



Certified Graph View Maintenance with Regular Datalog

Angela Bonifati, Stefania Dumbrava, Emilio Jesús Gallego Arias

► To cite this version:

Angela Bonifati, Stefania Dumbrava, Emilio Jesús Gallego Arias. Certified Graph View Maintenance with Regular Datalog. Theory and Practice of Logic Programming, 2018, 34th International Conference on Logic Programming, 18 (3-4), pp.372-389. 10.1017/S1471068418000224 . hal-01932818

HAL Id: hal-01932818

<https://hal.science/hal-01932818>

Submitted on 23 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Certified Graph View Maintenance with Regular Datalog

Angela Bonifati, Stefania Dumbrava

LIRIS, Université Lyon 1, France

Emilio Jesús Gallego Arias

MINES ParisTech, PSL Research University, France

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We employ the Coq proof assistant to develop a mechanically-certified framework for evaluating graph queries and incrementally maintaining materialized graph instances, also called views. The language we use for defining queries and views is Regular Datalog (RD) – a notable fragment of non-recursive Datalog that can express complex navigational queries, with transitive closure as native operator. We first design and encode the theory of RD and then mechanize a RD-specific evaluation algorithm capable of fine-grained, incremental graph view computation, which we prove sound with respect to the declarative RD semantics. By using the Coq extraction mechanism, we test an OCaml version of the verified engine on a set of preliminary benchmarks. Our development is particularly focused on leveraging existing verification and notational techniques to: a) define mechanized properties that can be easily understood by logicians and database researchers and b) attain formal verification with limited effort. Our work is the first step towards a unified, machine-verified, formal framework for dynamic graph query languages and their evaluation engines.

KEYWORDS: Regular Datalog, Graph Queries, Graph Views, Incremental Maintenance, Finite Semantics, Theorem Proving

1 Introduction

Modern graph query engines¹ are gaining momentum, due to the proliferation of *interconnected* data and related applications, spanning from social networks to scientific databases and the Semantic Web. The adopted query languages are *navigational*, focusing on data topology and, particularly, on *label-constrained reachability*. Notable examples (Angles et al. 2017) include Neo4j’s openCypher (Cypher), Facebook’s GraphQL (GraphQL), SPARQL (SPARQL), Gremlin (Gremlin), Oracle’s PGX (Oracle PGX), and the recent G-CORE (G-Core), even though a standard graph query language is still undefined.

At a foundational level, these languages are based on *conjunctive queries* (CQ), corresponding to Datalog (Ceri et al. 1989) clauses, i.e., function-free Horn formulas. Their common denominator is that they perform *edge traversals* (through join chains), while

¹ Several successful commercial implementations exist, e.g., Neo4j (Neo4j), Google’s Cayley (Cayley), Twitter’s FlockDB (FlockDB) and Facebook’s Apache Giraph (Giraph).

specifying and testing for the existence of label-constrained paths. Recently, a solution to the long standing open problem of identifying a suitable graph query Datalog fragment, balancing expressivity and tractability, has been proposed in (Reutter et al. 2017). We call this *Regular Datalog* (RD), a binary linear Datalog subclass that allows for complex, regular expression patterns between nodes. RD provides additional properties over full Datalog. First, its evaluation has NLOGSPACE-complete complexity, belonging to the NC class of *highly parallelizable* problems. Second, RD query containment is *decidable*, with an elementary tight bound (2EXPSPACE-complete) (Reutter et al. 2017).

To the best of our knowledge, no *specific evaluation algorithm* for RD queries has been proposed yet. What are the main desiderata that such an algorithm should have? On the one hand, real-world graphs are highly dynamic, ever-increasing in size, and ever-changing in content. Hence, *efficiency*, i.e., accelerating graph-based query processing, has become more relevant than ever. On the other hand, the sensitive nature of the data that large-scale, commercial graph database engines operate on makes ensuring *strong reliability guarantees* paramount.

We argue that having a deeply-specified framework for performing *incremental graph computation*, based on logic programming, opens important perspectives in a variety of areas, especially in light of security-sensitive applications involving graph-shaped topologies, e.g., financial transaction, forensic analysis, and network communication protocols.

Problem Statement: We target both requisites by developing a **mechanically-verified** framework for the *fine-grained incremental view maintenance of graph databases* (**IVMGD**). Specifically, let \mathcal{G} be a graph database instance, $\Delta\mathcal{G}$, a set of modifications, and $V[\mathcal{G}]$, the materialization of an RD view *or query* over \mathcal{G} . We provide an IVMGD-aware engine that computes an *incremental view update* $\Delta V[\mathcal{G}; \Delta\mathcal{G}]$, making the resulting view consistent with the updated graph database, i.e., $V[\mathcal{G}] \cup \Delta V[\mathcal{G}; \Delta\mathcal{G}] = V[\mathcal{G} \cup \Delta\mathcal{G}]$.

Contributions: We build our mechanically-certified engine using *theorem proving* techniques, in particular we use the Coq proof assistant (The Coq Development Team 2018) to develop both the theory of the RD language and the evaluation engine itself. We make three major contributions: a) we *formalize*, in Coq, the syntax and semantics of Regular Queries, as Regular Datalog programs; b) we *implement*, in Coq’s functional programming language Gallina, an executable engine for incremental RD maintenance; c) we *prove* the engine is *sound*, i.e. that it correctly computes incremental view updates.

We encode the semantics of RD using the finite set theory in the Mathematical Components library, which was developed to carry out the mechanized proof of the Feit-Thompson theorem (Gonthier et al. 2013) on finite group classification; it thus provides excellent support for finite reasoning. This brings the actual written-down mechanized semantics very close to a mathematical language, making it more accessible to non-expert practitioners – a central point to understand the guarantees provided by our development.

To develop our incremental graph view maintenance algorithm, we adapt the classical delta-rule one for IVM (Gupta et al. 1993) — initially designed for non-recursive SQL — to the RD evaluation setting.

Lastly, we prove our main result: “Let \mathcal{G} be a base instance and Π , a RD program of view V . If Π is satisfied by the materialized view $V[\mathcal{G}]$, then, for a \mathcal{G} update $\Delta\mathcal{G}$, the IVMGD engine outputs an *incremental view update*, $\Delta V[\mathcal{G}; \Delta\mathcal{G}]$, such that Π is satisfied

by $V[\mathcal{G}] \cup \Delta V[\mathcal{G}; \Delta \mathcal{G}]$. The proof relies on two key, newly-developed mechanized theories for *stratified* and *incremental* satisfaction of Datalog programs.

As mentioned in (Fan et al. 2017), theoretical work on graph view maintenance is still in its infancy and, as noted in (Beyhl and Giese 2016), most mainstream commercial graph databases do not provide concepts for defining graph views and maintenance.² Thus, we believe that our verified engine builds the foundations for certifying realistic graph query and update engines, using declarative paradigms, such as Datalog subsets, as their query language. Specifically, this certified specification could serve as a blueprint for future graph query design and ongoing standardisation efforts (G-Core). Additionally, we consider that most of our verification techniques are not restricted to the Regular Datalog setting, but are also applicable to broader logic programming contexts.

Organization: The paper is organized as follows. In Section 2, we illustrate the syntax and semantics of RD and its Coq formalization. In Section 3, we present the IVMGD algorithm and, in Section 4, we summarize its mechanized proof. Section 5 shows our extracted engine’s performance on graph datasets with synthetic queries in RD. We describe related work in Section 6 and conclude and outline perspectives in Section 7. The Coq code for this paper can be downloaded from: <https://github.com/VerDILog/VerDILog/tree/iclp-2018>.

2 Regular Datalog: Design and Formalization

In this section, we present the theory of *Regular Datalog* (RD) and its Coq mechanization. The language is based on Regular Queries (RQs) (Reutter et al. 2017). In Sec. 2.1 and Sec. 2.2, we detail our encoding of RD syntax and semantics. Sec. 2.3 illustrates potential usages of the language, in the context of financial transaction and social network graphs.



2.1 RD Syntax

We fix the finite sets of constants (nodes) and symbols (edge labels), namely, \mathbf{V} and Σ .

Definition 1 (Graph Database)

A *graph instance* \mathcal{G} over Σ is a set of *directed* labelled edges, \mathbf{E} , where $\mathbf{E} \subseteq \mathbf{V} \times \Sigma \times \mathbf{V}$. A *path* ρ of length k in \mathcal{G} is a sequence $n_1 \xrightarrow{s_1} n_2 \dots n_{k-1} \xrightarrow{s_k} n_k$. Its label is the concatenation of edge symbols, i.e., $\lambda(\rho) = s_1 \dots s_k \in \Sigma^*$.

\mathcal{G} can be seen as a database $\mathcal{D}(\mathcal{G}) = \{s(n_1, n_2) \mid (n_1, s, n_2) \in \mathbf{E}\}$, by interpreting its edges as binary *relations* between nodes. $\mathcal{D}(\mathcal{G})$ is also called the *extensional database* (EDB).

In order to model graphs in Coq, we assume a pair of *finite types* (`finType`), representing edge labels and nodes. The graph encoding (`egraph` ) is, thus, a *finitely supported function* (`lrel` ) , mapping labels to their corresponding set of edges:

```
Variables (V Σ : finType).
Inductive L := Single | Plus.
Inductive egraph := EGraph of {set V * V}.
Inductive lrel    := LRel of {ffun Σ * L -> egraph}
```

² The only recent exception – which can handle named queries and updates – is Cypher for Apache Spark <https://github.com/opencypher/cypher-for-apache-spark>.

t	$::= n \in \mathbf{V} \mid x \in \mathcal{V}$	(Terms, Node ids)
A	$::= s(t_1, t_2)$, where $s \in \Sigma \mid t_1 = t_2$	(Atoms),
L	$::= A \mid A^+$	(Literals)
B	$::= L_1 \wedge \dots \wedge L_n$	(Conjunctive Body)
D	$::= B_1 \vee \dots \vee B_n$	(Disjunctive Body)
C	$::= (t_1, t_2) \leftarrow D$	(Clause)
Π	$::= \Sigma \rightarrow \{C_1, \dots, C_n\}$	(Program)

Fig. 1: Regular Datalog Grammar

Note that, for each label, our graph representation maintains both the regular set of edges and its transitive closure, denoted by the L type. We explain its usage in later sections.

Definition 2 (Regular Datalog (RD))

Regular Datalog is the binary — all atoms have arity 2 — Datalog fragment, with *recursion restricted to transitive closure* and *internalized* as labels on literals.

There are several approaches to making the representation of logic programs amenable to efficient mechanical reasoning. We have found that indexing *completed* clauses by *head* symbols works well, as the corresponding canonical disjunctive definitions, from the clause map, become readily accessible in proofs. Fig. 1 provides the formal syntax for Regular Datalog programs. A program is a map from each symbol in Σ , to a *single* clause head and *normalized* disjunctive body. The normalized form is obtained by first transforming all clauses to a common head representation and then grouping their respective bodies. This classical process is similar to the completion procedure in (Clark 1977). For example, the program: $s(a, b). s(z, y) \leftarrow p(x, y), q^+(z, x)$ is normalized as $s(x, y) \leftarrow (a = x \wedge b = y) \vee (p(z, y) \wedge q^+(x, z))$ and represented by a *function* from s to the head and disjunctive body. We encode RD primitives in Coq as records:

```

Record atom    := Atom    { syma :  $\Sigma$ ; arga :  $T * T$  }.
Record lit     := Lit     { tagl :  $L$ ; atoml : atom }.
Record cbody   := CBody   { litb : seq lit }.
Record clause  := Clause  { headc :  $T * T$ ; bodyc : seq cbody }.
Inductive program := Program of {ffun  $\Sigma \rightarrow$  clause  $T \Sigma L$ }.

```

A key feature of this formalization is that it is parametric in the variables Σ, T, L . This design choice allows sharing the representation for ground and non-ground clauses, and is central to the incrementality proof, in which we will decorate literals with customs labels. Also, note our naming convention for Coq constants, whereby the last letter denotes the type. For example, `syma` 🐦 is the function that returns the symbol for an atom. `syml` 🐦 does the same for a literal. Similarly, satisfaction conditions will be named `sTa`, `sTl`, etc., depending on the argument type.

Definition 3 (Regular Queries (RQ))

A *regular query* Ω over \mathcal{G} is a RD program Π , together with a distinguished query clause, whose head is the top-level *view* (V) and whose body is a conjunction of Π literals.

2.2 RD Semantics

The semantics of RD programs follows a standard term-model definition. As noted in Sec. 2.1, for optimization purposes, interpretations \mathcal{G} are modeled as *indexed relations*

$(\Sigma \times \{\emptyset, +\}) \rightarrow \mathcal{P}(\mathbf{V} \times \mathbf{V})$, containing labeled graphs and their transitive closure. Then, program satisfaction builds on the below definition of satisfaction for (ground) literals:

Definition 4 (Literal Satisfaction)

The satisfaction $\mathcal{G} \models L$ of a ground literal $L = s^l(n_1, n_2)$ is defined as:

$$\mathcal{G} \models s^l(n_1, n_2) \iff (n_1, n_2) \in \mathcal{G}(s, l)$$

Note that, in order for this definition to be correct, \mathcal{G} must be *well-formed*, that is to say, the information stored in $\mathcal{G}(s, +)$ has to correspond to the actual transitive closure of $\mathcal{G}(s, \emptyset)$. We can state this condition as:

$$\begin{aligned} \text{wfG}(\mathcal{G}) &\iff \forall s, \text{is_closure}(\mathcal{G}(s, \emptyset), \mathcal{G}(s, +)) \\ \text{is_closure}(g_s, g_p) &\iff \forall (n_1, n_2) \in g_p, \exists \rho \in \mathbf{V}^+, \text{path}(g_s, n_1, \rho) \wedge \text{last}(\rho) = n_2 \\ \text{path}(g, n_1, \rho) &\iff \forall i \in \{1 \dots |\rho|\}, (n_i, n_{i+1}) \in g \end{aligned}$$

where the node list $\rho \in \mathbf{V}^+$ represents the path without including the initial node n_1 .

Note that we compile the surface syntax $s^-(X, Y)$ to $s(Y, X)$ and $s^*(X, Y)$ to $X = Y \vee s^+(X, Y)$. The Coq encoding `sTl gl` of $\mathcal{G} \models L$ is a direct transcription of Def. 4, where we henceforth omit the \mathbf{G} parameter, as is it assumed to be implicit:

Definition `sTl gl := G (syml gl, tagl gl) (argl gl).1 (argl gl).2.`

Definition 5 (Clause Satisfaction)

A RD clause with disjunctive body $D \equiv (L_{1,1} \wedge \dots \wedge L_{1,n}) \vee \dots \vee (L_{m,1} \wedge \dots \wedge L_{m,n})$ and head symbol s is satisfied by \mathcal{G} iff, for all groundings η , whenever the corresponding instantiation of a body in D is satisfied, then the head is also satisfied. Formally:

$$\begin{aligned} \mathcal{G} \models_s (t_1, t_2) \leftarrow (L_{1,1} \wedge \dots \wedge L_{1,n}) \vee \dots \vee (L_{m,1} \wedge \dots \wedge L_{m,n}) &\iff \\ \forall \eta, \bigvee_{i=1..m} (\bigwedge_{j=1..n} \mathcal{G} \models \eta(L_{i,j})) \Rightarrow \eta(s(t_1, t_2)) & \end{aligned}$$

The Coq encoding of Def. 5 relies on the definition of an instantiation with a *grounding* η . We model η as a function \mathbf{g} , of type `gr`, mapping from \mathcal{V} (the ordinal type `'I_n`) to \mathbf{V} . This extends straightforwardly from terms to clauses.

Definition `gr n := {ffun 'I_n -> V}.`

Definition `sTb b := all sTl (litb b).`

Definition `sTc n s c := [forall g : gr n, let gc := grc g c in
has sTb (bodyc gc) ==> G (s, Single) (headc gc)].`

Note that `has` and `all` are the counterparts of the corresponding logical operations, extended to lists. We can define a model for an RD program as:

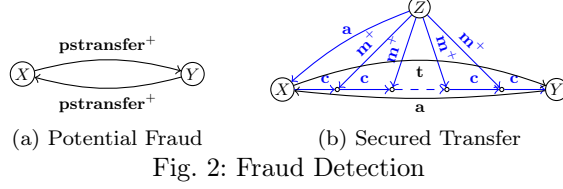
Definition 6 (Program Satisfaction)

A *well-formed* interpretation \mathcal{G} is a model for a program Π with respect to Σ iff \mathcal{G} satisfies all of the Π clauses indexed by symbols in Σ : $\mathcal{G} \models_\Sigma \Pi \iff \forall s \in \Sigma, \mathcal{G} \models_s \Pi(s)$.

The Coq encoding is straightforward:

Definition `sTp (p : program n) := [forall s, sTc n s (p s)].`

The formalized definition of satisfaction in Def. 6 is crucial to understanding the main soundness theorem in Sec. 4. It establishes that the output of the mechanized engine is a *model* of the input program, i.e., that it complies to the satisfaction specification of Def. 6 above. Hence, if this definition were to be incorrect — for example, by making `sTp p = true` — the theorem would become meaningless.



2.3 RD Examples

We write $(r + s)(x, y)$ for $r(x, y) \vee s(x, y)$ and $(r \cdot s)(x, y)$ for $r(x, z) \wedge s(z, y)$.

Example 1 (Fraud Detection)

Consider a financial transaction network, in which entities can **connect** and **transfer** money to each other, as well as **monitor** and **accredit** each other. A potential fraud/*suspect transaction*, e.g., money laundering, is a cycle of $\mathbf{pstransfer}^+$, i.e., potentially secured transfer chains (Fig. 2). A potentially secured transfer ($\mathbf{pstransfer}$) is either a transfer (**t**) or a *secured transfer*. A *secured transfer* from X to Y occurs if Y is accredited (**a**) by X and if X **secures** a connection and transfers (**t**) to Y . X **secures** a connection to Y (blue subgraph in Fig. 2), if it connects (**c**) via a chain of intermediaries that are centrally monitored (**m**⁺) by an accredited (**a**) entity Z . Potentially fraudulent (*suspect*) transactions can be computed with the RD program:

```

suspect(X, Y)   ← pstransfer+(X, Y), pstransfer+(Y, X)
pstransfer(X, Y) ← (transfer + transfer)(X, Y)
transfer(X, Y)  ← accredited(Y, X), secures(X, Y), transfers(X, Y)
secures(X, Y)   ← (connected · cmonitored+ · connected)(X, Y)
cmonitored(X, Y) ← connected(X, Y), monitors+(Z, X), monitors+(Z, Y), accredited(Z, X)

```

Example 2 (Brand Reach)

Consider a platform, such as Twitter, with asymmetric connections and, thus, an underlying directed graph topology. Let Z be a brand (central node in Fig. 3) that wants to determine its *reach*, i.e. *potential clients* pairs (empty nodes in Fig. 3). A pair of users (X, Y) are *potential clients*, if both were *exposed* to Z . We say X is *exposed* to Z , if X *endorses* Z or if it is connected, through a potential chain of followers, to an influencer that *endorses* Z . We say a user *endorses* a brand, if it *likes* and *advertises* the brand.

```

reach(X, Y)      ← (pclients + pclients-)(X, Y)
pclients(X, Y)   ← exposed(X, Z), exposed(Y, Z)
exposed(X, Z)    ← (follows* · endorses)(X, Z)
endorses(X, Z)   ← likes(X, Z), advertises+(X, Z)

```

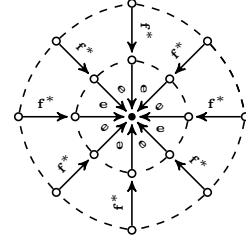


Fig. 3: Brand Reach

3 Regular Datalog Evaluation: A Mechanized IVMGD-aware Engine

We now describe the design of the mechanized IVMGD engine, which is based on the non-recursive bottom-up evaluation of RD programs. We first describe the top-level interface

of the engine and the main execution loop. Then, in Sec. 3.2, we outline the building block of *non-incremental* clause evaluation; in Sec. 3.3, we describe the delta-join *incremental clause evaluation* algorithm. Finally, Sec. 3.4 explains the implementation of the delta-join heuristic in the actual engine.


3.1 Top-level Interface and Overview

The incremental RD evaluation engine is designed around bottom-up non-recursive program model computation. This is indeed adequate for graph and regular queries as these internalize recursion using *closure operations* — usually computed by specialized tools optimized for efficiency. While our engine builds on ideas from (Benzaken et al. 2017), we have considerably redesigned all its core components: syntax is now based on a new *parametric* and *normalized* representation, and the core evaluation infrastructure and theory have been redesigned to account for *stratified*, *single-pass*, non-recursive *incremental* model computation. The formalization workload is higher in our setting, as we cannot rely on the usual fixpoint theorems, but must define a custom theory for modular program satisfaction.

A key problem to solve when reasoning about *incremental* computation is the representation of changes. Indeed, a formal definition of updates can be delicate to state, as it must account for potentially overlapping additions and removals, order issues, etc. To this end, we define *canonical* graph updates, reminiscent of, but weaker than “change structures” — defined in (Cai et al. 2014):

Definition 7 (Graph Updates)

An *update* $\Delta \equiv (\Delta_+, \Delta_-)$ is a pair of *disjoint* graphs, respectively representing insertions and deletions.

In Coq, updates are encoded as a record `edelta`  packing the graphs and a disjointness proof; this allows us to consider only well-formed updates. Note that `add` and `deld` are the record fields representing Δ_+ and Δ_- :

Definition `wf_edelta` `add deld` := `[forall s, [disjoint add s & deld s]]`.
Structure `edelta` := `{ add : lrel; deld : lrel; _ : wf_edelta add deld }`.

The core operations for updates are their *application* to a base graph, and the *modification* of the changes pertaining to a particular symbol:

Definition 8 (Update application and modification)

$$\begin{aligned} \mathcal{G} \text{ :+ } \Delta &\equiv \mathcal{G} \setminus \Delta_- \cup \Delta_+ && (\text{application}) \\ \Delta \{s \rightarrow (g_+, g_-)\} &\equiv (\Delta_+ \{s \rightarrow g_+\}, \Delta_- \{s \rightarrow g_- \setminus g_+\}) && (\text{modification}) \end{aligned}$$

Armed with these definitions, plus those corresponding to the RD syntax and semantics from Sec. 2, we can now define the top-level interface to our engine. A particular challenge that arose during the development of the interface was allowing repeated, incremental Δ -aware calls. Achieving composition proved to be quite challenging, as the soundness invariant must be preserved along calls. In total, six parameters had to be used. The *static* input parameters are: a program Π , a graph \mathcal{G} , and a set of symbols, or *support* `supp`, which indicates the validity of a subset of \mathcal{G} , and thus what information

the incremental engine may not recompute. Indeed, a precondition of the engine is that the input graph must be a model of Π up to **supp**, that is to say, $\mathcal{G} \models_{\text{supp}} \Pi$. Note that in the database literature, Σ is usually seen as a disjoint set pair, (Σ_E, Σ_I) , corresponding to the *extensional* and *intensional* parts of a program. For our engine, this distinction is “dynamic”, in the sense that an already-processed strata-level is seen as “extensional”, or immutable, for the rest of the execution. Thus, typical cases for **supp** will be $\text{supp} \equiv \Sigma_E$, when the engine has been never been run before, or $\text{supp} \equiv \Sigma$, where \mathcal{G} is the output of a previous run, and thus the consequences for all clauses have been computed.

Whereas the program, graph, and support are fixed during the execution of the engine, the latter will take an additional three *dynamic* parameters, representing the current execution state. These are: Δ , the current update, which is modified at each iteration, and $\Sigma_{\triangleright}, \Sigma_{\triangleleft}$, which respectively represent the set of processed symbols/stratum and the “todo” list. We write $\Delta_O = T_{\mathcal{G}, \text{supp}}^{\Pi}(\Sigma_{\triangleright}, \Sigma_{\triangleleft}, \Delta)$ (written **fwd_program** \heartsuit in Coq) for a call to the engine returning an update Δ_O . We prove that the engine implements a *program consequence algorithm*, thus satisfying $\mathcal{G} :+ \Delta_O \models_{\Sigma} \Pi$. Assuming a *clause consequence operator* $T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta)$ (or **fwd_or_clause** \heartsuit), the below Coq implementation of the engine iterates over the unprocessed symbol list Σ_{\triangleleft} and, for each of its symbols s to be inspected, it computes a new update Δ' . To this end, it modifies Δ with the consequences for the clause indexed by s and computes the corresponding closure of s . The algorithm then makes the recursive call, adding s and $s+$ to the set of processed symbols.

```

Fixpoint fwd_program  $\Pi$   $G$  supp  $\Delta$   $\Sigma_{\triangleright}$   $\Sigma_{\triangleleft}$  : edelta := match  $\Sigma_{\triangleleft}$  with
| [::] =>  $\Delta$ 
| [::  $s$  &  $ss$ ] =>
  let (arg, body) :=  $\Pi$   $s$  in
  let  $\Delta'$  := fwd_or_clause  $G$  supp  $\Delta$   $s$  arg body in
  let  $\Delta'$  := compute_closures  $G$   $\Delta'$   $s$  in
  fwd_program  $\Pi$   $G$  supp  $\Delta'$  ( $s \cup s+ \cup \Sigma_{\triangleright}$ )  $ss$ 

```

Note that the **compute_closures** function above is an abstraction over an arbitrary algorithm for closure computation, that we assume correct. For instance, we hope to use the verified implementation of Tarjan’s algorithm from (Cohen and Théry 2017).

As can be observed from the above code, the core part of the algorithm is concentrated in clause evaluation. Hence, we now proceed to presenting *base* and *incremental* for clause evaluation. Note that the base — or *non-incremental* — method is still needed since in some cases, incremental evaluation is either not possible or not sensible, as full re-computation may be faster.

3.2 The Base Engine for RD Evaluation

The *base*, or *non-incremental*, clause evaluation implements a forward-chaining *consequence operator*, using a *matching algorithm* M ; this takes as input atoms, literals, or clause bodies and returns a set of substitutions, as explained in (Abiteboul et al. 1995). Basic literal matching, $M_G^L(l)$ (**match_lit** \heartsuit in Coq) takes a literal and returns the set of all substitutions ss that satisfy it, so that $\forall \sigma \in ss, \mathcal{G} \models \sigma(l)$. Literal matching is extended to body matching in a straightforward way, with $M_G^B(B)$ (**match_body** \heartsuit) traversing B and accumulating the set of substitutions obtained from the individual matching. The algorithm we consider corresponds to computing *nested-loop* join and we implement it

in a functional style, using a monadic fold. Substitutions are then accumulated for each disjunctive clause and *grounded* heads are added to the interpretation:

Definition 9 (Clausal Consequence Operator)

Given a RD clause $\Pi(s) \equiv (t_1, t_2) \leftarrow \bigvee_{i=1..n} B_i$, the *base clausal consequence operator* $T^{\Pi,s}(\mathcal{G})$ computes the set of facts that can be *inferred* from \mathcal{G} :

$$T^{\Pi,s}(\mathcal{G}) \equiv \{\sigma(t_1, t_2) \mid \sigma \in \bigcup_{i=1..n} M_{\mathcal{G}}^B(B_i)\}.$$

The Coq version \heartsuit is almost a direct transcription:

```
Definition fwd_or_clause_base G Δ s c : edelta := let GΔ := edb :+: Δ in
  let T = [set gr σ c.headc | σ in \bigcup_(b <- c.bodyc) match_body G b ] in
  Δ{s → GΔ s ⊖ T}
```

in the first line, we use a set comprehension to build the set of ground facts corresponding to the consequence operator. Note that $\backslash\text{bigcup_}(x \leftarrow X)$ is the Coq notation for $\bigcup_{x \in X}$ and that $\text{gr } \sigma \text{ head}$ denotes the application of the substitution σ to the input head. The second line updates the resulting Δ , using the operator for *modification* (see Def. 8) and that for *graph difference* (\ominus). We remark that this base operator will re-derive all the facts, as it does not make use of incrementality information.

3.3 Incremental Delta-Join Maintenance

Given a graph \mathcal{G} , a program Π , and updates Δ , the engine in Sec. 3.2 *non-incrementally* maintains the top-level view of Π . However, the engine is unable to *reuse* and *adjust* previously computed maintenance information. This makes it especially inefficient when few nodes are added to an otherwise high-cardinality graph.

To remedy this situation, we would like to extend our algorithm so that it can take into account the information of previously computed models. The key idea is to restrict *matching* to graph updates. For example, let V be a materialized view, defined as a simple *join*, in our case, as the path over two base edges, r and s , i.e. $V(X, Y) \leftarrow r(X, Z), s(Z, Y)$. We abbreviate this as $V = r \bowtie s$. Given *base deltas*, r^Δ and s^Δ , we can compute the *view delta* as $\Delta V = (r^\Delta \bowtie s) \cup (r \bowtie s^\Delta) \cup (r^\Delta \bowtie s^\Delta)$, or, after factoring, as $V^\Delta = (r^\Delta \bowtie s) \cup (r^\nu \bowtie s^\Delta)$, where $r^\nu = r \cup r^\Delta$. Hence, $V^\Delta = V_1^\Delta \cup V_2^\Delta$, with V_1^Δ and V_2^Δ computable via the *delta clauses*: $\delta_1 : V_1^\Delta \leftarrow r^\Delta(X, Z), s^\Delta(Z, Y)$ and $\delta_2 : V_2^\Delta \leftarrow r^\nu(X, Z), s^\Delta(Z, Y)$.

Generalizing, for a database \mathcal{G} and a purely additive update Δ , we can determine the *view delta* $V^\Delta[\mathcal{G}; \Delta]$, i.e., the set of facts such that $V[\mathcal{G} \text{ :+ } \Delta] = V[\mathcal{G}] \cup V^\Delta[\mathcal{G}; \Delta]$.

Definition 10 (Delta Program)

Let V be a view defined by $V \leftarrow L_1, \dots, L_n$. The *delta program* $\delta(V)$ is $\{\delta_i \mid i \in [1, n]\}$. Each *delta clause* δ_i has the form $V \leftarrow L_1, \dots, L_{i-1}, L_i^\Delta, L_{i+1}^\nu, \dots, L_n^\nu$, where: L_j^ν marks that we match L_j against atoms in $\mathcal{G} \cup \Delta\mathcal{G}$ with the same symbol as L_j and L_j^Δ marks that we match L_j against atoms in $\Delta\mathcal{G}$ with the same symbol as L_j .

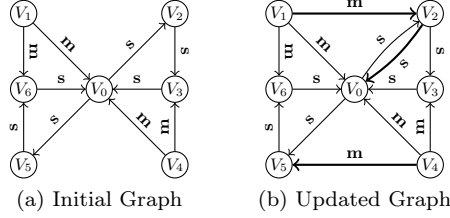
We revisit Example 1 to illustrate the computation of incremental view updates.

Example 3 (Detectable Frauds)

Consider the transaction configuration in Fig. 4a, marking suspect transactions. A suspect transaction between X and Y is *detectable*, if there exists an entity that monitors both X and Y . We can compute all detectable suspect transactions with the RQ:

$$\text{detectable}(X, Y) \leftarrow \text{suspect}(X, Y), \text{monitors}(Z, X), \text{monitors}(Z, Y)$$

These are $\text{detectable} = \{(V_6, V_0), (V_3, V_0)\}$. When *updating* the previous graph to the one in Figure 4b, we have: $\text{detectable}^\nu = \{(V_6, V_0), (V_3, V_0), (\mathbf{V}_0, \mathbf{V}_2), (\mathbf{V}_2, \mathbf{V}_0), (\mathbf{V}_0, \mathbf{V}_5)\}$. The delta update $\text{detectable}^\Delta = \{(\mathbf{V}_0, \mathbf{V}_2), (\mathbf{V}_2, \mathbf{V}_0), (\mathbf{V}_0, \mathbf{V}_5)\}$ can be *incrementally* computed with the program $\Pi_\Delta = \delta_1 \cup \delta_2 \cup \delta_3$, as follows: $\delta_1 = \emptyset$, $\delta_2 = \{(\mathbf{V}_2, \mathbf{V}_0)\}$, and $\delta_3 = \{(\mathbf{V}_0, \mathbf{V}_2), (\mathbf{V}_0, \mathbf{V}_5)\}$. Indeed, $\text{detectable}^\Delta = \text{detectable}^\nu \setminus \text{detectable}$.



$$\begin{aligned} \delta_1 : \quad & \text{detectable}^\Delta(X, Y) \leftarrow \text{suspect}^\Delta(X, Y), \text{monitors}(Z, X), \text{monitors}(Z, Y) \\ \delta_2 : \quad & \text{detectable}^\Delta(X, Y) \leftarrow \text{suspect}^\nu(X, Y), \text{monitors}^\Delta(Z, X), \text{monitors}(Z, Y) \\ \delta_3 : \quad & \text{detectable}^\Delta(X, Y) \leftarrow \text{suspect}^\nu(X, Y), \text{monitors}^\nu(Z, X), \text{monitors}^\Delta(Z, Y) \end{aligned}$$

(c) Delta Program for Detectable Frauds

Fig. 4: Detectable Frauds

3.4 The Δ -Engine for RD Evaluation and Incremental View Maintenance

We now present the incremental version of the clause evaluation operator defined in Sec. 3.2. We follow Sec. 3.3 and modify base matching to take into account Δ -clauses and programs. Thus, for each body to be processed incrementally, we generate a *body mask*, placing a tag — $\{\mathbf{B}, \mathbf{D}, \mathbf{F}\}$ — on each body literal, which indicates whether matching should proceed against the base interpretation, against the update, or against both. The *incremental atom matching* operator $M_{\mathcal{G}, \Delta}^{A, m}$ (`match_delta_atoms` 🐦) is defined as:

$$M_{\mathcal{G}, \Delta}^{A, m}(a) = (\text{if } m \in \{\mathbf{B}, \mathbf{F}\} \text{ then } M_{\mathcal{G}}^A(a) \text{ else } \emptyset) \cup (\text{if } m \in \{\mathbf{D}, \mathbf{F}\} \text{ then } M_{\Delta}^A(a) \text{ else } \emptyset)$$

thus, base matching is called with the instance corresponding to the atom's tag.

Body Δ -matching, $M_{\mathcal{G}, \Delta}^B$ (`match_delta_body` 🐦) takes as an input a *body mask*, that is to say, a list of tag-annotated literals. A function `body_mask` 🐦 generates the set B_Δ of “decorated” literals. Generic syntax is extremely helpful here to avoid duplication and to help state mask invariants in an elegant way. `body_mask` follows the *diagonal factoring* described below, where each row corresponds to an element of B_Δ :

$$\begin{bmatrix} L_1^{\mathbf{D}} & L_2^{\mathbf{F}} & \dots & L_{n-1}^{\mathbf{F}} & L_n^{\mathbf{F}} \\ L_1^{\mathbf{B}} & L_2^{\mathbf{D}} & \dots & L_{n-1}^{\mathbf{F}} & L_n^{\mathbf{F}} \\ \dots & \dots & \dots & \dots & \dots \\ L_1^{\mathbf{B}} & L_2^{\mathbf{B}} & \dots & L_{n-1}^{\mathbf{B}} & L_n^{\mathbf{D}} \end{bmatrix}$$

The last piece to complete the incremental engine is the top-level clausal maintenance operator. This part of the engine is significantly more complex than its *base* counterpart, as it must take into account which incrementality heuristics to apply:

Definition 11 (Incremental Clausal Maintenance Operator)

The $T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta)$ operator for incremental clausal maintenance (`fwd_or_clause_delta` 🐦) acts in two cases. If $s \notin \text{supp}$, or Δ contains deletions for any of the literals in the body of $\Pi(s)$, it uses the base operator $T^{\Pi, s}(\mathcal{G} :+ \Delta)$ — as we either cannot reuse the previous model or cannot support deletions through our incremental strategy. Otherwise, the operator will generate $B_\Delta = \text{body_mask}(B)$, for each of the bodies B , and return $\bigcup_{B_m \in B_\Delta} M_{\mathcal{G}, \Delta}^B(B_m)$.

4 Regular Datalog Evaluation: Certified Soundness

We now summarize the main technical points of the mechanized proof developed in Coq. A key to effective mechanized proof development is the definition of the proper high-level concepts and theories; unfortunately we lack the space here to describe all the definitions used in our mechanized development in detail, so we highlight the main result, that proves the soundness of the engine, and we briefly survey the two core theories for *stratification* and *incrementality*. A few auxiliary results are described in Appendix A.

4.1 Stratification Conditions

Definition 12 (Stratified Programs)

A key precondition for the *soundness* of our engine is program stratification. A program Π is *stratified*, if there exists a mapping $\sigma : \Sigma \rightarrow [1, n]$ such that, for all s in Σ , the $\Pi(s)$ clause $(t_1, t_2) \leftarrow B$ satisfies: $\max_{r \in \text{sym}(B)} \sigma(r) < \sigma(s)$, where `sym` returns the set of symbols occurring in B . We then call σ a *stratification* of Π . In Coq, we encode stratification using a list of symbols and a predicate `is_strata` 🐦 that recursively checks each uninspected symbol against an accumulator. This choice of representation is practical as it will guide model computation in the engine.

Definition 13 (Well-formed Program Slices)

In order to reason about stratified satisfaction, we need a strengthened notion of well-formedness stating that a program is closed w.r.t. a symbol set. A symbol set Σ is a *well-formed slice* of Π if, for all s in Σ , $\text{sym}(\Pi(s)) \subseteq \Sigma$.

We establish that the engine operates over well-formed slices, which allows us to isolate reasoning about the current iteration, see the appendix for more technical details.

4.2 IGVM-Engine Characterization Result

Let Π be a *safe* RD program, Σ_{\triangleright} and Σ_{\triangleleft} , symbol sets corresponding the “extensional” (already processed) and “intensional” (to be processed) strata, \mathcal{G} a graph instance, and Δ an update. Then we establish:

Theorem 1 (IGVM-Engine Soundness)

Assume (H1) \mathcal{G} is a model of the program for supp , $\mathcal{G} \models_{\text{supp}} \Pi$, (H2) Σ_{\triangleright} is a *well-formed slice* of Π , (H3) Δ only contains information up to the processed strata, $\text{sym}(\Delta) \subseteq \Sigma_{\triangleright}$, (H4) $\Sigma_{\triangleright}, \Sigma_{\triangleleft}$ are a *stratification* of Π , and (H5) the currently model is sound, $\mathcal{G} :+ : \Delta \models_{\Sigma_{\triangleright}} \Pi$ then, the engine — implementing the maintenance operator $T_{\mathcal{G}, \text{supp}}^{\Pi}(\Sigma_{\triangleright}, \Sigma_{\triangleleft}, \Delta)$ — outputs an update Δ_O , such that $\mathcal{G} :+ : \Delta_O \models_{\Sigma} \Pi$ holds.

Proof

The proof is a consequence of the soundness of the *incremental clausal maintenance operator* $T_{\mathcal{G}, \text{supp}}^{\Pi, s}$ (Lem. 5). We proceed by *induction* on Σ_{\triangleleft} . The *base case* follows from $\Sigma_{\triangleright} = \Sigma$, as $\mathcal{G} :+ : \Delta \models_{\Sigma_{\triangleright}} \Pi$ holds by assumption. For the *inductive case*, let $\Sigma_{\triangleleft} \equiv \{s\} \cup \Sigma'_{\triangleleft}$ and C be the clause $\Pi(s)$. Given a set of symbols S and an update Δ_O , such that $\mathcal{G} :+ : \Delta_O \models_S \Pi$, the *induction hypothesis (IH)* ensures that $\mathcal{G} :+ : \Delta_O \models_{S \cup \Sigma'_{\triangleleft}} \Pi$.

Now, we need to prove that $\mathcal{G} :+ : T_{\mathcal{G}, \text{supp}}^{\Pi}(\{s\} \cup \Sigma_{\triangleright}, \Sigma'_{\triangleleft}, T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta)) \models_{\Sigma_{\triangleright} \cup \{s\} \cup \Sigma'_{\triangleleft}} \Pi$. The conclusion results from instantiating (IH) with $S = \{s\} \cup \Sigma_{\triangleright}$ and $\Delta_O = T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta)$. To this end, we need to first establish that $\mathcal{G} :+ : T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta) \models_{\{s\} \cup \Sigma_{\triangleright}} \Pi$. This ensues from clause maintenance soundness and from modularity of model satisfaction.

The rest of preconditions needed by **IH** hold, as follows. From H4, since $(\{s\} \cup \Sigma'_{\triangleleft}) \cup \Sigma_{\triangleright}$ *stratifies* Π , we have that $\Sigma'_{\triangleleft} \cup \{s\} \cup \Sigma_{\triangleright}$ also stratifies Π . Moreover, $s \notin \Sigma_{\triangleright}$, and $\text{sym}(C) \subseteq \Sigma_{\triangleright}$. From H2, we have that Σ_{\triangleright} is a *well-formed slice* of Π , which, together with $\text{sym}(C) \subseteq \Sigma_{\triangleright}$, proves $\{s\} \cup \Sigma_{\triangleright}$ is a well-formed slice. From H3, we know that $\text{sym}(\Delta) \subseteq \Sigma_{\triangleright}$. The auxiliary `supp Δ _fwd_or_clause` lemma ensures that $\text{sym}(\Delta_O) \subseteq \{s\} \cup \text{sym}(\Delta)$; it then follows, by transitivity, that $\text{sym}(\Delta_O) \subseteq \{s\} \cup \Sigma_{\triangleright}$. The Coq proof is about 25 lines long and comparable to this text-based version in size. The first line sets up the induction, with the rest of the proof consisting in the instantiation of the proper lemmas. The statement of the theorem itself takes 7 lines for the preconditions (1 per line) plus an additional line for the conclusion: `ssTp (edb :+ : Δ_O) ($\Sigma \cup \Sigma_{\triangleright}$) Π` . \square

5 Experimental Analysis

In this section, we present the experimental validation of our certified engine on realistic graph databases. Our empirical analysis aims to confirm that *incremental view maintenance (IVM)* is more beneficial than *full view materialization (FVM)*, corresponding to recomputing the view from scratch, at each modification of the underlying data. The comparison has been established by computing the runtimes on the same engine.

As it is common practice in the verification community, our extracted engine has been obtained through the mechanism of *program extraction* (Letouzey 2008), starting from our underlying Coq formalization. Assuming that Coq extraction is semantics preserving, also the underlying premise of our engine, the *correctness* of the obtained OCaml engine is readily guaranteed by the Coq specification we provided. This is a reasonable assumption, made by past verified tools, such as (Leroy 2009).

Our tests have been performed on a Intel Core i7 vPro G6 laptop, with 16GB RAM, running Ubuntu 17.10 64 bit, and OCaml 4.06.0.

For our experimental analysis, we generated synthetic datasets and query workloads using gMark (Bagan et al. 2017), which allowed us to encode, two state-of-the-art benchmarks: WD, the Waterloo SPARQL Diversity Test Suite (Wat-Div) (Aluç et al. 2014),

and SNB, the LDBC Social Network Benchmark (Erling et al. 2015). These graphs – henceforth denoted by \mathcal{G} – are diverse in terms of their density (increasing from SNB to WD) and of their in-degree and out-degree distributions (Bagan et al. 2017). They represent two extreme cases to be considered in benchmarking graph database engines. Each schema size $|supp(\mathcal{G})|$ is fixed at 82 and 27 predicates, for WD, respectively, SNB.

Based on this, we generate graph instances and companion query workloads, such that $|\mathcal{G}| = 1K$ nodes and $|\mathcal{W}| = 10$ queries (queries represent views in our setting). Note that the gMark-generated queries are UC2RPQs – a notable subset of Regular Queries (RQs)³. This should not be considered a restriction, as these queries already let us stress (for view maintenance) the navigational part of our engine and use the recursion in the form of Kleene-star – a bottleneck in many practical graph query engines (such as Neo4J) (Bagan et al. 2017).

In order to build the deltas necessary for incremental view maintenance, we sampled the original graph instance \mathcal{G} by *support size*, i.e., by considering arbitrary subgraphs, whose number of symbols represent certain fractions of those in \mathcal{G} . We call this method *symbol-based sampling*. Concretely, we retained a varying percentage $\rho_{supp} \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$ from $supp(\mathcal{G})$. Next, we took all corresponding instance facts with symbols in ρ_{supp} to be our *bulk insertions*, Δ_+ , where $\frac{|supp(\Delta_+)|}{|supp(\mathcal{G})|} = \rho_{supp}$. These bulk insertions correspond to adding the subgraphs of \mathcal{G} , whose sets of symbols, sampled from $supp(\mathcal{G})$, reflect the ρ_{supp} ratio. We consider $\mathcal{G}' = \mathcal{G} \setminus \Delta_+$ to be our *base instances*, i.e., the initial sets of facts to be processed by our engine.

Since we relied on symbol-based sampling, the actual sizes of deltas ($|\Delta_+|$) vary depending on the content of \mathcal{G} . We denote the percentage capturing the relative sizes of the bulk inserts with respect to those of the base instances, as $\rho = \frac{|\Delta_+|}{|\mathcal{G}'|} * 100$ and report its actual values in the second column of each of the Tables 1 and 2.

Next, we evaluated each query in a workload \mathcal{W} over \mathcal{G}' and materialized the resulting views. Upon updating \mathcal{G}' with Δ_+ , we compared the *average timings* for *incremental view maintenance* (IVM) and *full view recomputation* (FVM), over all view materializations, in each of the workloads. We summarize the results in Table 1 and Table 2.

We observe that the *absolute time gain (ms)* of our engine running IVM with respect to it running FVM, i.e., **Time Gain** = FVM – IVM, is *always positive* and that the *relative ratio gain (%)*, i.e., **Ratio Gain** = $100 - \frac{100 * IVM}{FVM}$, is always better for sparser graphs. As expected, our engine works best on bulk updates involving very small individual symbol updates, as these types of updates are targeted by delta join matching. Indeed, the complexity of our delta join depends on how many matchings have to be computed. Note that the lower the sparsity of the underlying graph, the less matches we have and the faster our engine is. This explains why the runtimes over SNB (less dense) are comparatively much better than the ones over WD (very dense).

³ To the best of our knowledge, there currently is no practical benchmark capable of generating query workloads over the full fragment of Regular Datalog studied in this paper. The generation of graph query workloads for UC2RPQs has indeed been proved to be NP-complete (Bagan et al. 2017).

ρ_{supp}	ρ	FVM	IVM	Time Gain	Ratio Gain
0.05	1.4%	558.7	484.75	73.95	13.23%
0.1	3.67%	561.89	472.7	89.19	15.87%
0.15	17.93%	562.67	475.96	86.71	15.41%
0.2	9.7%	562.13	476.4	85.73	15.25%
0.25	18.26%	563.4	482.64	80.76	14.33%

Table 1: Avg. \mathcal{W}_{WD} Runtimes (ms) for Varying Support Update Size (ρ_{supp})

ρ_{supp}	ρ	FVM	IVM	Time Gain	Ratio Gain
0.05	10.89%	18.75	10.88	7.87	41.97%
0.1	19.3%	17.77	10.55	7.22	40.63%
0.15	10.77%	17.55	11.68	5.82	33.25%
0.2	26.09%	17.17	11.71	5.46	31.79%
0.25	28.34%	14.71	11	3.71	25.22%

Table 2: Avg. \mathcal{W}_{SNB} Runtimes (ms) for Varying Support Update Size (ρ_{supp})

6 Related Work

To the best of our knowledge, no verified graph query or IVM engine exists, which we both design and mechanize in this paper, using Regular Datalog. Bounded ⁴ incremental graph computation has been addressed in (Fan et al. 2017) and shown beyond reach already with Regular Path Queries (RPQs), a restricted navigational RQ subset. The paper’s idea is to incrementalize the bulk RPQ evaluation, by leveraging NFAs and auxiliary structures on large-scale graphs. We focus instead on the *verification* of forward-chaining-based IVM for the more expressive RQ graph query fragment. Although recent work has addressed certifying SQL semantics (Chu et al. 2017), by proving the semantic preservation of rewriting rules in SQL query optimizers for relational data, such a mechanization is not applicable to the graph-data setting, where the key data model component is no longer a tuple, but a *path* (connecting pairs of graph nodes). To this end, our Coq development is based on the standard connectivity notion from the Mathematical Components library.

Similarly, verified frameworks for the relational data model and nested relational algebra query compilers (Benzaken et al. 2014; Auerbach et al. 2017) are fairly orthogonal to our work. The optimization issues around RQ evaluation have never been *formally addressed* in the database literature and, even for the simple UC2RPQ class, current graph database engines perform poorly (see (Bagan et al. 2017)). Even though our goal is not to provide a RQ optimizer, we touch base with some simple optimizations. These are clause normalization, a lightweight indexing mechanism (leveraging graph edges in the definition of *supports*), and our *incremental supported satisfaction* definition. This leads to a more elegant framework for reasoning about incremental properties.

Despite RD not being fully recursive, our engine handles *stratified evaluation* in a Datalog style. Focusing on linear recursion (Jagadish et al. 1987) is indeed sufficient for our purposes, as it allows us to build a *RQ-specific engine* that inherently handles transitive closures. By limiting recursion, we can express *graph recursive* queries (RQs) that are ef-

⁴ The theory of bounded computational complexity for dynamic graph problems (Ramalingam and Reps 1996) considers the cost of incremental computation as a polynomial function of the input and output changes.

iciency prone, being highly parallelizable (Greenlaw et al. 1995). The work in (Benzaken et al. 2017) presents the SSReflect certification of a stratified static Datalog toy engine, implementing the bottom-up heuristic. While it supports Datalog’s full recursion, *this is actually a bottleneck* even for non-verified graph query engines (Bagan et al. 2017); also, it does not handle graph updates and IVM.

Finally, efficient Datalog engines have been designed in the last two decades, such as DLV2 (Alviano et al. 2017) and LogicBlox (Aref et al. 2015). Our proposed methodology can be implemented on top of such advanced engines, to combine their efficiency with our certified mechanization. We hope that our work paves the way for a future interplay of the various optimized heuristics and implemented semantics for Datalog, with its comprehensively verified evaluation. While the Coq extraction mechanism is mature and well-tested, we plan to integrate the recent advances on the trusted extraction and compilation (Anand et al. 2017; Mullen et al. 2018) of Coq code into our framework.

7 Conclusion and Perspectives

We propose a Coq formal library for *certified incremental graph query evaluation and view maintenance* in the Regular Datalog fragment. It consists of 1062 lines of definitions, specifying our mechanized theory, and 734 lines of proofs, establishing the central *soundness guarantee*. Our mechanized specification builds on a library fine-tuned for the computer-aided theorem proving of finite-set theory results. We take advantage from this, by giving a high-level, *mathematical* representation of core engine components. This leads to composable lemmas that boil down to set theoretic statements and, ultimately, to a condensed development, avoiding the proof-complexity explosion characteristic of formal verification efforts. Moreover, we managed to *extract* a runnable engine exhibiting performance gains versus the non-incremental approach on realistic graph database benchmarks. Our foundational approach shows it is promising to combine logic programming and proof assistants, such as Coq, to give certified specifications for both a uniform graph query language and its evaluation. We plan to mechanize more optimized heuristics, particularly for efficiently handling joins, and to integrate custom algorithms, such as Tarjan, for transitive closure computation.

Acknowledgments: We would like to thank the anonymous referees and Pierre Jouvelot for their very useful comments and feedback. S.Dumbrava was funded by the Datacert project, ANR-15-CE39-0009, and by ANR-11-IDEX-0007.

References

- ABITEBOUL, S., HULL, R., AND VIANU, V., Eds. 1995. *Foundations of Databases: The Logical Level*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- ALUÇ, G., HARTIG, O., ÖZSU, M. T., AND DAUDJEE, K. 2014. Diversified stress testing of RDF data management systems. In *The Semantic Web (ISWC 2014)*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, and C. Goble, Eds. LNCS, vol. 8796. Springer International Publishing, Cham, 197–212.
- ALVIANO, M., CALIMERI, F., DODARO, C., FUSCÀ, D., LEONE, N., PERRI, S., RICCA, F., VELTRI, P., AND ZANGARI, J. 2017. The ASP system DLV2. In *Logic Programming and Nonmonotonic Reasoning LPNMR 2017*. 215–221.

- ANAND, A., APPEL, A. W., MORRISSETT, G., PARASKEVOPOULOU, Z., POLLACK, R., BÉLANGER, O. S., SOZEAU, M., AND WEAVER, M. 2017. Certicoq: A verified compiler for Coq. In *CoqPL 2017: The 3rd International Workshop on Coq for Programming Languages*.
- ANGLES, R., ARENAS, M., BARCELÓ, P., HOGAN, A., REUTTER, J. L., AND VRGOC, D. 2017. Foundations of modern query languages for graph databases. In *ACM Comput. Surv.* Vol. 50. 68:1–68:40.
- AREF, M., TEN CATE, B., GREEN, T. J., KIMELFELD, B., OLTEANU, D., PASALIC, E., VELD-HUIZEN, T. L., AND WASHBURN, G. 2015. Design and implementation of the LogicBlox system. In *Proceedings of ACM SIGMOD*. 1371–1382.
- AUERBACH, J. S., HIRZEL, M., MANDEL, L., SHINNAR, A., AND SIMÉON, J. 2017. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. ACM, New York, NY, USA, 1555–1569.
- BAGAN, G., BONIFATI, A., CIUCANU, R., FLETCHER, G. H. L., LEMAY, A., AND ADVOKAAT, N. 2017. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (April), 856–869.
- BENZAKEN, V., CONTEJEAN, E., AND DUMBRAVA, S. 2014. A Coq formalization of the relational data model. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag New York, Inc., New York, NY, USA, 189–208.
- BENZAKEN, V., CONTEJEAN, E., AND DUMBRAVA, S. 2017. Certifying standard and stratified Datalog inference engines in SSreflect. In *Interactive Theorem Proving*. LNCS, vol. 10499. Springer International Publishing, 171–188.
- BEYHL, T. AND GIESE, H. 2016. Incremental view maintenance for deductive graph databases using generalized discrimination networks. In *GaM@ETAPS*. EPTCS, vol. 231. 57–71.
- CAI, Y., GIARRUSSO, P. G., RENDEL, T., AND OSTERMANN, K. 2014. A theory of changes for higher-order languages: Incrementalizing λ -calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. ACM, New York, NY, USA, 145–155.
- Cayley. <https://github.com/cayleygraph/cayley> (visited: 2018-02).
- CERI, S., GOTTLOB, G., AND TANCA, L. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1, 146–166.
- CHU, S., WEITZ, K., CHEUNG, A., AND SUCIU, D. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. ACM, New York, NY, USA, 510–524.
- CLARK, K. L. 1977. Negation as failure. In *Logic and Data Bases*, Gallaire and Minker, Eds. Plenum Press, 293–322.
- COHEN, C. AND THÉRY, L. 2017. Full script of Tarjan SCC Coq/SSreflect proof. Tech. rep., INRIA. <https://github.com/CohenCyril/tarjan> (visited: 2018-02).
- Cypher. <https://www.opencypher.org/> (visited: 2018-02).
- ERLING, O., AVERBUCH, A., LARRIBA-PEY, J., CHAFI, H., GUBICHEV, A., PRAT, A., PHAM, M.-D., AND BONCZ, P. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, New York, NY, USA, 619–630.
- FAN, W., HU, C., AND TIAN, C. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. ACM, New York, NY, USA, 155–169.
- FlockDB. <https://github.com/twitter-archive/flockdb> (visited: 2018-02).
- G-Core. https://github.com/ldbc/ldbc_gcore_parser (visited: 2018-02).
- Giraph. <http://giraph.apache.org/> (visited: 2018-02).

- GONTHIER, G., ASPERTI, A., AVIGAD, J., BERTOT, Y., COHEN, C., GARILLOT, F., ROUX, S. L., MAHBOUBI, A., O’CONNOR, R., BIHA, S. O., PASCA, I., RIDEAU, L., SOLOVYEV, A., TASSI, E., AND THÉRY, L. 2013. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*. LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 163–179.
- GraphQL. <http://graphql.org/> (visited: 2018-02).
- GREENLAW, R., HOOVER, H. J., AND RUZZO, W. L. 1995. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., New York, NY, USA.
- Gremlin. <http://tinkerpop.apache.org/> (visited: 2018-02).
- GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. 1993. Maintaining views incrementally. *SIGMOD Rec.* 22, 2, 157–166.
- JAGADISH, H. V., AGRAWAL, R., AND NESS, L. 1987. A study of transitive closure as a recursion mechanism. *SIGMOD Rec.* 16, 3, 331–344.
- LEROY, X. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7, 107–115.
- LETOUZEY, P. 2008. Extraction in Coq: An overview. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms*. CiE ’08. Springer-Verlag, Berlin, Heidelberg, 359–369.
- MULLEN, E., PERNSTEINER, S., WILCOX, J. R., TATLOCK, Z., AND GROSSMAN, D. 2018. (Euf: Minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. ACM, New York, NY, USA, 172–185.
- Neo4j. <https://neo4j.com/> (visited: 2018-02).
- Oracle PGX. <http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics> (visited: 2018-02).
- RAMALINGAM, G. AND REPS, T. W. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science* 158, 1&2, 233–277.
- REUTTER, J. L., ROMERO, M., AND VARDI, M. Y. 2017. Regular queries on graph databases. *Theory of Computing Systems* 61, 1, 31–83.
- SPARQL. <https://www.w3.org/TR/sparql11-query/> (visited: 2018-02).
- THE COQ DEVELOPMENT TEAM. 2018. The Coq proof assistant, version 8.7.2.

Appendix A Notations and Proofs Highlights

The main notations used in the paper are summarized in Table. A 1. We describe in some more detail the theories used in our formal proof, in particular we summarize the main lemma for *modular* model reasoning as well as the intermediate soundness results. The reader is encouraged to look at the development directly, whose definitions are intended to be readable and understood even by non-experts.

In the rest of the section, \mathcal{G} is assumed to be a labelled graph, g a non-labelled graph (set of edges), Δ an update, Π a program, C a clause, Σ , Σ_{\triangleright} , Σ_{\triangleleft} set of symbols, and s a symbol.

A.1 Formal Theory for Modular Satisfaction

Lemma 1 (Modularity of Clause Satisfaction (sTc_mod ♡)) Assume $s \notin \text{sym}(\Delta)$ and also $\text{sym}(C) \cap \text{sym}(\Delta) = \emptyset$. Then, $\mathcal{G} :+ \Delta \models_s C \iff \mathcal{G} \models_s C$.

$\Sigma, \mathcal{V}, \mathbf{V}$	\triangleq	Symbol (Signature), Variable, and Constant (Domain) Sets
$\sigma, \bar{\sigma}, \eta$	\triangleq	Substitution, Substitution Extension, and Closed Substitution (Grounding)
$\Sigma_{\triangleright}, \Sigma_{\triangleleft}$	\triangleq	Already and To-Be Processed Stratas
$\Pi, \mathcal{G}, \Delta, \Delta_+, \Delta_-$	\triangleq	RD Program, Graph Instance, Batch Updates, Insertions, and Deletions
$\Delta_+(s), \Delta_-(s)$	\triangleq	Batch Insertions and Deletions for symbol s
$V, V[\mathcal{G}]$	\triangleq	Top-Level RD Program View, View Materialization over Base Instance \mathcal{G}
$V[\mathcal{G} :+ \Delta]$	\triangleq	View Re-Materialization over Updated Instance $\mathcal{G} :+ \Delta$
$\Delta V[\mathcal{G} :+ \Delta]$	\triangleq	Incremental View Update
$T_{\mathcal{G}, \text{supp}}^{\Pi}(\Sigma_{\triangleright}, \Sigma_{\triangleleft}, \Delta)$	\triangleq	Program Maintenance Operator (<code>fwd_program</code>)
$T^{\Pi, s}(\mathcal{G})$	\triangleq	Base Clausal Maintenance Operator (<code>fwd_or_clause_base</code>)
$T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta)$	\triangleq	Incremental Clausal Maintenance Operator (<code>fwd_or_clause_delta</code>)
$M_{\mathcal{G}}^B, M_{\mathcal{G}, \Delta}^B$	\triangleq	Base and Incremental Body Matching (<code>match_body</code> , <code>match_delta_body</code>)
$M_{\mathcal{G}}^A, M_{\mathcal{G}, \Delta}^{A, m}$	\triangleq	Base and Incremental Atom Matching (<code>match_atom</code> , <code>match_delta_atom</code>)

Table A 1: Notation Table

Lemma 2 (Modularity of Program Satisfaction (`ssTp_mod` ♡)) Assume Σ a *well-formed slice* of Π and $s \notin \Sigma$. Let $\Delta' = (\Delta'_+, \Delta'_-)$, where $\Delta'_+ = \Delta_+ \cup \{s(t_1, t_2) \mid (t_1, t_2) \in g\}$ and $\Delta'_- = \Delta_- \setminus \{s(t_1, t_2) \mid (t_1, t_2) \in g\}$. Then,

$$\mathcal{G} :+ \Delta' \models_{\{s\} \cup \Sigma} \Pi \iff \mathcal{G} :+ \Delta' \models_s \Pi(s) \wedge \mathcal{G} :+ \Delta \models_{\Sigma} \Pi$$

A.2 Formal Theory for Clause-level Operators

Lemma 3 (Soundness of Base Clausal Maintenance (`fwd_or_clause_baseP` ♡)) Assume (H1) $\Pi(s)$ is a *safe* clause; (H2) Σ_{\triangleright} is complete for closures; (H3) Σ_{\triangleright} is a *well-formed slice* of Π ; (H4) $s \notin \Sigma_{\triangleright}$; (H) $\text{sym}(\Pi(s)) \subseteq \Sigma_{\triangleright}$, and $\mathcal{G} \models_{\Sigma_{\triangleright}} \Pi$. Then $\mathcal{G} :+ \Delta_s \models_{\{s\} \cup \Sigma_{\triangleright}} \Pi$, where $\Delta_s = T^{\Pi, s}(\mathcal{G} :+ \Delta)$.

Lemma 4 (Soundness of Incr. Clausal Maintenance (`fwd_or_clause_deltaP` ♡)) Assume (H1) $\Pi(s)$ is a *safe* clause; (H2) Σ_{\triangleright} is complete for closures; (H3) Σ_{\triangleright} is a *well-formed slice* of Π ; (H4) $s \notin \Sigma_{\triangleright}$; (H5) $\text{sym}(\Pi(s)) \subseteq \Sigma_{\triangleright}$; (H6) $\mathcal{G} :+ \Delta \models_{\Sigma_{\triangleright}} \Pi$; (H7) $\text{sym}(\Delta) \subseteq \Sigma_{\triangleright}$. Also assume the *incrementality conditions*: (H8) $\mathcal{G} \models_{\Sigma} \Pi$; (H9) $s \in \Sigma$; (H10) $\text{sym}(\Pi(s)) \cap \text{sym}(\Delta_-)$. Then, $\mathcal{G} :+ \Delta_s \models_{\{s\} \cup \Sigma_{\triangleright}} \Pi$, where $\Delta_s = T_{\mathcal{G}}^{\Pi, s}(\Delta)$. This operator is called by the supported maintenance operator when incrementality can be used.

Lemma 5 (Soundness of Supported Clausal Maintenance (`fwd_or_clauseP` ♡)) Assume (H1) $\Pi(s)$ is a *safe* clause, (H2) $\mathcal{G} \models_{\Sigma} \Pi$; (H3) Σ_{\triangleright} is well-formed wrt closures; (H4) Σ_{\triangleright} is a *well-formed slice* of Π ; (H5) $s \notin \Sigma_{\triangleright}$; (H6) $\text{sym}(\Pi(s)) \subseteq \Sigma_{\triangleright}$; (H7) $\text{sym}(\Delta) \subseteq \Sigma_{\triangleright}$. If $\mathcal{G} :+ \Delta \models_{\Sigma_{\triangleright}} \Pi$, then $\mathcal{G} :+ \Delta_s \models_{\{s\} \cup \Sigma_{\triangleright}} \Pi$, where $\Delta_s = T_{\mathcal{G}, \text{supp}}^{\Pi, s}(\Delta)$.

Lemma 6 (Soundness of Incr. Body Matching (`fwd_delta_body_sound` ♡)) Let B a conjunctive body; σ a substitution. Assume $\text{sym}(B) \cap \text{sym}(\Delta_-) = \emptyset$, that is to say, no deletions are scheduled for B , then for all $\sigma \in M_{\mathcal{G}, \Delta}^B(B)$ there exists an instantiation of B , \bar{B} , such that $\sigma(B) = \bar{B}$.