empty.

Based on this idea, function $\mathsf{Add}^{\mathsf{stc}(R)}$, shown in Algorithm 7, uses an auxiliary function CLOSEEDGES to incrementally update the set C_R by processing each fact $R(u, v) \in \Delta$ in lines 67–75: if either u or v does not occur in a component in C_R, then the respective component is created in C_R (lines 69 and 71); and if u and v belong to distinct components U and V, then U and V are merged into a single component and all R-facts connecting U and V are added (lines 72–75). For the same reasons as in Section 5, function $\mathsf{Diff}^{\mathsf{tc}(R)}$ can simply return the empty set. Function $\mathsf{Del}^{\mathsf{stc}(R)}$, shown in Algorithm 8, simply overdeletes all facts $R(u', v')$ whose nonrecursive counter is zero and where both u' and v' belong to a component U containing both vertices of a fact $R(u, v)$ in Δ. Those facts $R(u', v')$ for which the nonrecursive counter is nonzero will hold after overdeletion, so they are kept in an initially empty global set Y_R so that they can be used for rederivation later. Finally, function $\mathsf{Red}^{\mathsf{stc}(R)}$, shown in Algorithm 9, simply closes the set Y_R in the same way as during addition, and it empties the set Y_R. While this creates a dependency between $\mathsf{Del}^{\mathsf{stc}(R)}$ and $\mathsf{Red}^{\mathsf{stc}(R)}$, the order in which these functions are called in Algorithm 3 ensures that the set Y_R is maintained correctly.

Theorem 10. *In each call in Algorithms 2 and 3, functions* $\mathsf{Add}^{\mathsf{stc}(R)}$, $\mathsf{Del}^{\mathsf{stc}(R)}$, $\mathsf{Diff}^{\mathsf{stc}(R)}$, *and* $\mathsf{Red}^{\mathsf{stc}(R)}$ *capture a datalog program that axiomatises R as symmetric–transitive.*

7 Evaluation

We have implemented our modular materialisation and incremental maintenance algorithms, as well as the seminaïve materialisation and the DRed^c algorithms, and we have compared their performance empirically.

Test Benchmarks We used the following real-world and synthetic benchmarks in our tests. LUBM (Guo, Pan, and Heflin 2005) is a well-known benchmark that models individuals and organisations in a university domain. Claros describes archeological artefacts. We used the LUBM and Claros datasets with the *lower bound extended* (-LE) programs by Motik et al. (2014); roughly speaking, we converted a subset of the accompanying OWL ontologies into datalog and manually extended them with several 'difficult rules'. DBpedia (Lehmann et al. 2015) contains structured information extracted from Wikipedia. DBpedia represents Wikipedia categories using the SKOS vocabulary (Miles and Bechhofer 2009), which defines several transitive properties. We used the datalog subset of the SKOS RDF schema. Moreover, the materialisation of DBpedia-SKOS is too large to fit into the memory of our test server, so we used a random sample of the DBpedia dataset consisting of five mil-

Benchmark	$\lvert E\rvert$	$\lvert I\rvert$	S	$\lvert\Pi_{\mathsf{nr}}\rvert$	$\lvert\Pi_{\mathsf{r}}\rvert$	$\lvert$TC$\rvert$	$\lvert$STC$\rvert$	Mat-Mod	Mat
Claros-LE	18.8 M	533.3 M	11	1031	306	27	2	733.55	3593.32
LUBM-LE	133.6 M	332.6 M	5	85	22	1	2	291.90	1100.22
DBpedia-SKOS	5.0 M	97.0 M	5	26	15	2	1	103.23	3623.37
DAG-R	0.1 M	22.9 M	1	1	1	1	0	29.60	3238.86

Table 1: Running times for materialisation computation (seconds)

Benchmark	Small Deletions		Small Insertions		Large Deletions	
	DRed^c-Mod	DRed^c	DRed^c-Mod	DRed^c	DRed^c-Mod	DRed^c
Claros-LE	0.93	1035.28	0.17	0.80	314.33	3616.93
LUBM-LE	0.32	3.87	0.01	0.01	182.93	1369.77
DBpedia-SKOS	21.77	691.32	0.20	2.78	111.28	3826.87
DAG-R	64.92	3005.11	14.56	116.78	62.48	4316.71

Table 2: Running times for incremental maintenance (seconds)

lion facts. Finally, DAG-R is a synthetic benchmark consisting of a randomly generated dataset containing a directed acyclic graph with 10k nodes and 100k edges, and a program that axiomatises the path relation as transitive. Table 1 shows the numbers of explicit facts ($|E|$), derived facts ($|I|$), strata (S), nonrecursive rules ($|\Pi_{\mathsf{nr}}|$), recursive rules ($|\Pi_{\mathsf{r}}|$), transitivity modules (|TC|), and symmetric–transitive modules (|STC|) for each benchmark.

Test Setup and Results We conducted all experiments on a Dell PowerEdge R730 server with 512 GB RAM and two Intel Xeon E5-2640 2.6 GHz processors running Fedora 27, kernel version 4.17.6. For each benchmark, we loaded the test data into our system and then compared the performance of our modular algorithms with the seminaïve and DRed^c algorithms using the following methodology.

We first computed the materialisation and measured the wall-clock time. The results are shown in Table 1. We then conducted two groups of incremental reasoning tests.

In the first group, we tested the performance of our incremental algorithms on small changes. To this end, we used uniform sampling to select ten subsets $E_i \subseteq E$, $1 \leq i \leq 10$, each consisting of 1000 facts from the input dataset. We deleted and then reinserted E_i for each i while measuring the wall-clock times, and then we computed the average times for deletion and insertion over the ten samples. The results are shown in the 'Small Deletions' and 'Small Insertions' columns of Table 2, respectively.

In the second group, we tested the performance of incremental algorithms on large deletions. To this end, we used uniform sampling to select a subset $E^- \subseteq E$ containing 25% of the explicit facts, and we measured the wall-clock time needed to delete E^- from the materialisation. The results are shown in the 'Large Deletions' column of Table 2. We did not consider large insertions because our algorithms handle insertion in the same way as materialisation, so the relative performance of our algorithms should be similar to the performance of materialisation shown in Table 1.

Discussion Mat-Mod significantly outperformed Mat on all test inputs. For example, Mat-Mod was several times faster than Mat on Claros-LE and LUBM-LE. The programs of both benchmarks contain transitivity and symmetric–transitivity modules, which are efficiently handled by our custom algorithm. The performance improvement is even more significant for DBpedia-SKOS and DAG-R: Mat-Mod is more than 30 times faster than Mat on DBpedia-SKOS, and the difference reaches two orders of magnitude on DAG-R. In fact, DBpedia contains long chains/cycles over the *skos:broader* relation (Bishop et al. 2011b), which is axiomatised as transitive in SKOS. Mat-Mod outperforms Mat in this case since our custom algorithm for transitivity skips a large number of rule instances. The same observation explains the superior performance of DAG-R.

Similarly, DRed^c-Mod considerably outperformed DRed^c on small deletions: the performance speedup ranges from around ten times on LUBM-LE to three orders of magnitude on Claros-LE. The program of Claros-LE contains a symmetric–transitive closure module for the predicate *relatedPlaces*, and the materialisation contains large cliques of constants connected to each other via this predicate. Thus, when a *relatedPlaces(a,b)* fact is deleted, DRed^c can end up considering up to n^3 rule instances where n is the number of constants in the clique containing a and b. In contrast, our custom algorithm for this module maintains a connected component for the clique and requires only up to n^2 steps. It is worth noticing that, while DRed^c-Mod significantly outperforms DRed^c on DAG-R, the incremental update times for small deletion were larger than both the update times for large deletions and even for the initial materialisation. This is because deleting one thousand edges from the graph ('Small Deletion') caused a large part of the materialisation to be overdeleted and rederived again. In contrast, when 25% of the explicit facts are deleted ('Large Deletion'), a larger propertion of the materialisation is overdeleted, but only a few facts are rederived. For DRed^c the situation is similar, but rederivation in DRed^c benefits from a global recursive counter (at the expense of considering each applicable rule instance), which makes small deletion still faster than large deletion and initial materialisation. Finally, as shown in Table 2, DRed^c-Mod scaled well and maintained its advantage over DRed^c on large deletions.

Incremental insertions are in general easier to handle than

deletions since during insertion the algorithms can rely on the whole materialisation to prune the propagation of facts whereas during deletion the algorithms can only rely on the nonrecursive counters of facts to do the same. This is clearly reflected in Table 2. Nevertheless, in our tests for small insertions, DRed^c-Mod was several times faster than DRed^c in all cases but LUBM-LE, for which both algorithms updated the materialisation instantateously.

8 Conclusion

We have proposed a modular framework for the computation and maintenance of datalog materialisations. The framework allows integrating custom algorithms for specific types of rules with standard datalog reasoning methods. Moreover, we have presented such custom algorithms for programs axiomatising the transitive and the symmetric–transitive closure of a relation. Finally, we have shown empirically that our algorithms typically significantly outperform then existing ones, sometimes by orders of magnitude. In future, we plan to extend our framework also to the B/F^c algorithm, which eliminates overdeletion by eagerly checking alternative derivations. This could potentially be useful in cases such as DBpedia-SKOS and DAG-R, where overdeletion is a major source of inefficiency.

Acknowledgements

We thank David Tena Cucala for his help with obtaining the SKOS benchmark. This work was supported by the EPSRC projects AnaLOG, DBOnto, and ED^3.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Bishop, B.; Kiryakov, A.; Ognyanoff, D.; Peikov, I.; Tashev, Z.; and Velkov, R. 2011a. OWLIM: A family of scalable semantic repositories. *SWJ* 2(1):33–42.

Bishop, B.; Kiryakov, A.; Ognyanov, D.; Peikov, I.; Tashev, Z.; and Velkov, R. 2011b. Factforge: A fast track to the web of data. *Semantic Web* 2(2):157–166.

Demetrescu, C., and Italiano, G. F. 2000. Fully dynamic transitive closure: breaking through the o (n/sup 2/) barrier. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, 381–389. IEEE.

Dong, G.; Su, J.; and Topor, R. W. 1995. Nonrecursive Incremental Evaluation of Datalog Queries. *Annals of Mathematics and Artificial Intelligence* 14(2–4):187–223.

Guo, Y.; Pan, Z.; and Heflin, J. 2005. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(2-3):158–182.

Gupta, A.; Mumick, I. S.; and Subrahmanian, V. S. 1993. Maintaining Views Incrementally. In *SIGMOD*. ACM.

Horrocks, I.; Patel-Schneider, P. F.; Boley, H.; Tabet, S.; Grosof, B.; Dean, M.; et al. 2004. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission.

Hu, P.; Motik, B.; and Horrocks, I. 2018a. Modular Materialisation of Datalog Programs. *CoRR* abs/1811.02304.

Hu, P.; Motik, B.; and Horrocks, I. 2018b. Optimised maintenance of datalog materialisations. In *AAAI*, 1871–1879.

Ibaraki, T., and Katoh, N. 1983. On-line computation of transitive closures of graphs. *Information Processing Letters* 16(2):95–97.

King, V. 1999. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, 81–89. IEEE.

La Poutre, J. A., and van Leeuwen, J. 1987. Maintenance of transitive closures and transitive reductions of graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, 106–120. Springer.

Lehmann, J.; Isele, R.; Jakob, M.; Jentzsch, A.; Kontokostas, D.; Mendes, P. N.; Hellmann, S.; Morsey, M.; Van Kleef, P.; Auer, S.; et al. 2015. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web* 6(2):167–195.

Miles, A., and Bechhofer, S. 2009. Skos simple knowledge organization system reference. *W3C recommendation* 18:W3C.

Motik, B.; Patel-Schneider, P.; Parsia, B.; Bock, C.; Fokoue, A.; Haase, P.; Hoekstra, R.; Horrocks, I.; Ruttenberg, A.; Sattler, U.; et al. 2009. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C.

Motik, B.; Nenov, Y.; Piro, R.; Horrocks, I.; and Olteanu, D. 2014. Parallel materialisation of datalog programs in centralised, main-memory rdf systems. In *AAAI*, 129–137.

Motik, B.; Nenov, Y.; Piro, R.; and Horrocks, I. 2015. Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm. In *AAAI*, 1560–1568.

Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2015. RDFox: A Highly-Scalable RDF Store. In *ISWC*, 3–20.

Staudt, M., and Jarke, M. 1996. Incremental Maintenance of Externally Materialized Views. In *VLDB*, 75–86.

Subercaze, J.; Gravier, C.; Chevalier, J.; and Laforest, F. 2016. Inferray: fast in-memory RDF inference. *PVLDB* 9(6):468–479.

Urbani, J.; Kotoulas, S.; Maassen, J.; Van Harmelen, F.; and Bal, H. 2012. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *JWS* 10:59–75.

Urbani, J.; Jacobs, C. J.; and Krötzsch, M. 2016. Column-Oriented Datalog Materialization for Large Knowledge Graphs. In *AAAI*, 258–264.

Wu, Z.; Eadon, G.; Das, S.; Chong, E. I.; Kolovski, V.; Annamalai, M.; and Srinivasan, J. 2008. Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In *ICDE*, 1239–1248. IEEE.