Materializing Knowledge Bases via Trigger Graphs

Efthymia Tsamoura, David Carral, Enrico Malizia, Jacopo Urbani

Materializing Knowledge Bases via Trigger Graphs (Technical Report)

Efthymia Tsamoura*, David Carral†, Enrico Malizia‡, Jacopo Urbani◇
*Samsung AI Research, United Kingdom; †TU Dresden, Germany;
‡ University of Bologna, Italy; ◇Vrije Universiteit Amsterdam, The Netherlands

ABSTRACT

The *chase* is a well-established family of algorithms used to materialize Knowledge Bases (KBs), like Knowledge Graphs (KGs), to tackle important tasks like query answering under dependencies or data cleaning. A general problem of chase algorithms is that they might perform redundant computations. To counter this problem, we introduce the notion of *Trigger Graphs* (TGs), which guide the execution of the rules avoiding redundant computations. We present the results of an extensive theoretical and empirical study that seeks to answer when and how TGs can be computed and what are the benefits of TGs when applied over real-world KBs. Our results include introducing algorithms that compute (minimal) TGs. We implemented our approach in a new engine, and our experiments show that it can be significantly more efficient than the chase enabling us to materialize KBs with 17B facts in less than 40 min on commodity machines.

PVLDB Reference Format:
. Materializing Knowledge Bases via Trigger Graphs. *PVLDB*, 12(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Motivation. Knowledge Bases (KBs) are becoming increasingly important with many industrial key players investing on this technology. For example, Knowledge Graphs (KGs) [32] have emerged as the main vehicle for representing factual knowledge on the Web and enjoy a widespread adoption [48]. Moreover, several tech giants are building KGs to support their core business. For instance, the KG developed at Microsoft contains information about the world and supports question answering, while, at Google, KGs are used to help Google products respond more appropriately to user requests by mapping them to concepts in the KG. The use of KBs and KGs in such scenarios is not restricted only to database-like analytics or query answering: KBs play also a central role in neural-symbolic systems for efficient learning and explainable AI [23, 36].

A KB can be viewed as a classical database B with factual knowledge and a set of logical rules P, called *program*, allowing the derivation of additional knowledge. One class of rules that is of particular interest both to academia and to industry is Datalog [2]. Datalog is a recursive language with declarative semantics that allows users to succinctly write recursive graph queries. Beyond expressing graph queries, e.g., reachability, Datalog allows richer fixed-point graph analytics via aggregate functions. LogicBlox and LinkedIn used Datalog to develop high-performance applications, or to compute analytics over its KG [3, 46]. Google developed their own Datalog engine called Yedalog [21]. Other industrial users include Facebook, BP [10] and Samsung [40].

Materializing a KB (P, B) is the process of deriving all the facts that logically follow when reasoning over the database B using the rules in P. Materialization is a core operation in KB management. An obvious use is that of caching the derived knowledge. A second use is that of *goal-driven query answering*, i.e., deriving the knowledge specific to a given query *only*, using database techniques such as magic sets and subsumptive tabling [8, 9, 13, 55]. Beyond knowledge exploration, other applications of materialization are data wrangling [35], entity resolution [37], data exchange [26] and query answering over OWL [44] and RDFS [16] ontologies. Finally, materialization has been also used in probabilistic KBs [56].

Problem. The increasing sizes of modern KBs [48], and the fact that materialization is not a one-off operation when used for goal-driven query answering, make improving the materialization performance critical. The *chase*, which was introduced in 1979 by Maier et al. [42], has been the most popular materialization technique and has been adopted by several commercial and open source engines such as VLog [58], RDFox [47] and Vadalog [10].

To improve the performance of materialization, different approaches have focused on different inefficiency aspects. One approach is to reduce the number of

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

facts added in the KB. This is the take of some of the chase variants proposed by the database and AI communities [11, 24, 49]. A second approach is to parallelize the computation. For example, RDFox proposes a parallelization technique for Datalog rules [47], while WebPIE [59] and Inferray [54] propose parallelization techniques for fixed RDFS rules. Orthogonal to those approaches are those employing compression and columnar storage layouts to reduce memory consumption [34, 58].

In this paper, we focus on a different aspect: that of avoiding redundant computations. Redundant computations is a problem that concerns all chase variants and has multiple causes. A first cause is the derivation of facts that either have been derived in previous rounds, or are logically redundant, i.e., they can be ignored without compromising query answering. The above issue has been partially addressed in Datalog with the well-known seminaïve evaluation (SNE) [2]. SNE restricts the execution of the rules over at least one new fact. However, it cannot block the derivation of the same or logically redundant facts by different rules. A second cause of redundant computations relates to the execution of the rules: when executing a rule, the chase may consider facts that cannot lead to any derivations.

Our approach. To reduce the amount of redundant computations, we introduce the notion of *Trigger Graphs (TGs)*. A TG is an acyclic directed graph that captures all the operations that should be performed to materialize a KB (P, B). Each node in a TG is associated with a rule from P and with a set of facts, while the edges specify the facts over which we execute each rule.

Intuitively, a TG can be viewed as a blueprint for reasoning over the KB. As such, we can use it to "guide" a reasoning procedure without resorting to an exhaustive execution of the rules, as it is done with the chase. In particular, our approach consists of traversing the TG, executing the rule r associated with a node v over the union of the facts associated with the parent nodes of v and storing the derived facts "inside" v. After the traversal is complete, then the materialization of the KB is simply the union of the facts in all the nodes.

TG-guided materialization addresses *at the same time all* causes of inefficiencies described above. In particular, TGs block the derivation of the same or logically redundant facts that cannot be blocked by SNE. This is achieved by effectively partitioning into smaller sub-instances the facts currently in the KB. This partitioning also enables us to reduce the cost of executing the rules.

Furthermore, in specific cases, TGs allow us reasoning via either completely avoiding certain steps involved when executing rules, or performing them at the end and collectively for all rules. Our experiments show that we get good runtime improvements with both alternatives.

Contributions. We propose techniques for computing both instance-independent and instance-dependent TGs. The former TGs are computed exclusively based on the rules of the KB and allow us to reason over *any* possible instance of the KB making them particularly useful when the database changes frequently. In contrast, instance-dependent TGs are computed based both on the rules and the data of the KB and, thus, support reasoning over the given KB *only*. We show that not every program admits a finite instance-independent TG. We define a special class, called FTG, including all programs that admit a finite instance-independent TG and explore its relationship with other known classes.

As a second contribution, we propose algorithms to compute and minimize (instance-independent) TGs for linear programs: a class of programs relevant in practice. A program P not admitting a finite *instance-independent* TG may still admit a finite *instance-dependent* TG.

As a third contribution, we show that all programs that admit a finite universal model also admit a finite *instance-dependent* TG. We use this finding to propose a TG-guided materialization technique that supports *any* such program (not necessarily in FTG). The technique works by interleaving the reasoning process with the computation of the TG, and it reduces the number of redundant computations via query containment and via a novel TG-based rule execution strategy.

We implemented our approach in a new reasoner, called GLog, and compared its performance versus multiple state-of-the-art chase and RDFS engines including RDFox, VLog, WebPIE [59] and Inferray [54], using well-established benchmarks, e.g., ChaseBench [11]. Our evaluation shows that GLog outperforms all its competitors in all benchmarks. Moreover, in our largest experiment, GLog was able to materialize a KB with 17B facts in 37 minutes on commodity hardware.

Summary. We make the following contributions:

- (*New idea*) We propose a new reasoning technique based on traversing acyclic graphs, called TGs, to tackle multiple sources of inefficiency of the chase;
- (*New theoretical contribution*) We study the class of programs admitting finite instance-independent TGs and its relationship with other known classes;
- (*New algorithms*) We propose techniques for computing minimal instance-independent TGs for linear programs, and techniques for computing minimal instance-dependent TGs for Datalog programs;
- (*New system*) We introduce a new reasoner, GLog, which has competitive performance, often superior to the state-of-the-art, and has good scalability.

Supplementary material with all proofs, code and evaluation data is in `https://bitbucket.org/tsamoura/trigger-graphs/src/master/`.

2. MOTIVATING EXAMPLE

We start our discussion with a simple example to describe how the chase works, its inefficiencies, and how they can be overcome with TGs. For the moment, we give only an intuitive description of some key concepts to aid the understanding of the main ideas. In the following sections, we will provide a formal description.

The chase works in rounds during which it executes the rules over the facts that are currently in the KB.

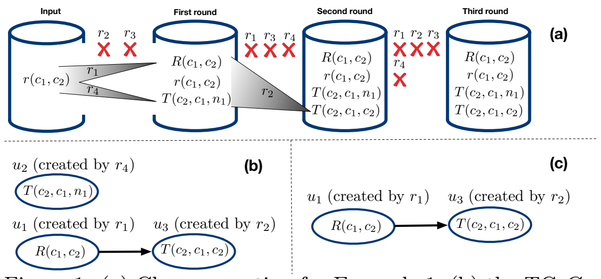

Figure 1: (a) Chase execution for Example 1, (b) the TG G_1, (c) the TG G_2. In (b) and (c), the facts shown inside the nodes are the results of reasoning over B using the TG.

In most chase variants, the execution of a rule involves three steps: retrieving all the facts that instantiate the premise of the rule, then, checking whether the facts to be derived logically hold in the KB and finally, adding them to the KB if they do.

EXAMPLE 1. *Consider the KB $B = \{r(c_1, c_2)\}$ with a single fact and the program $P_1 = \{r_1, r_2, r_3, r_4\}$:*

$$r(X, Y) \to R(X, Y) \qquad (r_1)$$

$$R(X, Y) \to T(Y, X, Y) \qquad (r_2)$$

$$T(Y, X, Y) \to R(X, Y) \qquad (r_3)$$

$$r(X, Y) \to \exists Z.T(Y, X, Z) \qquad (r_4)$$

Figure 1 (a) depicts the rounds of the chase with such an input. In the first round, the only rules that can derive facts are r_1 and r_4. Rule r_1 derives the fact $R(c_1, c_2)$. Since this fact is not in the KB, the chase adds it to the KB. Let us now focus on r_4. Notice that variable Z in r_4 does not occur in the premise of r_4. The chase deals with such variables by introducing fresh null *(values). Nulls can be seen as "placeholders" for objects that are not known. In our case, r_4 derives the fact $T(c_2, c_1, \mathsf{n}_1)$, where n_1 is a null, and the chase adds it to the KB.*

The chase then continues to the second round where rules are executed over $B' = B \cup \{R(c_1, c_2), T(c_2, c_1, \mathsf{n}_1)\}$. The execution of r_2 derives the fact $T(c_2, c_1, c_2)$, which is added to the KB, yielding $B'' = B' \cup \{T(c_2, c_1, c_2)\}$. Finally, the chase proceeds to the third round where only rule r_3 derives $R(c_1, c_2)$ from B''. However, since this fact is already in B'', the chase stops.

The above steps expose two inefficiencies of the chase. The first is that of incurring in the cost of deriving the same or logically redundant facts.

EXAMPLE 2. *Let us return back to Example 1. The chase pays the cost of executing r_3 despite that r_3's execution always derives facts derived in previous rounds. This is due to the cyclic dependency between rules r_2 and r_3: r_2 derives T-facts by flipping the arguments of the R-facts, while r_3 derives R-facts by flipping the arguments of the T-facts. Despite that the SNE effectively blocks the execution of r_1 and r_2 in the third chase round, it cannot block the execution of r_3 in the third chase round, since $T(c_2, c_1, c_2)$ was derived in the second round.*

Now, consider the fact $T(c_2, c_1, \mathsf{n}_1)$. This fact is logically redundant, because it provides no extra information over $T(c_2, c_1, c_2)$, derived by r_2. Despite being logically redundant, the chase pays the cost of deriving it.

The second inefficiency that is exposed is that of suboptimally executing the rules themselves: when computing the facts instantiating the premise of a rule, the chase considers all facts currently in the KB even the ones that cannot instantiate the premise of the rule.

EXAMPLE 3. *Continuing with Example 1, consider the execution of r_3 in the second round of the chase. No fact derived by r_4 can instantiate the premise of r_3, since the premise of r_3 requires the first and the third arguments of the T-facts to be the same. Hence, the cost paid for executing r_3 over those facts is unnecessary.*

The root of these inefficiencies is that the chase, in each round, considers the entire KB as a source for potential derivations, with only SNE as means to avoid some redundant derivations. If we were able to "guide" the execution of the rules in a more clever way, then we can avoid the inefficiencies stated above.

For instance, consider an alternative execution strategy where r_2 is executed *only* over the derivations of r_1, while r_3 and r_4 are not executed at all. This strategy would not face any of the inefficiencies highlighted above, and it can be defined with a graph like the one in Figure 1 (c). Informally, a *Trigger Graph* (TG) is precisely such a graph-based blueprint to compute the materialization. In the remaining, we will first provide a formal definition of TGs and study their properties. Then, we will show that in some cases we can build a TG that is optimal for any possible set of facts given as input. In other cases, we can still build TGs incrementally. Such TGs allow to avoid redundant computations that will occur with the chase but only with the given input.

3. PRELIMINARIES

Let Consts, Nulls, Vars, and Preds be mutually disjoint, (countably infinite) sets of *constants*, *nulls*, *variables*, and *predicates*, respectively. Each predicate p is associated with a non-negative integer $\mathsf{arity}(p) \geq 0$, called the *arity* of p. Let EDP and IDP be disjoint subsets of Preds of *intensional* and *extensional predicates*, respectively. A *term* is a constant, a null, or a variable. A term is ground if it is either a constant or a null. An *atom* A has the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate, and $t_1, \ldots, t_n$ are terms. An atom A is extensional (resp., intensional), if the predicate of A is in EDP (resp., IDP). A *fact* is an atom of ground terms. A *base fact* is an atom of constants whose predicate is extensional. An *instance* I is a set of facts (possibly comprising null terms). A *base instance* B is a set of base facts.

A *rule* is a first-order formula of the form

$$\forall \mathbf{X} \forall \mathbf{Y} \bigwedge_{i=1}^{n} P_i(\mathbf{X}_i, \mathbf{Y}_i) \to \exists \mathbf{Z} P(\mathbf{Y}, \mathbf{Z}), \qquad (1)$$

where, P is an intensional predicate and for all $1 \leq i \leq n$, $\mathbf{X}_i \subseteq \mathbf{X}$ and $\mathbf{Y}_1 \subseteq \mathbf{Y}$ ($\mathbf{X}_i$ and $\mathbf{Y}_i$ might be empty). We assume w.l.o.g. that the body of a rule includes only extensional predicates or intensional predicates. We will denote extensional predicates with lowercase letters, while intensional predicates with uppercase letters. Quantifiers are commonly omitted. The left-hand and the right-hand side of a rule r are its *body* and *head*, respectively, and are denoted by $\mathsf{body}(r)$ and $\mathsf{head}(r)$. A rule is *Datalog* if it has no existentially quantified variables, *extensional* if $\mathsf{body}(r)$ includes only extensional atoms, and *linear* if it has a single atom in its body.

A *program* is a set of rules. A *knowledge base* (KB) is a pair (P, B) with P a program and B a base instance.

Symbol $\models$ denotes logical entailment, where sets of atoms and rules are viewed as first-order theories. Symbol $\equiv$ denotes logical equivalence, i.e., logical entailment in both directions.

A *term mapping* σ is a (possibly partial) mapping of terms to terms; we write $\sigma = \{t_1 \mapsto s_1, \dots, t_n \mapsto s_n\}$ to denote that $\sigma(t_i) = s_i$ for $1 \leq i \leq n$. Let α be a term, an atom, a conjunction of atoms, or a set of atoms. Then $\sigma(\alpha)$ is obtained by replacing each occurrence of a term t in α that also occurs in the domain of σ with $\sigma(t)$ (i.e., terms outside the domain of σ remain unchanged). A *substitution* is a term mapping whose domain contains only variables and whose range contains only ground terms. For two sets, or conjunctions, of atoms $\mathcal{A}_1$ and $\mathcal{A}_2$, a term mapping σ from the terms occurring in $\mathcal{A}_1$ to the terms occurring in $\mathcal{A}_2$ is said to be a *homomorphism* from $\mathcal{A}_1$ to $\mathcal{A}_2$ if the following hold: (i) σ maps each constant in its domain to itself, (ii) σ maps each null in its domain to $\mathsf{Consts} \cup \mathsf{Nulls}$ and (iii) for each atom $A \in \mathcal{A}_1$, $\sigma(A) \in \mathcal{A}_2$. We denote a homomorphism σ from $\mathcal{A}_1$ into $\mathcal{A}_2$ by $\sigma : \mathcal{A}_1 \to \mathcal{A}_2$.

It is known that, for two sets of facts $\mathcal{A}_1$ and $\mathcal{A}_2$, there exists a homomorphism from $\mathcal{A}_1$ into $\mathcal{A}_2$ iff $\mathcal{A}_2 \models \mathcal{A}_1$ (and hence, there exists a homomorphism in both ways iff $\mathcal{A}_1 \equiv \mathcal{A}_2$). When $\mathcal{A}_1$ and $\mathcal{A}_2$ are null-free instances, $\mathcal{A}_2 \models \mathcal{A}_1$ iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and $\mathcal{A}_2 \equiv \mathcal{A}_1$ iff $\mathcal{A}_1 = \mathcal{A}_2$.

For a set of two or more atoms $\mathcal{A} = \{A_1, \dots, A_n\}$ a *most general unifier* (MGU) μ for $\mathcal{A}$ is a substitution so that: (i) $\mu(A_1) = \cdots = \mu(A_n)$; and (ii) for each other substitution σ for which $\sigma(A_1) = \cdots = \sigma(A_n)$, there exists a σ' such that $\sigma = \sigma' \circ \mu$ [5].

Consider a rule r of the form (1) and an instance I. A *trigger* for r in I is a homomorphism from the body of r into I. We denote by h_s the extension of a trigger h mapping each $Z \in \mathbf{Z}$ into a unique fresh null. A rule r *holds* or is *satisfied* in an instance I, if for each trigger h for r in I, there exists an extension h' of h to a homomorphism from the head of r into I. A *model* of a KB (P, B) is a set $I \supseteq B$, such that each $r \in P$ holds in I. A KB may admit infinitely many different models. A model M is *universal*, if there exists a homomorphism from M into every other model of (P, B). A program P is *Finite Expansion Set* (*FES*), if for each base instance B, (P, B) admits a finite universal model.

A *conjunctive query* (CQ) is a formula of the form $Q(X_1, \dots, X_n) \leftarrow \bigwedge_{i=1}^{m} A_i$, where Q is a fresh predicate not occurring in P, A_i are null-free atoms and each X_j occurs in some A_i atom. We usually refer to a CQ by its head predicate. We refer to the left-hand and the right-hand side of the formula as the *head* and the *body* of the query, respectively. A CQ is *atomic* if its body consists of a single atom. A Boolean CQ (BCQ) is a CQ whose head predicate has no arguments. A substitution σ is an *answer* to Q on an instance I if the domain of σ is precisely its head variables, and if σ can be extended to a homomorphism from $\bigwedge_i A_i$ into I. We often identify σ with the n-tuple $(\sigma(X_1), \dots, \sigma(X_n))$. The *output* of Q on I is the set $Q(I)$ of all answers to Q on I. The answer to a BCQ Q on an instance I is true, denoted as $I \models Q$, if there exists a homomorphism from $\bigwedge_{i=1} A_i$ into I. The answer to a BCQ Q on a KB (P, B) is true, denoted as $(P, B) \models Q$, if $M \models Q$ holds, for each model M of (P, B). Finally, a CQ Q_1 is *contained* in a CQ Q_2, denoted as $Q_1 \subseteq Q_2$, if for each instance I, each answer to Q_1 on I is in the answers to Q_2 on I [20].

The *chase* refers to a family of techniques for repairing a base instance B relative to a set of rules P so that the result satisfies the rules in P and contains all base facts from B. In particular, the result is a universal model of (P, B), which we can use for query answering [26]. By "chase" we refer both to the procedure and its output.

The chase works in rounds during which it executes one or more rules from the KB. The result of each round $i \geq 0$ is a new instance I^i (with $I^0 = B$), which includes the facts of all previous instances plus the newly derived facts. The execution of a rule in the i-th chase round, involves computing all triggers from the body of r into I^{i-1}, then (potentially) checking whether the facts to be derived satisfy certain criteria in the KB and finally, adding to the KB or discarding the derived facts. Different chase variants employ different criteria for deciding whether a fact should be added to the KB or whether to stop or continue the reasoning process [11, 49]. For example, the restricted chase (adopted by VLog and RDFox) adds a fact if there exists no homomorphism from this fact into the KB and terminates when no new fact is added. The warded chase (adopted by Vadalog) replaces homomorphism checks by isomorphism ones [10] and terminates, again, when no new fact is added. The equivalent chase omits any checks and terminates when there is a round i which produces an instance that is logically equivalent to the instance produced in the $(i-1)$-th round [24]. Notice that when a KB includes only Datalog rules all chase variants behave the same: a fact is added when it has not been previously derived and the chase stops when no new fact is added to the KB.

Not all chase variants terminate even when the KB admits a finite universal model [24]. The core chase [25] and the equivalent one do offer such guarantees.

For a chase variant, we use $\mathit{Ch}^i(K)$ or $\mathit{Ch}^i(P, B)$ to denote the instance computed during the i-th chase round and $\mathit{Ch}(P, B)$ to denote the (possibly infinite) result of the chase. Furthermore, we define the *chase graph* $\mathsf{chaseGraph}(P, B)$ for a KB (P, B) as the edge-labeled directed acyclic graph having as nodes the facts in $\mathit{Ch}(P, B)$ and having an edge from a node f_1 to f_2

labeled with rule $r \in P$ if f_2 is obtained from f_1 and possibly from other facts by executing r.

4. TRIGGER GRAPHS

In this section, we formally define Trigger Graphs (TGs) and study the class of programs admitting finite *instance-independent* TGs. First, we introduce the notion of *Execution Graphs* (EGs). Intuitively, an EG for a program is a digraph stating a "plan" of rule execution to reason via the program. In its general definition, an EG is not required to characterize a plan of reasoning guaranteeing completeness. Particular EGs, defined later, will also satisfy this property.

Definition 4. *An* execution graph *(EG) for a program P is an acyclic, node- and edge-labelled digraph $G = (V, E, \mathsf{rule}, \ell)$, where V and E are the graph nodes and edges sets, respectively, and* rule *and ℓ are the node- and edge-labelling functions. Each node v (i) is labelled with some rule, denoted by $\mathsf{rule}(v)$, from P; and (ii) if the j-th predicate in the body of $\mathsf{rule}(v)$ equals the head predicate of $\mathsf{rule}(u)$ for some node u, then there is an edge labelled j from node u to node v, denoted by $u \rightarrow_j v$.*

Figures 1(b) and 1(c) show two EGs for P_1 from Example 1. Next to each node is the associated rule. Later we show that both EGs are also TGs for P_1.

Since the nodes of an execution graph are associated with rules of a program, when, in the following, we refer to the head and the body of a node v, we actually mean the head and the body of $\mathsf{rule}(v)$. Observe that, by definition, nodes associated with extensional rules do not have entering edges, and nodes v associated with an intensional rule have at most *one* incoming edge associated with the j-th predicate of the body of v, i.e., there is at most one node u such that $u \rightarrow_j v$. The latter might seem counterintuitive as, in a program, the j-th predicate in the body of a rule can appear in the heads of many different rules. It is precisely to take into account this possibility that, in an execution graph, more than one node can be associated with the same rule r of the program. In this way, different nodes $v_1, \dots, v_q$ associated with the same rule r can be linked with an edge labeled j to different nodes $u_1, \dots, u_q$ whose head's predicate is the j-th predicate of the body of r. This models that to evaluate a rule r we might need to match the j-th predicate in the body of r with facts generated by the heads of different rules.

We now define some notions on EGs that we will use throughout the paper. For an EG G for a program P, we denote by $\nu(G)$ and $\epsilon(G)$ the sets of nodes and edges in G. The depth of a node $v \in \nu(G)$ is the length of the longest path that ends in v. The depth $\mathsf{d}(G)$ of G is 0 if G is the empty graph; otherwise, it is the maximum depth of the nodes in $\nu(V)$.

As said earlier, EGs can be used to guide the reasoning process. In the following definition, we formalise how the reasoning over a program P is carried out by following the plan encoded in an EG for P. The definition assumes the following for each rule r in P: (i) r is of the form $\forall \mathbf{X} \forall \mathbf{Y} \bigwedge_{i=1}^{n} P_i(\mathbf{X}_i, \mathbf{Y}_i) \rightarrow \exists \mathbf{Z} P(\mathbf{Y}, \mathbf{Z})$; and (ii) if r is intensional and is associated with a node v in an EG for P, then the EG includes an edge of the form $u_i \rightarrow_i v$, for each $1 \leq i \leq n$.

Definition 5. *Let (P, B) be a KB, G be an EG for P and v be a node in G associated with rule $r \in P$. $v(B)$ includes a fact $h_s(\mathsf{head}(r))$, for each h that is either:*

- *a homomorphism from the body of r to B, if r is extensional; or otherwise*
- *a homomorphism from the body of r into $\bigcup_{i=1}^{n} u_i(B)$ so that the following holds: the restriction of h over $\mathbf{X}_i \cup \mathbf{Y}_i$ is a homomorphism from $P_i(\mathbf{X}_i, \mathbf{Y}_i)$ into $u_i(B)$, for each $1 \leq i \leq n$.*

We pose $G(B) = B \cup \bigcup_{v \in V} v(B)$.

TGs are EGs guaranteeing the correct computation of conjunctive query answering.

Definition 6. *An EG G is a TG for (P, B), if for each BCQ Q, $(P, B) \models Q$ iff $G(B) \models Q$. G is a TG for P, if for each base instance B, G is a TG for (P, B).*

TGs that depend both on P and B are called *instance-dependent*, while TGs that depend only on P are called *instance-independent.* The EGs shown in Figure 1 are both instance-independent TGs for P_1.

We provide an analysis of the class of programs that admit a finite instance-independent TG denoted as *FTG*. Theorem 7 summarizes the relationship between *FTG* and the classes of programs that are bounded (*BDD*, [24]), term-depth bounded (*TDB*, [39]) and first-order-rewritable (*FOR*, [18]).

Theorem 7. *The following hold: P is* FTG *iff it is* BDD*; and P is* TDB $\cap$ FOR *iff it is* BDD.

This result is obtained by showing that if P is *FTG*, then it is *BDD* with bound the maximal depth of any instance-independent TG for P. If it is *BDD* with bound k, then the (finite) EG G^k, which is described after Definition 9, is a TG for P.

If a program is *FOR*, then all facts that contain terms of depth at most k are produced in a fixed number of chase steps. Therefore, if it is also *TDB*, then all relevant facts in the chase are also produced in a fixed number of steps. Finally, the undecidability of *FTG* follows from the fact that *FOR* and *FTG* coincide for Datalog programs, which are always *TDB.* See the appendix for a detailed explanation.

We conclude our analysis by showing that any KB that admits a finite model, also admits a finite instance-dependent TG, as stated in the following statement.

Theorem 8. *For each KB (P, B) that admits a finite model, there exists an instance-dependent TG.*

The key insight is that we can build a TG that mimics the chase. Below, we analyze the conditions under which the same rule execution takes place both in the chase and when reasoning over a TG. Based on this analysis we present a technique for computing instance-dependent TGs that mimic breadth-first chase variants.

Consider a rule of the form (1) and assume that the chase over a KB (P, B) executes r in some round k by instantiating its body using the facts $R(\mathbf{c}_i)$. Consider now a TG G for (P, B). If $k = 1$, then this rule execution (notice that the rule has to be extensional) takes place in G if there is a node v associated with r. Otherwise, if $k > 1$, then this rule execution takes place in G if the following holds: (i) there is a node v associated with r, (ii) each $R(\mathbf{c}_i)$ is stored in some node u_i and (iii) there is an incoming edge $u_i \rightarrow_i v$, for each $1 \leq i \leq n$. We refer to each combination of nodes of depth $< k$ whose facts may instantiate the body of a rule r when reasoning over an EG, as k-compatible nodes for r:

Definition 9. *Let P be a program, r be an intensional rule in P and G be an EG for P. A combination of n (not-necessarily distinct) nodes $(u_1, \dots, u_n)$ from G is k-compatible with r, where $k \geq 2$ is an integer, if:*

- *the predicate in the head of u_i is R_i;*
- *the depth of each u_i is less than k; and*
- *at least one node in $(u_1, \dots, u_n)$ is of depth $k - 1$.*

The above ideas are summarized in an iterative procedure, which builds at each step k a graph G^k:

- (**Base step**) if $k = 1$, then for each extensional rule r add to G^k a node v associated with r.
- (**Inductive step**) otherwise, for each intensional rule r and each combination of nodes $(u_1, \dots, u_n)$ from G^{k-1} that is k-compatible with r, add to G^k: (i) a fresh node v associated with r and (ii) an edge $u_i \rightarrow_i v$, for each $1 \leq i \leq n$.

The inductive step ensures that G^k encodes each rule execution that takes place in the k-th chase round.

So far, we did not specify when the TG computation process stops. When P is Datalog, we can stop when $G^{k-1}(B) = G^k(B)$. Otherwise, we can employ the termination criterion of the equivalent chase, e.g., $G^{k-1}(B) \models G^k(B)$, or of the restricted chase.

5. TGS FOR LINEAR PROGRAMS

In the previous section, we outlined a procedure to compute *instance-dependent TGs* that mimics the chase. Now, we propose an algorithm for computing *instance-independent* TGs for linear programs.

Our technique is based on two ideas. The first one is that, for each base instance B, the result of chasing B using a linear program P is logically equivalent to the union of the instances computed when chasing each single fact in B using P.

The second idea is based on *pattern-isomorphic* facts: facts with the same predicate name and for which there is a bijection between their constants. For example, $R(1, 2, 3)$ is pattern-isomorphic to $R(5, 6, 7)$ but not to $R(9, 9, 8)$. We can see that two different pattern-isomorphic facts will have the same linear rules executed in the same order during chasing. We denote by $\mathcal{H}(P)$ a set of facts formed over the extensional predicates in a program P, where no fact $f_1 \in \mathcal{H}(P)$ is pattern isomorphic to some other fact $f_2 \in \mathcal{H}(P)$.

Algorithm 1 $\mathsf{tglinear}(P)$

```
1: Let $G$ be an empty EG
2: for each $f \in \mathcal{H}(P)$ do
3:   $\Gamma$ is an empty $EG$; $\mu$ is the empty mapping
4:   for each $f_1 \rightarrow_r f_2 \in \mathsf{chaseGraph}(P, \{f\})$ do
5:     add a fresh node $u$ to $\nu(\Gamma)$ with $\mathsf{rule}(u) := r$
6:     $\mu(u) := f_1 \rightarrow_r f_2$
7:   for each $v, u \in \nu(\Gamma)$ do
8:     if $\mu(v) = f_1 \rightarrow_r f_2$ and $\mu(u) = f_2 \rightarrow_{r'} f_3$ then
9:       add $v \rightarrow_1 u$ to $\epsilon(\Gamma)$
10:  $G := G \cup \Gamma$
11: return $G$
```

Algorithm 1 combines these two ideas: it runs the chase for each fact in $\mathcal{H}(P)$ then tracks the rule executions and (iii) based on these rule executions it computes a TG. In particular, for each fact f_2 that is derived after executing a rule r over f_1, Algorithm 1 will create a fresh node u and associate it with rule r, lines 4–6. The mapping μ associates nodes with rule executions. Then, the algorithm adds edges between the nodes based on the sequences of rule executions that took place during chasing, lines 7–9.

Algorithm 1 is (implicitly) parameterized by the chase variant. The results below are based on the equivalent chase, as it ensures termination for *FES* programs.

Theorem 10. *For any linear program P that is* FES*,* $\mathsf{tglinear}(P)$ *is a TG for P.*

Algorithm 1 has a double-exponential overhead.

Theorem 11. *The execution time of Algorithm 1 for* FES *programs is double exponential in the input program P. If the arity of the predicates in P is bounded, the execution time is (single) exponential.*

5.1 Minimizing TGs for linear programs

The TGs computed by Algorithm 1 may comprise nodes which can be deleted without compromising query answering. Let us return to Example 1 and to the TG G_1 from Figure 1: we can safely ignore the facts associated with the node u_2 from G_1 and still preserve the answers to all queries over (P_1, B). In this section, we show a technique for minimizing TGs for linear programs.

Our minimization algorithm is based on the following. Consider a TG G for a linear program P, a base instance B of P and the query $Q(X) \leftarrow R(X, Y) \wedge S(Y, Z, Z)$. Assume that there exists a homomorphism from the body of the query into the facts $f_1 = R(c_1, \mathsf{n}_1)$ and $f_2 = S(\mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_2)$ and that $f_1 \in v(B)$ and $f_2 \in u(B)$ with v, u being two nodes of G. Since n_1 is shared among two different facts associated with two different nodes, it is safe to remove u if there is another node $u' \in \nu(G)$ whose instance $u'(B)$ includes a fact of the form $S(\mathsf{n}_1, \mathsf{n}'_2, \mathsf{n}'_2)$.

Equivalently, it is safe to remove u if there exists a homomorphism from $u(B)$ into $u'(B)$ that maps to itself each null occurring both in $u(B)$ and $u'(B)$. Since a null can occur both in $u(B)$ and in $u'(B)$ if u, u' share a common ancestor we can rephrase the previous statement as follows: we can remove $u(B)$ if there exists a homomorphism from $u(B)$ into $u'(B)$ preserving each null (from $u(B)$) that also occurs in some $w(B)$ with w being an ancestor of u in G. We refer to such homomorphisms as *preserving homomorphisms*:

DEFINITION 12. *Let G be a TG for a program P, $u, v \in \nu(G)$ and B be a base instance. A homomorphism from $u(B)$ into $v(B)$ is* preserving, *if it maps to itself each null occurring in some $u'(B)$ with u' being an ancestor of u.*

It suffices to consider only the facts in $\mathcal{H}(P)$ to verify the existence of preserving homomorphisms.

LEMMA 13. *Let P be a linear program, G be an EG for P and $u, v \in \nu(G)$. Then, there exists a preserving homomorphism from $u(B)$ into $v(B)$ for each base instance B, iff there exists a preserving homomorphism from $u(\{f\})$ into $v(\{f\})$, for each fact $f \in \mathcal{H}(P)$.*

From Definition 12 and from Lemma 13 it follows that a node v of a TG can be "ignored" for query answering if there exists a node v' and a preserving homomorphism from $v(\{f\})$ into $v'(\{f\})$, for each $f \in \mathcal{H}(P)$. If the above holds, then we say that v *is dominated by* v'. The above implies a strategy to reduce the size of TGs.

DEFINITION 14. *For a TG G for a linear program P, the EG* $\mathsf{minLinear}(G)$ *is obtained by exhaustively applying the steps: (i) choose a pair of nodes v, v' from G where v is dominated by v', (ii) remove v from $\nu(G)$; and (iii) add an edge $v' \rightarrow_1 u$, for each edge $v \rightarrow_1 u$ from $\epsilon(G)$.*

The minimization procedure described in Definition 14 is correct: given a TG for a linear program P, the output of $\mathsf{minLinear}$ is still a TG for P.

THEOREM 15. *For a TG G for a linear program P,* $\mathsf{minLinear}(G)$ *is a TG for P.*

We present an example demonstrating the TG computation and minimizes techniques described above.

EXAMPLE 16. *Recall Example 1. Since r is the only extensional predicate in P_1, $\mathcal{H}(P_1)$ will include two facts, say $r(c_1, c_2)$ and $r(c_3, c_3)$, where c_1, c_2 and c_3 are constants. Algorithm 1 computes a TG by tracking the rule executions that take place when chasing each fact in $\mathcal{H}(P_1)$. For example, when considering $r(c_1, c_2)$, the graph Γ computed in lines 3–9 will be the TG G_1 from Figure 1(b), where nodes are denoted as u_1, u_2, and u_3.*

Let us now focus on the minimization algorithm. To minimize G_1, we need to identify nodes that are dominated by others. Recall that a node u in G_1 is dominated by a node v, if for each f in $\mathcal{H}(P_1)$, there exists a preserving homomorphism from $u(\{f\})$ into $v(\{f\})$. Based on the above, we can see that u_2 is dominated by u_3. For example, when $B^* = \{r(c_1, c_2)\}$, there exists a preserving homomorphism from $u_2(B^*) = \{R(c_2, c_1, \mathsf{n}_1)\}$ into $u_3(B^*) = \{R(c_2, c_1, c_1)\}$ mapping n_1 to c_1. Since u_2 is dominated by u_3, the minimization process eliminates u_2 from G_1. The result is the TG G_2 from Figure 1(c), since no other node in G_2 is dominated.

6. OPTIMIZING TGS FOR DATALOG

There are cases where we cannot compute instance-independent TG, e.g., for Datalog programs that are not also in *FTG* class. In such cases, we can still create an instance-dependent TG using the procedure outlined in Section 4. In this section, we present two optimizations to this procedure which avoid redundant computations. These optimizations work with Datalog programs; thus also with non-linear rules.

6.1 Eliminating redundant nodes

Our first technique is based on a simple observation. Consider a node v of a TG G. Assume that v is associated with the rule $a(X, Y, Z) \rightarrow A(Y, X)$ with a being extensional. We can see that for each base instance B and each fact $a(\sigma(X), \sigma(Y), \sigma(Z))$ in B, where σ is a variable substitution, the fact $A(\sigma(Y), \sigma(X))$ is in $v(B)$. Equivalently, for each answer σ to $Q(Y, X) \leftarrow a(X, Y, Z)$, a fact $A(\sigma(Y), \sigma(X))$ is associated with $v(B)$. The above can be generalized. Consider a node v of a TG G such that $\mathsf{rule}(v)$ is $\bigwedge_{i=i}^{n} A_i(\mathbf{Y}_i) \rightarrow A(\mathbf{X})$. The facts in $v(B)$ can be obtained by (i) computing the rewriting of the query $Q(\mathbf{X}) \leftarrow \bigwedge_{i=i}^{n} A_i(\mathbf{Y}_i)$ w.r.t. the rules in the ancestors of v up to the extensional predicates; (ii) evaluating the rewritten query over B; and (iii) adding $A(\mathbf{t})$ to $v(B)$, for each answer $\mathbf{t}$ to the rewritten query over B– recall that we denote answers either as substitutions or as tuples, cf. Section 3. We refer to $Q(\mathbf{X}) \leftarrow \bigwedge_{i=i}^{n} A_i(\mathbf{Y}_i)$ as the *characteristic query* of v.

This observation suggests we can use query containment tests to identify nodes that can be safely removed from TGs (and EGs). Intuitively, the naïve algorithm above can be modified so that, at each step i, right after computing G^i, and before computing $G^i(B)$, we eliminate each node u if the EG-guided rewriting over of the characteristic query of u is contained in the EG-guided rewriting of the characteristic query of another node v.

Below, we formalize the notion of EG-rewritings, then we show the correspondence between the answers to EG-rewritings and the facts associated with the nodes, and we finish with an algorithm eliminating nodes from TGs.

DEFINITION 17. *Let v be a node in an EG G for a Datalog program. Let* $\mathsf{rule}(v)$ *be $\bigwedge_{i=1}^{n} A_i \rightarrow R(\mathbf{Y})$. The EG-rewriting of v, denoted as* $\mathsf{rew}(v)$*, is the CQ computed as follows (w.l.o.g. no pair of rules* $\mathsf{rule}(u)$ *and* $\mathsf{rule}(v)$ *with $u, v \in \nu(G)$ and $u \neq v$ shares variables):*

- *form $Q(\mathbf{Y}) \leftarrow R(\mathbf{Y})$; associate $R(\mathbf{Y})$ with v;*

- *repeat the following rewriting step until no intensional atom is left in* $\mathsf{body}(Q)$*: (i) choose an* intensional *atom* $\alpha \in \mathsf{body}(Q)$*; (ii) compute the MGU* θ *of* $\{\mathsf{head}(u), \alpha\}$*, where* u *is the node associated with* α*; (iii) replace* α *in* $\mathsf{body}(Q)$ *with* $\mathsf{body}(u)$ *and apply* θ *on the resulting* Q*; (iv) associate* $\theta(B_j)$ *in* $\mathsf{body}(Q)$ *with the node* w_j*, where* B_j *is the* j*-th atom in* $\mathsf{body}(u)$ *and* $w_j \rightarrow_j u \in \epsilon(G)$*.*

The rewriting algorithm described in Definition 17 is a variant of the rewriting algorithm in [29]. Our difference from [29] is that at each step of the rewriting process, we consider only the rule $\mathsf{rule}(u)$ with u being the node with which α is associated with.

There is a correspondence between the answers to the nodes' EG-rewritings with the facts stored in the nodes.

Lemma 18. *Let* G *be an EG for a Datalog program* P *and* B *be a base instance of* P*. Then for each* $v \in \nu(G)$ *we have:* $v(B)$ *includes exactly a fact* $A(\mathbf{t})$ *with* A *being the head predicate of* $\mathsf{rule}(v)$*, for each answer* $\mathbf{t}$ *to the EG-rewriting of* v *on* B*.*

Our algorithm for removing nodes from EGs is below.

Definition 19. *The EG* $\mathsf{minDatalog}(G)$ *is obtained from an EG* G *for a program* P *by exhaustively applying these steps: for each pair of nodes* u *and* v *such that (i) the depth of* v *is equal or larger than that of* u*, (ii) the predicates of* $\mathsf{head}(\mathsf{rule}(v))$ *and of* $\mathsf{head}(\mathsf{rule}(u))$ *are the same and (iii) the EG-rewriting of* v *is contained in the EG-rewriting of* u*: (a) remove the node* v *from* $\nu(G)$*, and (b) add an edge* $u \rightarrow_j w$*, for each edge* $v \rightarrow_j w$ *occurring in* G*.*

The minimization technique of Definition 19 can be proven sound and to produce a TG with fewest nodes.

Theorem 20. *If* G *is a TG for a Datalog program* P*, then* $\mathsf{minDatalog}(G)$ *is a minimum size TG for* P*.*

Deciding whether a TG of a Datalog program is of minimum size can be proven co-NP-complete. The problem's hardness lies is the necessity of performing query containment tests, carried out via homomorphism tests, which require exponential time on deterministic machines (unless $P = NP$) [20]. This hardness result supports the optimality of $\mathsf{minDatalog}$ in terms of complexity.

Theorem 21. *For a Datalog program* P *and a TG* G *for* P*, deciding whether* G *is a TG of minimum size for* P *is co-NP-complete.*

6.2 A more efficient rule execution strategy

EG-rewritings can be further used to optimize the execution of the rules, as shown in the example below.

Example 22. *Consider the program* P_2

$$a(X) \wedge b(X) \rightarrow A(X) \quad (r_8)$$

$$a'(X) \wedge b'(X) \rightarrow A(X) \quad (r_9)$$

where a*,* a'*,* b *and* b' *are extensional predicates. We denote by* a*,* a'*,* b *and* b' *the relations storing the tuples of the corresponding predicates in the input instance. The data of each relation are shown in Figure 2.*

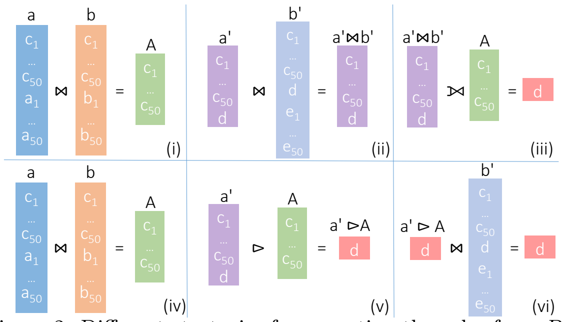

Figure 2: Different strategies for executing the rules from P_2.

The upper part of Figure 2 shows the steps involved when executing r_8 *and* r_9 *using the chase: (i) shows the joins involved when executing* r_8*; (ii)–(iii) show the joins involved when executing* r_9*: (ii) shows the join to compute* $\mathsf{body}(r_9)$ *while (iii) shows the outer join involved when checking whether the conclusions of* r_9 *have been previously derived. Assuming that the cost of executing each join is the cost of scanning the smallest relation, the total cost of the chase is: 100 (step (i)) + 51 (step (ii)) + 50 (step (iii))=201.*

The lower part of Figure 2 shows a more efficient strategy. The execution of r_8 *stays the same (step (iv)), while for* r_9 *we first compute all tuples that are in* a' *but not in* A *(step (v)) and use* $\mathsf{a}' \setminus \mathsf{A}$ *to restrict the tuples instantiating the body of* r_9 *(step (vi)). The intuition is that the tuples of* a' *that are already in* A *will be discarded, so it is not worth considering them when instantiating the body of* r_9*. The total cost of this strategy is: 100 (step (iv)) + 51 (step (v)) + 1 (step (vi))=152.*

Example 22 suggests a way to optimize the execution of the rules, which reduces the cost of instantiating the rule bodies. This is achieved by considering only the instantiations leading to the derivation of new conclusions. Our new rule execution strategy is described below.

Definition 23. *Let* v *be a node of an EG* G *for a Datalog program* P*,* B *be a base instance and* $I \subseteq G(B)$*. Let* $A(\mathbf{X})$ *be the head atom of* $\mathsf{rule}(v)$ *and let* $Q(\mathbf{Y}) \leftarrow \bigwedge_{i=1}^{n} f_i$ *be the EG-rewriting of* v*. The computation of* $v(B)$ under I*, denoted as* $v(B, I)$*, is:*

1. *pick* $m \geq 1$ *atoms* $f_{i_1}, \ldots, f_{i_m}$ *from the body of* Q *whose variables include all variables in* $\mathbf{Y}$ *and form* $Q'(\mathbf{Y}) \leftarrow f_{i_1} \wedge \cdots \wedge f_{i_m}$*;*
2. *compute* $v(B)$ *as in Definition 5, however restrict to homomorphisms* h *for which (i)* $h(\mathbf{X})$ *is an answer to* Q' *on* B *and (ii)* $A(h(\mathbf{X})) \notin I$*.*

To help us understand Definition 23, let us apply it to Example 22. We have $Q'(X) \leftarrow a'(X)$. The antijoin between Q' and A (step (v) of Figure 2) corresponds to restricting to homomorphisms that are answers to Q' (step (2.i) of Definition 23), but are not in I (step (2.ii) of Definition 23). In our implementation, we pick one

Algorithm 2 $\mathsf{TGmat}(P, B)$

```
1: $k := 0$;   $G^0$ is the empty graph;   $I^0 := \emptyset$
2: do
3:   $k := k + 1$;   $I^k := I^{k-1}$
4:   Compute $G^k$ starting from $G^{k-1}$ as in Section 4
5:   $G^k := \mathsf{minDatalog}(G^k)$
6:   for each node $v$ of depth $k$ do
7:     add $v(B, I^{k-1})$ (cf. Definition 23) to $I^k$
8: while $I^k \neq I^{k-1}$
9: return $I^\infty$
```

extensional atom ($m = 1$) in step (1). To pick this atom, we consider each f_i in the body of $\mathsf{rew}(v)$, then compute the join as in step (v) of Example 22 between a subset of the f_i-tuples and the A-tuples in I and finally, choose the f_i leading to the highest join output.

We summarize TG-guided reasoning for Datalog programs in Algorithm 2. Correctness is stated below.

Theorem 24. *For a Datalog program P and a base instance B, $\mathsf{TGmat}(P, B) = Ch(P, B)$.*

7. EVALUATION

We implemented Algorithm 1, TG-guided reasoning over a fixed TG (Def. 5) and Algorithm 2 in a new open-source reasoner called *GLog*. GLog is a fork of VLog [60] that shares the same code for handling the extensional relations while the code for reasoning is entirely novel.

We consider three performance measures: the absolute reasoning runtime, the peak RAM consumption observed at reasoning time, and the total number of triggers. The last measure is added because it reflects the ability of TGs to reduce the number of redundant rule executions and it is robust to most implementation choices.

7.1 Testbed

Systems. We compared against the following systems:

- VLog, as, to our knowledge, is the most efficient system both time- and memory- wise [58, 60];
- the latest public release of RDFox from [1] as it outperforms all chase engines tested against ChaseBench [11]: ChaseFun, DEMo [50], LLunatic [27], PDQ [12] and Pegasus [43];
- the commercial state of the art chase engine COM (name is anonymized due to licensing restrictions);
- Inferray, an RDFS reasoner that uses a columnar layout and that outperforms RDFox [54]; and
- WebPIE, another high-performance RDFS reasoner that runs over Hadoop [59].

We ran VLog, RDFox and the commercial chase engine COM using their most efficient chase implementations. For VLog, this is the restricted chase, while for RDFox and COM this is the Skolem one [11]. All engines ran using a single thread. We could not obtain access to the Vadalog [10] binaries. However, we perform an indirect comparison against Vadalog: we both compare against RDFox using the ChaseBench scenarios from [11].

Benchmarks. To asses the performance of GLog on linear and Datalog scenarios, we considered benchmarks

		#Rules			#IDP's		
Scenario	#EDP's	LI	L	LE	LI	L	LE
Linear and Datalog scenarios							
LUBM	var.	163	170	182	116%	120%	232%
UOBM	2.1	337	561	NA	3.5	3.9	NA
DBpedia	29	4204	9396	NA	31.9	33.1	NA
Claros	13.8	1749	2689	2749	65.8	8.9	548
React.	5.6	259	NA	NA	11.3	NA	NA
ChaseBench scenarios							
S-128	0.15		167			1.9	
O-256	1		529			5.6	
RDFS (ρDF) scenarios							
LUBM	16.7		160			18	
YAGO	18.2		498016			27	

Table 1: The considered benchmarks. #EDP's and #IDP's absolute numbers are stated in millions of facts.

previously used to evaluate the performance of reasoning engines including VLog and RDFox: LUBM [30] and UOBM [41] are synthetic benchmarks; DBpedia [14] (v2014, available online[1]) is a KG extracted from Wikipedia; Claros [51] and Reactome [22] are real-world ontologies[2]. With both VLog and GLog, the KBs are stored with the RDF engine Trident [57].

Linear scenarios. Linear scenarios were created using LUBM, UOBM, DBpedia, Claros and Reactome. For the first four KBs, we considered the linear rules returned by translating the OWL ontologies in each KB using the method described by [61], which was the technique used for evaluating our competitors [45, 58]. This method converts an OWL ontology $\mathcal{O}$ into a Datalog program P_L such that $\mathcal{O} \models P_L$. For instance, the OWL axiom $A \sqsubseteq B$ (concept inclusion) can be translated into the rule $A(X) \rightarrow B(X)$. This technique is ideal for our purposes since this subset is what is mostly supported by RDF reasoners [45]. Here, the subscript "L" stands for "lower bound". In fact, not every ontology can be fully captured by Datalog (e.g., ontologies that are not in OWL 2 RL) and in such cases the translation captures a subset of all possible derivations.

For Reactome, we considered the subset of linear rules from the program used in [60]. The programs for the first four KBs do not include any existential rules while the program for Reactome does. Linear scenarios are suffixed by "LI", e.g., LUBM-LI.

Datalog scenarios. Datalog scenarios were created using LUBM, UOBM, DBpedia and Claros, as Reactome includes non-Datalog rules only. LUBM comes with a generator, which allows controlling the size of the base instance by fixing the number of different universities X in the instance. One university roughly corresponds to 132k facts. In our experiments, we set X to the following values: 125, 1k, 2k, 4k, 8k, 32k, 64k, 128k. This means that our largest KB contains about 17B facts. As programs, we used the entire Datalog programs (linear and non-linear) obtained with [61] as described above. These programs are suffixed by "L". For Claros and LUBM, we

[1] `https://www.cs.ox.ac.uk/isg/tools/RDFox/2014/AAAI/input/DBpedia/ttl/`

[2] Both datasets are available in our code repository.

	VLog		**RDFox**		**COM**		**GLog**					**TG Sizes**		
Scenario	Run.	Mem	Run.	Mem	Run.	Mem	Comp	Reason	w/o cleaning	w/ cleaning	Mem	#N	#E	D
LUBM-LI	1.3	1617	22	2353	18.4	5122	0.007	0.2	0.207	1.1	1674	155	101	6
UOBM-LI	0.3	221	3.9	726	3.3	3570	0.01	0.015	0.025	0.2	219	313	206	9
DBpedia-LI	6.9	2579	44.1	3197	36.3	3767	0.448	0.776	1.224	4.5	2647	12 660	8970	17
Claros-LI	5.6	2870	78.4	3918	72.3	5122	0.006	0.407	0.413	4.8	2586	792	621	23
React.-LI	1.8	1312	12.7	1448	9.9	4479	0.002	0.329	0.329	0.9	1312	386	263	8

Table 2: Linear scenarios. Time is in sec and memory in MB.

	VLog		**RDFox**		**COM**		**GLog Runtime**			**GLog Memory**			**TG Sizes**		
Scenario	Run.	Mem	Run.	Mem	Run.	Mem	No opt	m	m+r	No opt	m	m+r	#N	#E	D
LUBM-L	1.5	324	23	2301	20.4	4479	2.4	2.2	1.0	446	424	264	56	33	4
LUBM-LE	170.5	2725	116.6	3140	115.9	3610	17.3	17.2	16.1	1340	1310	1338	63	43	5
UOBM-L	7.3	1021	10	784	10	4215	2.6	2.4	2.6	335	335	342	527	859	6
DBpedia-L	41.6	827	64.4	3290	198.4	3878	20	19	19	1341	1352	1339	4144	3062	8
Claros-L	431	3170	2512	5491	2373.0	6453	122	118.3	119	6076	6077	6078	438	404	9
Claros-LE	2771.8	11 895	*	*	*	*	1040.8	1012.2	1053.9	48 464	48 474	48 455	1461	3288	9

Table 3: Datalog scenarios. Time is in sec and memory in MB. $*$ denotes timeout after 1h.

used two additional programs, suffixed by "LE", created by [45] as harder benchmarks. These programs extend the "L" ones with extra rules, such as the transitive and symmetric rules for *owl:sameAs*. The relationship between the various rulesets is $LI \subset L \subset LE$.

ChaseBench scenarios. ChaseBench was introduced for evaluating the performance of chase engines [11]. The benchmark comes with four different families of scenarios. Out of these four families, we focused on the iBench scenarios, namely STB-128 and ONT-256 [4] because they come with non-linear rules with existentials that involve many joins and that are highly recursive. Moreover, as we do compare against RDFox which was the top-performing chase engine in [11], we can use these two scenarios to indirectly compare against all the engines considered in [11].

RDFS scenarios. In the Semantic Web, it has been shown that a large part of the inference that is possible under the RDF Schema (RDFS) [16] can be captured into a set of Datalog rules. A number of works have focused on the execution of such rules. In particular, WebPIE and more recently Inferray returned state-of-the-art performance for ρDF – a subset of RDFS that captures its essential semantics. It is interesting to compare the performance of GLog, which is a generic engine not optimized for RDFS rules, against such ad-hoc systems. To this end, we considered YAGO [31] and a LUBM KB with 16.7M triples. As rules for GLog, we translated the ontologies under the ρDF semantics.

Table 1 shows, for each scenario, the corresponding number of rules and EDP-facts as well as the number of IDP-facts in the model of the KB. With LUBM and the linear and Datalog scenarios, the number of IDP-facts is proportional to the input size, thus it is stated as %. For instance, with the "LI" rules, the output is 116%, which means that if the input contains 1M facts, then reasoning returns 1.16M new facts.

Hardware. All experiments except the ones on scalability (Section 7.5) ran on an Ubuntu 16.04 Linux PC with Intel i7 64-bit CPU and 94.1 GiB RAM. For our experiments on scalability, we used a second machine with an Intel Xeon E5 and 256 GiB of RAM due to the large sizes of the KBs. The cost of both machines is <$5k, thus we arguably label them as commodity hardware.

7.2 Results for linear scenarios

Table 2 summarizes the results of our empirical evaluation for the linear scenarios. Recall that when a program is linear and *FES* it admits a finite TG which can be computed prior to reasoning using tglinear (Algorithm 1) and minimized using minLinear from Definition 14. Columns two to seven show the runtime and the peak memory consumption for VLog, RDFox and the commercial engine COM. The remaining columns show results related to TG-guided reasoning. Column *Comp* shows the time to compute and minimize a TG using tglinear and minLinear. Column *Reason* shows the time to reason over the computed TG given a base instance (i.e., apply Definition 5). Column *w/o cleaning* shows the total runtime if we do not filter out redundant facts at reasoning time, while column *w/ cleaning* shows the total runtime if we additionally filter out redundancies at the end and collectively for all the rules. Notice that in both cases the total runtime includes the time to compute and reason over the TG (columns *Comp* and *Reason*). Column *Mem* shows the peak memory consumption. As we will explain later, in the case of linear rules, the memory consumption in GLog is the same both with and without filtering out redundant facts. Finally, the last three columns #N, #E, and D show the number of nodes, edges, and the depth (i.e., length of the longest shortest path) in the resulting TGs.

We summarize two main conclusions of our analysis.

C1: TGs outperform the chase in terms of runtime and memory. The runtime improvements over the chase vary from multiple orders of magnitude (w/o filtering of redundancies) to almost two times (w/o filtering). When redundancies are discarded, the vast improvements are attributed to *structure sharing*, a technique which is also implemented in VLog.

Structure sharing is about reusing the same columns to store the data of different facts. For example, consider the rule $R(X, Y) \rightarrow S(Y, X)$. Instead of creating different S- and R-facts, we can simply add a pointer from the first column of R to the second column of S and a pointer from the second column of R to the first column of S. When a rule is linear, both VLog and GLog perform structure sharing and, hence, do not allocate extra memory to store the derived facts. Apart

	VLog		RDFox		COM		GLog		TG Sizes		
S	Run.	Mem	Run.	Mem	Run.	Mem	Run.	Mem	#N	#E	D
S	0.5	1350	13.4	1747	10	5217	0.2	1266	192	0	0
O	2.3	4930	49	3997	35	6340	1	4930	577	65	3

Table 4: ChaseBench scenarios (S=STB-128,O=ONT-256). Runtime in sec, memory in MB.

Scenario	**VLog**	**GLog**		
		no opt	m	m+r
LUBM-L	38	32	29	25
LUBM-LE	239	100	98	93
UOBM-L	47	9	8	8
DBpedia-L	79	63	61	47
Claros-L	286	218	195	185
Claros-LE	1099	1072	1049	1039

Table 5: #Triggers (millions), Datalog scenarios.

	WebPIE		**Inferray**		**GLog**		**TG Sizes**		
S	Run.	Mem	Run.	Mem	Run.	Mem	#N	#E	D
L	338	1124	39	7000	0.3	186	53	25	4
Y	745	1075	116.6	14 000	25	1603	1.07M	888k	20

Table 6: RDFS scenarios (L=LUBM,Y=YAGO). Runtime in sec, memory in MB.

	133M	267M	534M	1B	2B	4B	8B	17B
Run.	13	27	56	203	226	520	993	2272
Mem	1	3	6	23	34	49	98	174
#IDP's	160M	320M	641M	1B	2B	5B	10B	20B

Table 7: Scalability results. Runtime in sec, memory in GB.

from the obvious benefits memory-wise, structure sharing also provides benefits in runtime as it allows deriving new facts without actually executing rules. The above, along with the fact that the facts (redundant or not) are not explicitly materialized in memory makes GLog very efficient time-wise.

When redundancies are filtered out, GLog still outperforms the other engines: it is multiple orders of magnitude faster than RDFox and COM and almost two times faster than VLog (Reactome-LI). The performance improvements are attributed to a more efficient strategy for filtering out redundancies: TGs allow filtering out redundancies after reasoning has terminated, in contrast to the chase, which is forced to filter out redundancies right after the derivation of new facts. This strategy is more efficient because we can use a single n-way join rather than multiple binary joins to remove redundancies.

With regards to memory, GLog has similar memory requirements with VLog, while it is much more memory efficient than RDFox and the commercial engine COM.
C2: The TG computation overhead is small. The time to compute and minimize a TG in advance of reasoning is only a small fraction of the total runtime, see Table 2. We argue that even if this time was not negligible, TG-guided reasoning would still be beneficial: first, once a TG is computed reasoning over it is multiple times faster than the chase and, second, the same TG can be used to reason over the same rules independently of any changes in the database.

7.3 Results for Datalog and ChaseBench

Table 3 summarizes our results on generic (linear and non-linear) Datalog rules. The last nine columns show results for TGmat (Algorithm 2). To assess the impact of minDatalog and ruleExec, the rule execution strategy from Definition 23, we ran TGmat as follows: without minDatalog or ruleExec, column *No opt*; with minDatalog, but without ruleExec, column *m*; with both minDatalog and ruleExec, column *m+r*. The total runtime in the last two cases includes the runtime overhead of minDatalog and ruleExec. The last three columns report the number of nodes, edges, and depth of the computed TGs when both minDatalog or ruleExec are employed. Table 4 shows results for ChaseBench while Table 5 shows the number of triggers for the Datalog scenarios for VLog and GLog (we could not extract this information for RDFox and COM).

We summarize the main conclusions of our analysis.
C3: TGs outperform the chase in terms of runtime and memory. Even without any optimizations, GLog is faster than VLog, RDFox and COM in all but one case. With regards to VLog, GLog is up to nine times faster in the Datalog scenarios (LUBM-LE) and up to two times faster in ChaseBench (ONT-256). With regards to RDFox, GLog is up to 20 times faster in the Datalog scenarios (Claros-L) and up to 67 times faster in ChaseBench (ONT-256). When all optimizations are enabled GLog outperforms the competitors in all cases.

We have observed that the bulk of the computation lies in the execution of the joins involved when executing few expensive rules. In GLog, joins are executed more efficiently than in the other engines (GLog uses only merge joins), since the considered instances are smaller –recall that in TGs, the execution of a rule associated with a node v considers only the instances of the parents of v. Due to the above, the optimizations do not decrease the runtime considerably. The only exception is LUBM-L, where the optimizations half the runtime.

Continuing with the optimizations, their runtime overhead is very low: it is 9% of the total runtime (LUBM-L), while the overhead of minDatalog is less than 1% of the total runtime (detailed results are in the appendix). We consider this overhead to be acceptable, since, as we shall see later, the optimizations considerably decrease the number of triggers, a performance measure which is robust to hardware and most implementation choices.

It is important to mention that GLog implements the technique in [33] for executing transitive and symmetric rules. The improvements brought by this technique are most visible with LUBM-LE where the runtime increases from 18s with this technique to 71s without it. Other improvements occur with UOBM-L and DBpedia-L (69% and 57% resp.). In any case, even without this technique, GLog remains faster than its competitors in all cases.

Last, the ChaseBench experiments allow us to compare against Vadalog. According to [10], Vadalog is three times faster than RDFox on STB-128 and ONT-256. Our empirical results show that GLog brings more substantial runtime improvements: GLog is from 49 times to more than 67 times faster than RDFox in those scenarios.

With regards to memory, the memory footprint of GLog again is comparable to that of VLog and it is lower than that of RDFox and of COM.

C4: TGs outperform the chase in terms of the number of triggers. Table 5 shows that the total number of triggers and, hence, the amount of redundant computations, is considerably lower than the total number of triggers in VLog even when the optimizations are disabled. This is due to the different approaches employed to filter out redundancies: VLog filters out redundancies right after the execution of each rule [58], while GLog performs this filtering after each round. When the optimizations are enabled, the number of triggers further decreases: in the best case (DBpedia-L), GLog computes 1.69 times fewer triggers (79M/47M).

7.4 Results for RDFS scenarios

Table 6 summarizes the results of the RDFS scenarios where GLog is configured with both optimizations enabled. We can see that GLog is faster than both RDFS engines. With regards to Inferray, GLog is two orders of magnitude faster on LUBM and more than four times faster on YAGO. With regards to WebPIE, GLog is three orders of magnitude faster on LUBM and more than 32 times faster on YAGO. With regards to memory, GLog is more memory efficient in all but one cases.

7.5 Results on scalability

We used the LUBM benchmark to create several KBs with 133M, 267M, 534M, 1B, 2B, 4B, 8B, and 17B facts respectively. Table 7 summarizes the performance with the Datalog program LUBM-L. Columns are labeled with the size of the input database. Each column shows the runtime, the peak RAM memory consumption, and the number of derived facts for each input database. We can see that GLog can reason with up to 17B facts in less than 40 minutes without resorting to expensive hardware. We are not aware of any other centralized reasoning engine that can scale up to such an extent.

8. RELATED WORK

One approach to improve the reasoning performance is to parallelize the execution of the rules. RDFox proposes a parallelization technique for Datalog materialization with mostly lock-free data insertion. Parallelization has been also been studied for reasoning over RDFS and OWL ontologies. For example, WebPIE encodes the materialization process into a set of MapReduce programs while Inferray executes each rule on a dedicated thread. Our experiments show that GLog outperforms all these engines in a single-core scenario. This motivates further research on parallelizing TG-based materialization.

A second approach is to reduce the number of logically redundant facts by appropriately ordering the rules. In [59], the authors describe a rule ordering that is optimal *only* for a fixed set of RDFS rules. In contrast, we focus on generic programs. ChaseFun [15] proposes a new rule ordering technique that focuses on *equality generating dependencies*. Hence, it is orthogonal to our approach. In a similar spirit, the rewriting technique from [33] targets transitive and symmetric rules. GLog applies this technique by default to improve the performance, but our experiments show it outperforms the state of the art even without this optimization.

To optimize the execution of the rules themselves, most chase engines rely on external DBMSs or employ state of the art query execution algorithms: LLunatic [27], PDQ and ChaseFun run on top of PostgreSQL; RDFox and VLog implement their own in-memory rule execution engine. However, none of these engines can effectively reduce the instances over which rules are executed as TGs do. Other approaches involve exploring columnar memory layouts as in VLog and Inferray to reduce memory consumption and to guarantee sequential access and efficient sort-merge join inference.

Orthogonal to the above is the work in [10], which introduces a new chase variant for materializing KBs of warded Datalog programs. Warded Datalog is a class of programs not admiring a finite model for any base instance. The variant works as the restricted chase does but replaces homomorphism with isomorphism checks. As a result, the computed models become bigger. An implementation of the warded chase is also introduced in [10] which focuses on decreasing the cost of isomorphism checks. The warded chase implementation does not apply any techniques to detect redundancies in the offline fashion as we do for linear rules, or to reduce the execution cost of Datalog rules as we do in Section 6.

We now turn our attention to the applications of materialization in goal-driven query answering. Two well-known database techniques that use materialization as a tool for goal-driven query answering are *magic sets* and *subsumptive tabling* [8, 9, 53, 55]. The advantage of these techniques over the *query rewriting* ones, which are not based on materialization, e.g., [6, 19, 29], is the full support of Datalog. The query rewriting techniques can support Datalog of bounded recursion only. Beyond Datalog, materialization-based techniques have been recently proposed for goal-driven query answering over KBs with equality [13], as well as for probabilistic KBs [56], leading in both cases to significant improvements in terms of runtime and memory consumption. The above automatically turns TGs to a very powerful tool to also support query-driven knowledge exploration.

TGs are different from acyclic graphs of rule dependencies [7]: the former contain a single node per rule while TGs do not.

9. CONCLUSION

We introduced a novel approach for materializing KBs that is based on traversing acyclic graphs of rules called TGs. Our theoretical analysis and our empirical evaluation over well-known benchmarks show that TG-guided reasoning is a more efficient alternative to the chase, since it effectively overcomes all of its limitations.

Future research involves studying the problem of updating TGs in response to KB updates, as well as extending TGs to materialize distributed KBs.

10. REFERENCES

[1] RDFox public release. `https://github.com/dbunibas/chasebench/tree/master/tools/rdfox`. Accessed: 2020-11-10.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and Implementation of the LogicBlox System. In *SIGMOD*, pages 1371–1382, 2015.

[4] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench Integration Metadata Generator. In *VLDB*, page 108–119, 2015.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, USA, 1999.

[6] J. Baget, M. Leclère, M. Mugnier, S. Rocher, and C. Sipieter. Graal: A Toolkit for Query Answering with Existential Rules. In *RuleML*, 2015.

[7] J. Baget, M. Leclère, M. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9-10):1620–1654, 2011.

[8] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS*, pages 1–15, 1986.

[9] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3,4):255–299, 1991.

[10] L. Bellomarini, E. Sallinger, and G. Gottlob. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB*, 11(9):975–987, 2018.

[11] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, pages 37–52, 2017.

[12] M. Benedikt, J. Leblay, and E. Tsamoura. PDQ: Proof-driven query answering over web-based data. In *VLDB*, page 1553–1556, 2014.

[13] M. Benedikt, B. Motik, and E. Tsamoura. Goal-driven query answering for existential rules with equality. In *AAAI*, pages 1761 – 1770, 2018.

[14] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellman. DBpedia - A crystallization point for the Web of Data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[15] A. Bonifati, I. Ileana, and M. Linardi. Functional Dependencies Unleashed for Scalable Data Exchange. In *SSDBM*, 2016.

[16] D. Brickley, R. V. Guha, and B. McBride. Rdf schema 1.1. *W3C recommendation*, 25:2004–2014, 2014.

[17] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Int. Res.*, 48(1):115–174, 2013.

[18] A. Calì, G. Gottlob, and T. Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14:57–83, 2012.

[19] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[20] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[21] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M. S. Miller, F. Och, C. Olston, and F. Pereira. Yedalog: Exploring knowledge at scale. In *SNAPL*, pages 63–78, 2015.

[22] D. Croft, A. F. Mundo, R. Haw, M. Milacic, J. Weiser, G. Wu, M. Caudy, P. Garapati, M. Gillespie, M. R. Kamdar, et al. The reactome pathway knowledgebase. *Nucleic acids research*, 42(D1):D472–D477, 2013.

[23] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay. *Neural-symbolic learning systems: foundations and applications*. Perspectives in neural computing. Springer, 2002.

[24] S. Delivorias, M. Leclère, M. Mugnier, and F. Ulliana. On the k-Boundedness for Existential Rules. In *RuleML+RR*, pages 48–64, 2018.

[25] A. Deutsch, A. Nash, and J. B. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.

[26] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[27] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's All Folks! LLUNATIC Goes Open Source. In *VLDB*, page 1565–1568, 2014.

[28] G. Gottlob, G. Orsi, and A. Pieris. Ontological query answering via rewriting. In *ADBIS*, pages 1–18, 2011.

[29] G. Gottlob, G. Orsi, and A. Pieris. Query Rewriting and Optimization for Ontological Databases. *ACM TODS*, 39(3):25:1–25:46, 2014.

[30] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3), 2011.

[31] J. Hoffart, F. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[32] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A.-C. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. Knowledge Graphs. *arXiv:2003.02320 [cs]*, 2020. arXiv: 2003.02320.

[33] P. Hu, B. Motik, and I. Horrocks. Modular

materialisation of datalog programs. In *AAAI*, pages 2859–2866, 2019.

[34] P. Hu, J. Urbani, B. Motik, and I. Horrocks. Datalog Reasoning over Compressed RDF Knowledge Bases. In *CIKM*, pages 2065–2068, 2019.

[35] N. Konstantinou, M. Koehler, E. Abel, C. Civili, B. Neumayr, E. Sallinger, A. A. Fernandes, G. Gottlob, J. A. Keane, L. Libkin, and N. W. Paton. The VADA Architecture for Cost-Effective Data Wrangling. In *SIGMOD*, pages 1599–1602, 2017.

[36] B. Kruit, P. A. Boncz, and J. Urbani. Extracting novel facts from tables for knowledge graph completion. In *ISWC*, pages 364–381, 2019.

[37] B. Kruit, H. He, and J. Urbani. Tab2know: Building a knowledge base from tables in scientific papers. In *ISWC*, pages 349–365. Springer, 2020.

[38] M. Leclère, M. Mugnier, M. Thomazo, and F. Ulliana. A Single Approach to Decide Chase Termination on Linear Existential Rules. In *ICDT*, pages 18:1–18:19, 2019.

[39] M. Leclère, M. Mugnier, and F. Ulliana. On bounded positive existential rules. In *DL*, 2016.

[40] J. Lee, T. Hwang, J. Park, Y. Lee, B. Motik, and I. Horrocks. A Context-Aware Recommendation System for Mobile Devices. In *ISWC*, pages 380–382, 2020.

[41] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL Ontology Benchmark. In *ESWC*, pages 125–139, 2006.

[42] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):45–5469, 1979.

[43] M. Meier. The backchase revisited. *VLDB J.*, 23(3):495–516, 2014.

[44] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. OWL 2 web ontology language profiles. *W3C recommendation*, 27:61, 2009.

[45] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *AAAI*, pages 129–137, 2014.

[46] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch. Datalography: Scaling datalog graph analytics on graph processing systems. In *IEEE International Conference on Big Data*, pages 56–65, 2016.

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

[48] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor. Industry-scale Knowledge Graphs: Lessons and Challenges. *Commun. ACM*, 62(8):36–43, July 2019.

[49] A. Onet. The chase procedure and its applications in data exchange. In *DEIS*, pages 1–37, 2013.

[50] R. Pichler and V. Savenkov. DEMo: Data Exchange Modeling Tool. In *VLDB*, pages 1606–1609, 2009.

[51] S. Rahtz, A. Dutton, D. Kurtz, G. Klyne, A. Zisserman, and R. Arandjelovic. CLAROS—Collaborating on Delivering the Future of the Past. In *DH*, pages 355–357, 2011.

[52] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[53] D. Sereni, P. Avgustinov, and O. de Moor. Adding Magic to an Optimising Datalog Compiler. In *SIGMOD*, pages 553–566, 2008.

[54] J. Subercaze, C. Gravier, J. Chevalier, and F. Laforest. Inferray: Fast in-Memory RDF Inference. *Proceedings of the VLDB Endowment*, 9(6):468–479, 2016.

[55] K. T. Tekle and Y. A. Liu. More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets. In *SIGMOD*, pages 661–672, 2011.

[56] E. Tsamoura, V. Gutiérrez-Basulto, and A. Kimmig. Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs. In *AAAI*, pages 10284–10291, 2020.

[57] J. Urbani and C. Jacobs. Adaptive Low-level Storage of Very Large Knowledge Graphs. In *WWW*, pages 1761–1772, 2020.

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

[59] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In *ESWC*, pages 213–227, 2010.

[60] J. Urbani, M. Krötzsch, C. Jacobs, I. Dragoste, and D. Carral. Efficient Model Construction for Horn Logic with VLog. In *IJCAR*, pages 680–688, 2018.

[61] Y. Zhou, B. Cuenca Grau, I. Horrocks, Z. Wu, and J. Banerjee. Making the Most of your Triple Store: Query Answering in OWL 2 using an RL Reasoner. In *WWW*, pages 1569–1580, 2013.

Scenario	VLog	GLog
LUBM-LI	34 346	35 093
UOBM-LI	7625	6718
DBpedia-LI	61 134	115 150
Claros-LI	129 098	134 800
Reactome-LI	17 120	23 218

(a) #Triggers for VLog and GLog on the linear scenarios.

Scenario	m	r
LUBM-L	0.0005	0.16
LUBM-LE	0.0007	0.16
UOBM-L	0.08	0.02
DBpedia-L	0.05	0.5
Claros-L	0.03	2.8
Claros-LE	0.3	15.2

(b) Cost of optimizations.

Table 8: Additional Experimental Results

	VLog		**RDFox**		**COM**		**GLog Runtime**			**GLog Memory**		
Scenario	Runtime	Memory	Runtime	Memory	Runtime	Memory	No opt	m	m+r	No opt	m	m+r
LUBM-L	1.5	324	23	2301	20.4	4479	2.5	2.2	1	446	424	265
LUBM-LE	170.5	2725	116.6	3140	115.9	3610	71.1	68.8	67.7	2880	2688	2695
UOBM-L	7.3	1021	10	784	10	4215	4.4	6.3	6.3	506	590	590
DBpedia-L	41.6	827	64.4	3290	198.4	3878	31.4	32	31.1	2335	2319	2313
Claros-L	431	3170	2512	5491	2373.0	6453	128.6	125.6	126.7	5954	5957	5958
Claros-LE	2771.8	11 895	*	*	*	*	1104.3	1094	1106.3	48 246	48 251	48 223

Table 9: Datalog scenarios. GLog is ran without the optimization from [33]. Time is in sec and memory in MB.

	VLog		**RDFox**		**COM**		**GLog**	
Scenario	Runtime	Memory	Runtime	Memory	Runtime	Memory	Runtime	Memory
STB-128	0.5	1350	13.4	1747	10	5217	0.2	1266
ONT-256	2.3	4930	49	3997	35	6340	1	4929

Table 10: ChaseBench scenarios. GLog is ran without the optimization from [33]. Time is in sec and memory in MB.

	VLog		**WebPIE**		**Inferray**		**GLog Runtime**			**GLog Memory**		
Scenario	Runtime	Memory	Runtime	Memory	Runtime	Memory	No opt	m	m+r	No opt	m	m+r
LUBM	0.1	189	353	200	23	2000	0.4	0.4	0.3	186	187	181
YAGO	163	3192	808	200	116.6	14 000	20	23	25	1438	1602	1600

Table 11: RDFS scenarios. GLog is ran without the optimization from [33]. Time is in sec and memory in MB.

APPENDIX

A. ADDTIONAL EXPERIMENTAL RESULTS

Number of triggers in the linear scenarios. Table 8a summarizes the number of triggers for the linear scenarios. We can see that the number of triggers in GLog is often higher than in VLog. This is due to the fact that GLog does not eliminate redundancies right at their creation. However, these redundancies are harmless: due to structure sharing these redundant facts are not explicitly materialized in memory and hence, they do not slow down the runtime.

Cost of optimizations. Table 8b summarizes the cost of optimizations for the Datalog scenarios. Recall that the optimizations in Section 6 are not applicable to ChaseBench as the rules have existential variables. Column m shows the total runtime cost of minDatalog, while column r shows the total runtime cost of ruleExec.

Impact on rewriting on GLog. Tables 9, 10 and 11 summarize the performance of GLog when disabling the optimization from [33]. To ease the presentation, we also copy the results of the competitor engines on the same benchmarks from Tables 3, 4 and 6. We can see that the only scenario whose performance degrades considerably is LUBM-LE shown in Table 9. Even in this case though, the performance of GLog is still better than the performance of its competitors: it is twice as fast as VLog, RDFox and COM in most scenarios and more than an order of magnitude faster than RDFox and COM in Claros-L.

Running RDFox in multiple threads. Tables 12, 13 and 14 show the runtime performance of RDFox when increasing the number of threads from 1 to 8 and 16. For completeness, we also copy the runtime of GLog using a *single thread* on the same scenarios from Tables 2, 3 and 4. We can see that the runtime of RDFox drops considerably when using 16 threads. However, it is still higher than the runtime of GLog in all cases except UOBM-L, where RDFox's runtime is 1.6s, while GLog's runtime is 2.6s. In the other scenarios, GLog is up to 7.8 times faster than RDFox (ONT-256).

Scenario	RDFox Runtime (1 thread)	RDFox Runtime (8 threads)	RDFox Runtime (16 threads)	RDFox Runtime (32 threads)	GLog Runtime w/o cleaning	GLog Runtime w/ cleaning
LUBM-LI	22	4.7	3.7	4.3	0.010	1.1
UOBM-LI	3.9	0.9	0.8	1.6	0.012	0.2
DBpedia-LI	44.1	12.8	10.8	14.7	0.980	4.5
Claros-LI	78.4	16.1	12	14	0.051	4.8
React.-LI	12.7	2.8	2.3	2.9	0.131	0.9

Table 12: Linear scenarios. RDFox is ran in one, eight, 16 threads and 32 threads. GLog is ran in a single thread. Time in sec.

Scenario	RDFox Runtime (1 thread)	RDFox Runtime (8 threads)	RDFox Runtime (16 threads)	RDFox Runtime (32 threads)	GLog Runtime No opt	GLog Runtime m	GLog Runtime m+r
LUBM-L	23	4.8	3.8	4.3	2.4	2.2	1.0
LUBM-LE	116.6	20.9	16.2	17.2	17.3	17.2	16.1
UOBM-L	10	2	1.6	2.4	2.6	2.4	2.6
DBpedia-L	64.4	29	23.9	34	20	19	19
Claros-L	2512	296	171.1	244.9	122	118.3	119
Claros-LE	*	*	*	*	1040.8	1012.2	1053.9

Table 13: Datalog scenarios. RDFox is ran in one, eight, 16 threads and 32 threads. GLog is ran in a single thread. Time in sec. * denotes runtime exception.

Scenario	RDFox Runtime (1 thread)	RDFox Runtime (8 threads)	RDFox Runtime (16 threads)	RDFox Runtime (32 threads)	**GLog** no opt Runtime
STB-128	13.4	2.9	2.3	3.1	0.2
ONT-256	49	10	7.8	10	1

Table 14: ChaseBench scenarios. RDFox is ran in 1, 8, 16 threads and 32 threads. GLog is ran in a single thread. Time in sec.

B. ADDITIONAL DEFINITIONS

We provide some definitions that will be useful for the proofs in the next section.

For a KB K where $Ch(K)$ is defined, the depth $\mathsf{d}(t)$ of a term t in $Ch(K)$ is defined as follows: if $t \in \mathsf{Consts}$, then $\mathsf{d}(t) = 1$; otherwise, if t is a null of the form $n_{r,h,z}$, then $\mathsf{d}(t) = max(\mathsf{d}(t_1), \dots, \mathsf{d}(t_n)) + 1$, where $\{t_1, \dots, t_n\}$ are all terms in the range of h.

Next, we recapitulate the definitions of some known classes of programs.

Definition 25. *Consider a program P and some $k \geq 0$.*

- *P is* Finite Expansion Set *(*FES*), if for each base instance B, the KB (P, B) has a terminating chase.*
- *P is* k-Term Depth Bounded *(*k-TDB*), if for each base instance B, each $i \geq 0$ and each term t in $Ch^i(P, B)$, $\mathsf{d}(t) \leq k$. P is* TDB*, if it is k-TDB.*
- *P is* Finite Order Rewritable *(*FOR*), if, for each BCQ Q, there is a union of BCQs (UBCQs) Q' such that, for each base instance B, we have that $(P, B) \models Q$ iff $B \models Q'$.*

C. PROOFS FOR RESULTS IN SECTION 4

For a program P, we refer to the graph computed by applying the base and the inductive steps from Section 4 as the *level-k full* EG for P, and denote it as Φ^k_P. Below, we show that reasoning via a level-k full EG produces logically equivalent facts with the k-th round of the chase when the chase is applied in a breadth-first fashion:

Theorem 26. *For a program P, a base instance B and a $k \geq 0$, $Ch^k(P, B) \equiv \Phi^k_P(B)$.*

Proof. Let $P = \{r_1, \dots, r_n\}$. Let $\Phi^k_P = G^k = (V^k, E^k)$ and let $\Phi^{k+1}_P = G^{k+1} = (V^{k+1}, E^{k+1})$. Let $I^k = Ch^k((P, B))$ and let $J^k = G^k(B)$.

($\Leftarrow$) To prove the claim, we show the following property, for each $k \geq 0$:

- ψ. there exists a homomorphism $g^k : J^k \rightarrow I^k$.

For $k = 0$, ψ holds, since $I^0 = G^0(B) = B$. For $k + 1$ and assuming that ψ holds for k, the proof proceeds as follows. Let $v^{k+1}_1, \dots, v^{k+1}_m$ be all nodes in V^{k+1} of depth $k + 1$. For each $1 \leq \iota \leq m$, let $\mathsf{rule}(v^{k+1}_\iota) = r_\iota$ and let $v^{k+1}_\iota(B) = \{F_{\iota,1}, \dots, F_{\iota,n_\iota}\}$. Since each rule has a single atom in its head, it follows from Definition 5 that for each $1 \leq \iota \leq m$ and each $1 \leq \kappa \leq n_\iota$, there exists a homomorphism $h_{\iota,\kappa}$, such that $h_{\iota,\kappa_s}(\mathsf{head}(r_\iota)) = F_{\iota,\kappa}$. We distinguish the cases, for each rule r_ι, for $1 \leq \iota \leq m$:

- r_ι is an extensional rule. Hence, for each each $1 \leq \kappa \leq n_\iota$, $h_{\iota,\kappa}$ is a homomorphism from $\mathsf{body}(r_\iota)$ into B.
- r_ι is not an extensional rule. WLOG, assume that r_ι comprises only IDP-atoms in its body. Since G^{k+1} is a full EG, it follows that for each $1 \leq \lambda \leq e_\iota$, there exists an edge $u_{\iota,\lambda} \rightarrow_{\iota,\lambda} v^{k+1}_\iota$ in E^{k+1}. Due to the above, and due to Definition 5, it follows that for each $1 \leq \kappa \leq n_\iota$, $h_{\iota,\kappa}$ is a homomorphism from $\mathsf{body}(r_\iota)$ into $\bigcup^{e_\iota}_{\lambda} u_{\iota,\lambda}(B)$.

Let $N^k = \bigcup^{m}_{\iota} \bigcup^{n_\iota}_{\kappa} h_{\iota,\kappa_s}(\mathsf{head}(r_\iota))$. We further distinguish the cases:

- for each $1 \leq \iota \leq m$ and each $1 \leq \kappa \leq n_\iota$, $F_{\iota,\kappa} \in G^k(B)$. Then ψ trivially holds.

- there exists $1 \leq \iota \leq m$ and $1 \leq \kappa \leq n_\iota$, such that $F_{\iota,\kappa} \notin G^k(B)$. Since ψ holds for k and due to g^k, it follows that for each $1 \leq \iota \leq m$ and each $1 \leq \kappa \leq n_\iota$, there exists a homomorphism $\chi_{\iota,\kappa} = h_{\iota,\kappa} \circ g^k$ from $\mathsf{body}(r_\iota)$ into I^k. Due to the above, for each $1 \leq \iota \leq m$ and each $1 \leq \kappa \leq n_\iota$, there also exists a homomorphism $\omega_{\iota,\kappa}$ from $h_{\iota,\kappa_s}(\mathsf{head}(r_\iota))$ into $\chi_{\iota,\kappa_s}(\mathsf{head}(r_\iota))$, mapping each $h_{\iota,\kappa_s}(n_z)$ to $\chi_{\iota,\kappa_s}(n'_z)$, where $h_{\iota,\kappa_s}(z) = n_z$ and $\chi_{\iota,\kappa_s}(z) = n'_z$, for each existentially quantified variable z occurring in r. Due to the above and since for each $1 \leq \iota \leq m$ and each $1 \leq \kappa \leq n_\iota$, $h_{\iota,\kappa_s}(c) = g^k(c)$, there also exists a homomorphism from N^k into $M^k = \bigcup_{\iota}^{m} \bigcup_{\kappa}^{n_\iota} \chi_{\iota,\kappa_s}(\mathsf{head}(r_\iota))$ and hence from $I^k \cup N^k$ into $J^k \cup M^k$. The above shows that ψ holds for $k+1$.

($\Rightarrow$) To prove the claim, we show the following property, for each $k \geq 0$:

- ϕ. there exists a homomorphism $g^k : I^k \to J^k$ mapping each $F \in I^k$ to some fact $F' \in J^k$ of the same depth.

For $k = 0$, ϕ holds, since $I^0 = G^0(B) = B$. For $k+1$ and assuming that ϕ holds for k the proof proceeds as follows. For each $1 \leq i \leq n$ and each $1 \leq j \leq n_i$, let $h_{i,j}$ be the j-th homomorphism from the body of rule r_i into I^k, where $h_{i,j}(\mathsf{body}(r_i))$ comprises at least one fact of depth k. Due to the inductive hypothesis, we know that for each $1 \leq i \leq n$ and $1 \leq j \leq n_i$, there exists a homomorphism $\chi_{i,j} = h_{i,j} \circ g^k$ from $\mathsf{body}(r_i)$ into J^k. We distinguish the cases, for each rule r_i, for $1 \leq i \leq n$:

- r_ι is an extensional rule. Hence, $I^k = B$.
- r_ι is not an extensional rule. WLOG, assume that r_ι comprises only IDP-atoms in its body. Let $u^1_{i,j}, \dots, u^{|\mathsf{body}(r_i)|}_{i,j}$ be the nodes in V^k, such that for each $1 \leq j \leq n_i$ and each $1 \leq l \leq |\mathsf{body}(r_i)|$, the l-th fact in $\chi_{i,j}(\mathsf{body}(r_i))$ belongs to $u^l_{i,j}(B)$. Since ϕ holds for k, it follows that for each $1 \leq j \leq n_i$, some fact in $\chi_{i,j}(\mathsf{body}(r_i))$ is of depth k. Hence some node in $u^1_{i,j}, \dots, u^{|\mathsf{body}(r_i)|}_{i,j}$ is of depth k. Since G^{k+1} is a full EG for P, it follows that for each $1 \leq j \leq n_i$ and each $1 \leq l \leq |\mathsf{body}(r_i)|$, $u^l_{i,j} \to_j v \in E^{k+1}$, where $\mathsf{rule}(v) = r_i$.

Due to the above and due to Definition 5, it follows that for each $1 \leq i \leq n$ and each $1 \leq j \leq n_i$, we have $\chi_{i,j_s}(\mathsf{head}(r_i)) \in G^{k+1}(B)$. Since for each $1 \leq i \leq n$ and each $1 \leq j \leq n_i$, there exists a homomorphism from g^k from $h_{i,j}(\mathsf{body}(r_i))$ into $\chi_{i,j}(\mathsf{body}(r_i))$, it follows that for each $1 \leq i \leq n$ and each $1 \leq j \leq n_i$, there also exists a homomorphism $\omega_{i,j}$ from $h_{i,j_s}(\mathsf{head}(r_i))$ into $\chi_{i,j_s}(\mathsf{head}(r_i))$, mapping each $h_{i,j_s}(n_z)$ to $\chi_{i,j_s}(n'_z)$, where $h_{i,j_s}(z) = n_z$ and $\chi_{i,j_s}(z) = n'_z$, for each existentially quantified variable z occurring in r. Due to the above and since for each $1 \leq i \leq n$ and each $1 \leq j \leq n_i$, $h_{i,j_s}(c) = g^k(c)$, there also exists a homomorphism from $N^k = \bigcup_{i}^{n} \bigcup_{j}^{n_i} h_{i,j_s}(\mathsf{head}(r_i))$ into $M^k = \bigcup_{i}^{n} \bigcup_{j}^{n_i} \chi_{i,j_s}(\mathsf{head}(r_i))$ and hence from $I^k \cup N^k$ into $J^k \cup M^k$. The above shows that the induction holds for $k+1$. $\square$

Theorem 7. *The following hold: P is* FTG *iff it is* BDD*; and P is* TDB $\cap$ FOR *iff it is* BDD.

Proof. The proof is based on the proofs of Theorem 27 and Theorem 28. $\square$

Theorem 27. *P is* FTG *iff it is* BDD.

Proof. $\Longrightarrow$ If P is *FTG*, then there is some finite TG $G = (V, E)$ for this program. We proceed to show that P is *k-BDD* with k the depth of G. More precisely, we show by contradiction that $Ch^k(P, B) \models Ch^{k+1}(P, B)$ for any given base instance B.

1. Suppose for a contradiction that $Ch^k(K) \not\models Ch^{k+1}(K)$ with $K = (P, B)$.
2. By the definition of the standard chase, $Ch^k(K) \subseteq Ch^{k+1}(K)$.
3. By (1) and (2), there is some BCQ q such that $Ch^k(K) \not\models q$ and $Ch^{k+1}(K) \models q$.
4. We can show via induction that $G^i(B) \subseteq Ch^i(K)$ for all $i \geq 0$.
5. By (4), $G^k(B) \subseteq Ch^k(K)$.
6. By (3) and (5), $G(B) \not\models q$.
7. By (3), $K \models q$.
8. By (6) and (7), the graph G is not a TG for P.

$\Longleftarrow$ Since P is *BDD*, we have that P is *k-BDD* for some $k \geq 0$. Therefore, the graph Φ^k_P is a TG for P by Theorem 26 and the program P is *FTG*. $\square$

Theorem 28. *P is* TDB *and* FOR *iff it is* BDD.

Proof. $\Longrightarrow$

1. Assume that P is (a) *FOR* and (b) *TDB*.
2. Let h be a homomorphism that maps every $t \in \mathsf{Nulls} \cup \mathsf{Vars}$ to a fresh $c_t \in \mathsf{Consts}$ unique for t.
3. For all facts φ that can be defined over some $s \in \mathsf{Preds}$ in P, we introduce the following notions.
   (a) Let ω_φ be some (arbitrarily chosen) rewriting for the BCQ φ with respect to P. Note that, such a rewriting must exist by (1.a).
   (b) Let k_φ be the smallest number such that $\mathit{Ch}^{k_\varphi}(P, h(\beta)) \models \mathit{Ch}^{k_\varphi+1}(P, h(\beta))$ for every disjunct β in the rewriting ω_φ. Note that, such a number must exist by (1.b).
4. For all $s \in \mathsf{Preds}$, let k_s be the smallest number such that $k_s > k_\varphi$ for all facts φ that can be defined over the predicate s. Note that, the number k_s is well-defined despite the fact that we can define infinitely many different facts over any given predicate. This is because $k_{s(t_1,\ldots,t_n)} = k_{s(u_1,\ldots,u_n)}$ if we have that there is a bijective function mapping t_i to u_i for all $1 \leq i \leq n$.
5. Let k_P be the smallest number such that $k_P > k_s + 1$ for all $s \in \mathsf{Preds}$ in P.
6. Consider some fact $\varphi = s(t_1, \ldots, t_n)$, some base instance B, and some $i \geq 0$. We show that, if the terms $t_1, \ldots, t_n$ are in $\mathit{Ch}^i(P, B)$ and $\varphi \in \mathit{Ch}(P, B)$, then $\varphi \in \mathit{Ch}^{i+k_P-1}(P, B)$.
   a. $h(\varphi) \in \mathit{Ch}((P, B'))$ with $B' = h(\mathit{Ch}^i(P, B))$.
   b. By (a): $B' \models \omega_{h(\varphi)}$ where $\omega_{h(\varphi)}$ is a UBCQ of the form $\exists \mathbf{x}_1.\beta_1 \vee \ldots \vee \exists \mathbf{x}_n.\beta_n$.
   c. By (b): $B' \models \exists \mathbf{x}_k.\beta_k$ for some $1 \leq k \leq n$.
   d. By (c): there is a homomorphism such that $h'(\beta_k) \subseteq B'$.
   e. By (d): $h'(\varphi) \in \mathit{Ch}((P, h(\beta_k)))$.
   f. By (5) and (e): $\mathit{Ch}^{k_P-1}(P, h'(\beta_k)) \supseteq \mathit{Ch}(P, h'(\beta_k))$.
   g. By (e) and (f): $h'(\varphi) \in \mathit{Ch}^{k_P-1}(P, h'(\beta_k))$.
   h. By (d) and (g): $h'(\varphi) \in \mathit{Ch}^{k_P-1}(P, B')$.
   i. By (h): $\varphi \in \mathit{Ch}^{i+k_P-1}(P, B)$.
7. For a fact $\varphi = s(t_1, \ldots, t_n)$, let $\mathsf{d}(\varphi) = max_{1\leq i\leq n}(\mathsf{d}(t_i))$.
8. By (6) and (7): We show via induction that, for all $d \geq 1$, the set $\mathit{Ch}^{d \cdot k_P}(P, F)$ contains all of the facts $\varphi \in \mathit{Ch}(P, F)$ with $\mathsf{d}(\varphi) \leq d$.
   - Base case: The set $\mathit{Ch}^0(P, F)$ contains all terms of depth 1 (i.e., all constants) that occur in $\mathit{Ch}(P, F)$. Hence, by (6), the set $\mathit{Ch}^{k_P}(P, F)$ contains every $\varphi \in \mathit{Ch}(P, F)$ with $\mathsf{d}(\varphi) = 1$.
   - Inductive step: Let $i \geq 1$. Then, $\mathit{Ch}^{(i-1)\cdot k_P}(P, F)$ contains all $\varphi \in \mathit{Ch}(P, F)$ with $\mathsf{d}(\varphi) = i-1$. Hence, the set $\mathit{Ch}^{(i-1)\cdot k_P+1}(P, F)$ contains all $t \in \mathsf{Terms}$ in $\mathit{Ch}(P, F)$ with $\mathsf{d}(\varphi) = i$. By (6), the set $\mathit{Ch}^{i\cdot k_P}(P, F)$ contains all $\varphi \in \mathit{Ch}(P, F)$ with $\mathsf{d}(\varphi) = i$.
9. By (1.b): There is some $k_d \geq 0$ that depends only on P such that, for every term in t occurring in $\mathit{Ch}((P, F))$, the depth t is equal or smaller than k_d.
10. By (8) and (9): P is $(k_d \cdot k_P)$-*BDD*. Note that, neither k_d nor k_P depend on the set of facts F.

$\Longleftarrow$ Since $P \in \mathit{BDD}$, $P \in k\text{-}\mathit{BDD}$ for some $k \geq 0$. Via induction on $i \in \{0, \ldots, k\}$, we can show that all terms in $\mathit{Ch}^i(P, B)$ are of depth i or smaller for any instance B, and hence, $P \in k\text{-}\mathit{TDB}$. Note that $\mathit{BDD} \subseteq \mathit{FOR}$ has been shown in [39]. □

THEOREM 29. *The language of all programs that admit a finite TG is undecidable.*

PROOF. This follows from the fact that *FTG* decidability implies decidability of *FOR* for Datalog programs, which is undecidable [28]. □

THEOREM 8. *For each KB (P, B) that admits a finite model, there exists an instance-dependent TG.*

PROOF. Recall that we denote by Φ_P^k the level-k full EG for a program P. We can stop the expansion of the level-k full EG for a program P when $\Phi_P^{k-1}(B) \models \Phi_P^k(B)$ holds. The above, along with Theorem 26 and the fact that whenever $\mathit{Ch}^{k-1}(P, B) \models \mathit{Ch}^k(P, B)$ holds for some $k \geq 1$, then the KB (P, B) admits a finite model (see [24]) conclude the proof of Theorem 8. □

D. PROOFS FOR RESULTS IN SECTION 5

THEOREM 10. *For any linear program P that is* FES, $\mathsf{tglinear}(P)$ *is a TG for P.*

PROOF. In order to show that the EG $G = (V, E)$ computed by Algorithm 1 is a TG for P, it suffices to show that $Ch^{\infty}(P, B)$ is logically equivalent to $G(B)$. The proof makes use of Propositions 30 and 31, as well as of Lemma 32. Note that Propositions 30 is a known result and, hence, we skip its proof, while Proposition 31 directly follows from Definition 5.

PROPOSITION 30. *For each linear program P and each base instance B, the following holds:*

$$Ch^{\infty}(P, B) \equiv \bigcup_{F \in B} Ch^{\infty}(P, \{F\}) \tag{2}$$

PROPOSITION 31. *For each linear TG G, each node $v \in G$ and each base instance B, the following holds:*

$$v(B) = \bigcup_{F \in B} u(\{F\}) \tag{3}$$

LEMMA 32. *For each fact $f \in \mathcal{H}(P)$, $\Gamma(\{f\}) \equiv Ch^{\infty}(P, \{f\})$, where Γ is the EG computed in lines 2–10 for fact f.*

PROOF. ($\Rightarrow$) We want to show that for each fact $f \in \mathcal{H}(P)$, there exists a homomorphism from $\Gamma(\{f\})$ into $Ch^{\infty}(P, \{f\})$. Due to Algorithm 1, we know that Γ is a graph of the form

$$\left(\bigcup_{j=1}^{n} u_j, \bigcup_{j=1}^{n-1} \{u_j \rightarrow_1 u_{j+1}\} \right) \tag{4}$$

where $u_j \in V$, for each $0 \leq i \leq n$; $u_j \rightarrow_1 u_{j+1} \in E$, for each $0 \leq i < n$; and $u_n = v$. To prove this direction, we will show that the following property holds, for each $0 \leq i \leq n$:

- ϕ_1. there exists a homomorphism h^i from $\Gamma_{\preceq u_i}(\{f\})$ into $Ch^{\infty}(P, \{f\})$.

For $i = 0$, since $\Gamma_{\preceq v_i}$ is the empty graph and hence $\Gamma_{\preceq u_i}(B) = B$ by Definition 5, it follows that ϕ_1 holds. For $i + 1$ and assuming that ϕ_1 holds for i the proof proceeds as follows. Since ϕ_1 holds for i, we know that there exists a homomorphism h^i from $\Gamma_{\preceq u_i}(\{f\})$ into $Ch^{\infty}(P, \{f\})$. If $u_{i+1}(\{f\}) = \emptyset$, then ϕ_1 trivially holds for $i + 1$. Hence, we will consider the case where $v_{i+1}(\{f\}) \neq \emptyset$. Since v_{i+1} is a child of v_i and since v_{i+1} is associated with some rule $r_{i+1} \in P$, it follows that there exists a homomorphism g from $\mathsf{body}(r_{i+1})$ into $v_i(\{f\})$ and $v_{i+1}(\{f\}) = g_s(\mathsf{head}(r_{i+1}))$. Due to g and due to h^i, we know that there exists a homomorphism $\psi = g \circ h^i$ from $\mathsf{body}(r_{i+1})$ into $Ch^{\infty}(P, \{f\})$ and, hence, a homomorphism ω from $g_s(\mathsf{head}(r_{i+1}))$ into $\psi_s(\mathsf{head}(r_{i+1}))$ mapping each value c occurring in $\mathsf{dom}(g)$ into $(g \circ h^i)(c)$ and n_z into n'_z, for each existentially quantified variable z of r_{i+1}, where $g(z) = n_z$ and $\psi(z) = n'_z$. We distinguish the cases:

- $\psi_s(\mathsf{head}(r_{i+1})) \in Ch^{\infty}(P, \{f\})$. Due to h^i and due to ω, it follows that $h^{i+1} = h^i \cup \omega$ is a homomorphism from $\Gamma_{\preceq v_{i+1}}(\{f\})$ into $Ch^{\infty}(P, \{f\})$.
- $\psi_s(\mathsf{head}(r_{i+1})) \notin Ch^{\infty}(P, \{f\})$. Then $Ch^{\infty}(P, \{f\}) \models \psi_s(\mathsf{head}(r_{i+1}))$ holds and hence, there exists a homomorphism θ from $\psi_s(\mathsf{head}(r_{i+1}))$ into $Ch^{\infty}(P, \{f\})$. Due to h^i, due to ω and due to θ, we can see that $h^{i+1} = (h^i \cup \omega) \circ \theta$ is a homomorphism from $\Gamma_{\preceq v_{i+1}}(\{f\})$ into $Ch^{\infty}(P, \{f\})$.

The above shows that ϕ_1 holds for $i + 1$ concluding the proof of this direction.

($\Leftarrow$) We want to show that for each fact $f \in \mathcal{H}(P)$, there exists a homomorphism from $Ch^{\infty}(P, \{f\})$ into $\Gamma(\{f\})$. We use I^i to denote $Ch^i(P, \{f\})$ and $\mathsf{chaseGraph}^i(P, \{f\})$ to denote the chase graph corresponding to $Ch^i(P, \{f\})$. The proof of this direction proceeds by showing that the following properties hold, for each $i \geq 0$:

- ϕ_2. there exists a homomorphism h^i from $Ch^i(P, \{f\})$ into $\Gamma(\{f\})$.
- ϕ_3. for each $f_1 \rightarrow_{r_1} f_2 \rightarrow_{r_2} \cdots \rightarrow_{r_j} f_{j+1}$ in $\mathsf{chaseGraph}^i(P, \{f\})$, a path of the form $u_1 \rightarrow_1 \cdots \rightarrow_1 u_j$ is in Γ, where for each $1 \leq k \leq j$: u_k is associated with r_k and there exists a homomorphism from f_{k+1} into $u_k(\{f\})$.

For $i = 0$, since $Ch^0(P, \{f\}) = \{f\}$ and since $f \in \Gamma(\{f\})$ by definition, it follows that the inductive hypotheses ϕ_2 and ϕ_3 holds. For $i + 1$ and assuming that ϕ_2 and ϕ_3 hold for i the proof proceeds as follows. Let Σ_r be the set of all active triggers for each $r \in P$ in I^i. Let also

$$\Delta I = \bigcup\nolimits_{r \in P} \bigcup\nolimits_{h \in \Sigma_r} h_s(\mathsf{head}(r)) \tag{5}$$

We distinguish the following cases:

- $I^i \models I^i \cup \Delta I$ holds. Then the equivalent chase terminates and hence $Ch^i(P, \{f\}) = Ch^{\infty}(P, \{f\})$. Since the inductive hypotheses ϕ_2 and ϕ_3 hold for i and due to the above, it follows that the inductive hypotheses will hold for $i + 1$.

- $I^i \models I^i \cup \Delta I$ does not hold. Then, for each rule $r \in P$ for which $\Sigma_r \neq \emptyset$ and for each $h \in \Sigma_r$, the chase graph $\mathsf{chaseGraph}^{i+1}(P, \{f\})$ will include an edge $h(\mathsf{body}(r)) \rightarrow_r h_s(\mathsf{head}(r))$. Due to the steps in lines 4–6, we know that Γ includes a node v associated with r $(*)$. We have the following two subcases:
  - There is no edge of the form $f' \rightarrow_{r'} h(\mathsf{body}(r))$ in $\mathsf{chaseGraph}^{i+1}(P, \{f\})$, for some $r' \in P$. Then, it follows that $h(\mathsf{body}(r)) = f$. Due to the above, due to $(*)$ and due to Definition 5, it follows that the inductive hypotheses ϕ_2 and ϕ_3 hold for $i + 1$.
  - Otherwise. Since the inductive hypothesis ϕ_3 holds for i and due to the steps in lines 7–9, it follows that an edge of the form $v' \rightarrow_1 v$ will be in Γ, where node v' is associated with rule r'. Furthermore, due to ϕ_3, there exists a homomorphism g^i from $h(\mathsf{body}(r))$ into $v'(\{f\})$. Due to g^i, due to the fact that the edge $v' \rightarrow_1 v$ is in Γ and due to Definition 5, there exists a homomorphism g^{i+1} from $h_s(\mathsf{head}(r))$ into $v(\{f\})$. Finally, due to the above, and since ϕ_2 holds for i, it follows that $h^{i+1} = h^i \circ g^{i+1}$ is a homomorphism from $Ch^i(P, \{f\})$ into $\Gamma(\{f\})$. Hence, ϕ_2 and ϕ_3 hold for $i + 1$ concluding the proof of this direction and, consequently of Lemma 32.

□

We are now ready to return to the proof of Theorem 10. Let Γ_f be the EG computed in lines 2–10 for each $f \in \mathcal{H}(P)$. Since Algorithm 1 only adds new nodes and new edges to set of nodes and the set of edges of an input EG, it follows that

$$V = \bigcup_{\forall f \in \mathcal{H}(P)} \nu(\Gamma_f) \tag{6}$$

$$E = \bigcup_{\forall f \in \mathcal{H}(P)} \epsilon(\Gamma_f) \tag{7}$$

Since for each base instance B of P and each fact $f' \in B$, there exists a fact $f \in \mathcal{H}(P)$ and a bijective function g over the constants in $\mathcal{C}$, such that $g(f') = f$ and from Lemma 32, it follows that $\Gamma_f(\{f'\}) \equiv Ch^\infty(P, f')$. Due to the above, due to (6) and (7) and since each node in V has up to one incoming edge, it follows that for each base instance B of P

$$\bigcup_{F \in B} Ch^\infty(P, \{F\}) \equiv \bigcup_{F \in B} G(\{F\}) \tag{8}$$

Due to Definition 5 and since each node in V has up to one incoming edge, it follows that for each base instance B of P we have

$$\bigcup_{F \in B} G(\{F\}) = G(B) \tag{9}$$

Finally, due to Proposition 30, and due to (8) and (9), we have $Ch^\infty(P, B) \equiv G(B)$, for each base instance B of P. The above concludes the proof of the first part of Theorem 10.

Theorem 11. *The execution time of Algorithm 1 for* FES *programs is double exponential in the input program P. If the arity of the predicates in P is bounded, the execution time is (single) exponential.*

Proof. We first show the first part of the theorem with a step-by-step argument.

1. Consider some program P.
2. The set $\mathcal{H}(P)$ is exponential in P.
   (a) Let **Ary** be the maximal arity of an extensional predicate occurring in P.
   (b) For an extensional predicate p occurring in P, there are at most $\mathbf{Ary}^{\mathbf{Ary}}$ facts in $\mathcal{H}(P)$ defined over p.
   (c) Since P is a linear rule set and extensional predicates may only appear in the body of a rule, we have that number of extensional predicates in P is at most $|P|$.
   (d) By (b) and (c): there are at most $|P| \times \mathbf{Ary}^{\mathbf{Ary}}$ facts in $\mathcal{H}(P)$.
3. For every fact $f \in \mathcal{H}(P)$, we may have to add at most $|Ch(P, \{f\})|^2$ nodes to the graph $\mathsf{tglinear}(P)$. (Note that $Ch(P, \{f\})$ is defined since P is *FES*.) Therefore, this graph contains at most $|Ch(P, \{f\})|^4$ edges.
4. From results in [38], the size of $Ch(P, \{f\})$ is double exponential in P.
   (a) By Proposition 30 in [38]: if $Ch(P, \{f\})$ is finite, then there exists a finite entailment tree $\mathcal{T}$ such that the set of atoms associated with $\mathcal{T}$ is a complete core.
   (b) In Algorithm 1 in [38], the authors describe how to construct the finite entailment tree for P and $\{f\}$.

(c) The complexity of this algorithm, as well as the size of the output tree, is double exponential in the input P and $\{f\}$. Note the discussion right after Algorithm 1 in [38].

5. By (2–4): the algorithm $\mathsf{tglinear}(P)$ runs in double exponential in P.

To show that the procedure $\mathsf{tglinear}(P)$ runs in single exponential time when $(*)$ the arity of the predicates in P is bounded, we can show that the size of $Ch(P, \{f\})$ is (single) exponential in P if $(*)$. In fact, this claim also follows from then results in [38]. Namely, if $(*)$, then entailment trees for P and f (as they are defined in [38]) are of polynomial depth because the number of "sharing types" for P is polynomial. Therefore, the size of these trees is exponential and so is the size of $Ch(P, \{f\})$. □

LEMMA 13. *Let P be a linear program, G be an EG for P and $u, v \in \nu(G)$. Then, there exists a preserving homomorphism from $u(B)$ into $v(B)$ for each base instance B, iff there exists a preserving homomorphism from $u(\{f\})$ into $v(\{f\})$, for each fact $f \in \mathcal{H}(P)$.*

PROOF. $(\Rightarrow)$ This direction trivially holds.
$(\Leftarrow)$ We want to show that if there exists a preserving homomorphism from $u(\{f\})$ into $v(\{f\})$, for each fact $f \in \mathcal{H}(P)$, then there exists a preserving homomorphism from $u(B)$ into $v(B)$, for each base instance B. The proof works by contradiction. Suppose that there exists a base instance B', such that there does not exist a preserving homomorphism from $u(B')$ into $v(B')$. Since there does not exist a preserving homomorphism from $u(B')$ into $v(B')$ and due to Proposition 31, it follows that there does not exist a preserving homomorphism from $\bigcup_{F' \in B'} u(\{F'\})$ into $\bigcup_{F' \in B'} v(\{F'\})$.
Next we show that there exists some $F' \in B'$, so that there does not exist a preserving homomorphism $h_{F'}$ from $u(\{F'\})$ into $v(\{F'\})$. The proof proceeds as follows. Suppose by contradiction that there exists a preserving homomorphism $h_{F'}$ from $u(\{F'\})$ into $v(\{F'\})$, for each $F' \in B'$, but there does not exist a preserving homomorphism $h_{B'}$ from $u(\{B'\})$ into $v(\{B'\})$. The above assumption, will be referred to as Assumption (A_1). By definition, we know that a preserving homomorphism from $u(\{B'\})$ into $v(\{B'\})$ maps each value c either (i) to itself if c was a schema constant or a null occurring in $(G(B') \setminus G_{\succeq u}(B')) \cap G_{\succeq u}(B')$ or (ii) to a fresh null occurring in a single fact from $v(\{F'\})$. Since for each $F' \in B'$, $h_{F'}$ maps each schema constant and each null occurring in $(G(B') \setminus G_{\succeq u}(B')) \cap G_{\succeq u}(B')$ to itself according to Assumption (A_1) and due to the above, it follows that there exist two facts $F'_1, F'_2 \in B'$ and a null n occurring in $G_{\succeq u}(B') \setminus G(B')$, such that $h_{F'_1}(\mathsf{n}) = \mathsf{m}_1$ and $h_{F'_2}(\mathsf{n}) = \mathsf{m}_2$, where $h_{F'_j}$ is a preserving homomorphism from $u(\{F'_j\})$ into $v(\{F'_j\})$, for each $1 \leq j \leq 2$. However, the above leads to a contradiction, since in linear TGs each null from $u(\{B'\})$ or $v(\{B'\})$ occurs in only one fact. The above shows that if there does not exist a preserving homomorphism from $\bigcup_{F' \in B'} u(\{F'\})$ into $\bigcup_{F' \in B'} v(\{F'\})$, then there exists some $F' \in B'$, such that there does not exist a preserving homomorphism from $u(\{F'\})$ into $v(\{F'\})$. The proof proceeds as follows.
Since for each $F' \in B'$, there exists a bijective function g over the constants in $\mathcal{C}$ and an instance $f \in \mathcal{H}(P)$, such that $g(F') = f$, it follows that there does not exist a preserving homomorphism from $u(\{f\})$ into $v(\{f\})$, for some $f \in \mathcal{H}(P)$. This leads to a contradiction. Hence, there exists a preserving homomorphism from $u(B)$ into $v(B)$, for each base instance B and thus, Lemma 13 holds. □

THEOREM 15. *For a TG G for a linear program P, $\mathsf{minLinear}(G)$ is a TG for P.*

PROOF. Recall from Definition 14 that $\mathsf{minLinear}(G)$ results from G after applying the following step until reaching a fixpoint: (i) find a pair of nodes u and v with u being dominated by v; (ii) remove v from $\nu(G)$; and (iii) add an edge $v \rightarrow_j u'$, for each edge $u \rightarrow_j u'$ from $\epsilon(G)$. Let G^i be the EG computed at the beginning of the i-th step of this iterative process with $G^0 = G$. In order to show that $\mathsf{minLinear}(G)$ is a TG for P, we need to show that the following property holds for each BCQ Q entailed by (P, B):

$$G^i(B) \models Q \tag{$*$}$$

We can see that $(*)$ holds $i = 0$, since $G^0 = G$. For $i + 1$ and assuming that $(*)$ holds for $i \geq 0$ we proceed as follows. Suppose that there exists a homomorphism q from Q into $G^i(B)$. Furthermore, let C_1 and C_2 be two conjuncts, such that $Q = C_1 \wedge C_2$ and q maps C_1 into $G_{\succeq u}(B)$ and C_2 into $G(B) \setminus G_{\succeq u}(B)$. Due to the above, it follows that q maps each variable occurring both in C_1 and C_2 either to a constant or to a null occurring in $(G(B) \setminus G_{\succeq u}(B)) \cap G_{\succeq u}(B)$.
From Definition 14 we know that (i) there exists a pair of nodes $u, v \in \nu(G^i)$ with u being dominated by v and that (ii) the graph Γ^{i+1} that results from $G^i_{\succeq v}$ after adding an edge $v \rightarrow u'$, for each edge $u \rightarrow u'$ in $\epsilon(G^i)$, is a subgraph of G^{i+1}. Since q maps each variable occurring both in C_1 and C_2 either to a constant or to a null occurring in $(G(B) \setminus G_{\succeq u}(B)) \cap G_{\succeq u}(B)$ it follows that $*$ holds for $i + 1$ if the following holds:

LEMMA 33. *There exists a homomorphism g from $G^i_{\succeq u}(B)$ into $\Gamma^{i+1}(B)$ so that $g(c) = h(c)$, for each $c \in \mathsf{dom}(h)$.*

The proof of Lemma 33 directly follows from the facts that (i) there exists a preserving homomorphism from $u(B)$ into $v(B)$, for each base instance B and (ii) all subgraphs rooted at each child of u are copied below v.

From Lemma 33 and since Γ^{i+1} is a subgraph of G^{i+1}, we have that there exists a homomorphism $(q \circ g)$ from Q into $G^{i+1}(B)$ concluding the proof of $(*)$ for $i+1$ and hence the proof of Theorem 10. $\square$

E. PROOFS FOR RESULTS IN SECTION 5.1

Below, we recapitulate the notion of the answers of non-Boolean CQs on a KB. The answers to a CQ Q on a KB (P, B), denoted as $\mathsf{ans}(Q, P, B)$, is the set of tuples that are answers to each model of (P, B), i.e., $\{\mathbf{t} | \mathbf{t} \in Q(I),$ for each model I of $(P, B)\}$.

Lemma 18. *Let G be an EG for a Datalog program P and B be a base instance of P. Then for each $v \in \nu(G)$ we have: $v(B)$ includes exactly a fact $A(\mathbf{t})$ with A being the head predicate of* $\mathsf{rule}(v)$*, for each answer* $\mathbf{t}$ *to the EG-rewriting of v on B.*

Proof. Let R be the predicate in the body of the characteristic query of v. In order to prove Lemma 18, we have to show that $\mathbf{t}$ is an answer to $\mathsf{rew}(v)$ on B iff $R(\mathbf{t}) \in v(B)$. The proof is based on (i) the correspondence between the rewriting process of Definition 17 and the query rewriting algorithm from [29], called XRewrite and (ii) the correctness of XRewrite. In particular, the steps of the proof are as follows. First, we compute a new set of rules P^* by rewriting the rules associated with the nodes in $G_{\preceq v}$. This rewriting process is described in Definition 34. Then, we establish the relationship between the $\mathsf{rew}v$ and the rewritings computed by XRewrite. In particular, Lemma 36 shows that Definition 17 and XRewrite result in the same rewritings modulo bijective variable renaming, when the latter is provided with P^* and a rewriting of the characteristic query of v denoted as Q^*. A direct consequence of Lemma 36 is that XRewrite terminates when provided with P^* and Q^*. In order to show our goal, we make use of the above results as well as of Lemma 37 .

Below, we describe XRewrite. Given a CQ Q and a set of rules P, XRewrite computes a rewriting Q_r of Q so that for any null-free instance I, the answers to Q_r on (P, I) coincide with the answers to Q on (P, I). We describe how XRewrite(P, Q) works when P is Datalog. At each step i, XRewrite computes a tree $\mathcal{T}^i$, where each node κ is associated with a CQ denoted as $\mathsf{query}(\kappa)$. When $i = 0$, $\mathcal{T}^0$ includes a single root node associated with Q. When $i > 0$, then $\mathcal{T}^i$ is computed as follows: for each leaf node κ in $\nu(\mathcal{T}^{i-1})$, each atom β occurring in the body of $\mathsf{query}(\kappa)$ and each rule $r \in P$ whose head unifies with β, XRewrite:

1. computes a new query Q' by (i) computing the MGU θ of $\{\mathsf{head}(r), \alpha\}$, (ii) replacing α in the body of $\mathsf{query}(\kappa)$ with $\mathsf{body}(r)$ and (iii) applying θ on the resulting query;
2. adds a new node κ' in $\mathcal{T}^i$ and associates it with Q'; and
3. adds the edge $\kappa \xrightarrow{(\alpha,r)} \kappa'$ to $\mathcal{T}^i$.

Notice that XRewrite also includes a factorization step, however, this step is only applicable in the presence of existential rules.

We now introduce some notation related to Definition 17. We denote by $\mathsf{rew}^i(v)$ the CQ at the beginning of the i-th iteration of the rewriting step of Definition 17 with $\mathsf{rew}^0(v)$ being equal to the characteristic query of v. We use $\mathsf{rew}^i(v) \xrightarrow{\alpha_i} \mathsf{rew}^{i+1}(v)$ to denote that $\mathsf{rew}^{i+1}(v)$ results from $\mathsf{rew}^i(v)$ after choosing the atom α_i from the body of $\mathsf{rew}^i(v)$ at the beginning of the i-th rewriting step.

Below, we describe a process that computes a new ruleset by rewriting the rules associated with the nodes of an EG.

Definition 34. *Let G be an EG of a program P with a single leaf node. Let π be a mapping associating each edge $\varepsilon \in \epsilon(G) \cup \{\diamond\}$ with $\diamond$ denoting the empty edge, with a fresh predicate $\pi(\epsilon)$. We denote by $\rho(G, \pi)$ the rules obtained from the rules associated with the nodes in G after rewriting them as follows: replace each $A(\mathbf{X})$ that is either*

- *i. the i-th intensional atom in the body of* $\mathsf{rule}(v)$ *or the head atom of* $\mathsf{rule}(u)$ *and $\varepsilon := u \rightarrow_i v$ in $\epsilon(G)$; or*
- *ii. the head atom of* $\mathsf{rule}(\kappa)$ *with κ being the leaf node of G,*

by $A^*(\mathbf{X})$, where $A^* = \pi(\varepsilon)$ when (i) holds, or $A^* = \pi(\diamond)$ when (ii) holds.

For a node $u \in \nu(G)$, we denote by r^π_u the rule from $\rho(G, \pi)$ that results after rewriting $\mathsf{rule}(u)$*.*

From Definition 34 we can see that the following holds:

Corollary 35. *For an EG G of a program P with a single leaf node, a mapping π associating each edge $\varepsilon \in \epsilon(G) \cup \{\diamond\}$ with a fresh predicate and two rules ϱ_1 and ϱ_2 from $\rho(G, \pi)$, we have: the i-th body atom of ϱ_1 has the same predicate with the head atom of ϱ_2 only if ϱ_j is of the form $r^\pi_{u_j}$, for $1 \leq j \leq 2$ and $u_2 \rightarrow_i u_1$ is in $\epsilon(G)$.*

Let $\Gamma = G_{\preceq v}$ and n be the depth of Γ. Let π be a mapping from edges to predicate as defined above. Let P^* be the rules in $\rho(\Gamma, \pi)$. Let R be the predicate occurring in the head of $\mathsf{rule}(v)$ and let $\mathsf{head}(\mathsf{rule}(v)) = R(\mathbf{Y})$. Let

$R^* = \pi(\diamond)$. Let $Q(\mathbf{Y}) \leftarrow R(\mathbf{Y})$ and $Q^*(\mathbf{Y}) \leftarrow R^*(\mathbf{Y})$. Let $\mathcal{T}^i$ be the tree computed at the end of the i-th iteration of $\mathsf{XRewrite}(P^*, Q^*)$.

Below, we establish the relationship between the $\mathsf{rew}v$ and the rewritings computed by $\mathsf{XRewrite}$.

Lemma 36. *For each branch $\kappa^0 \xrightarrow{\epsilon_0} \dots \xrightarrow{\epsilon_n} \kappa^{n+1}$ with $\epsilon_i = (\beta_i, \varrho_i)$ and κ^i is a node of depth i in $\mathcal{T}^{n+1}$, there exists a sequence*

$$\mathsf{rew}^0(v) \xrightarrow{\alpha_0} \dots \xrightarrow{\alpha_n} \mathsf{rew}^{n+1}(v) \tag{10}$$

such that $\mathsf{rew}^{n+1}(v)$ equals $\mathsf{query}(\kappa^{n+1})$ modulo bijective variable renaming.

The proof of Lemma 36 follows from: (i) $Q(\mathbf{Y}) \leftarrow R(\mathbf{Y})$ and $Q^*(\mathbf{Y}) \leftarrow R^*(\mathbf{Y})$, where $R^* = \pi(\diamond)$, (ii) the correspondence between the rewriting steps (1)–(3) of $\mathsf{XRewrite}$ and the rewriting process of Definition 17 and (iii) Corollary 35.

From Lemma 36 and since $\mathsf{rew}^{n+1}(v)$ includes only EDP-atoms, we have that: $\mathsf{XRewrite}(P^*, Q^*)$ terminates after $n+1$ iterations. Furthermore, since $\mathsf{XRewrite}$ terminates and due to its correctness we have: $R^*(\mathbf{t}) \in \mathit{Ch}^n(P^*, B)$ iff $\mathbf{t}$ is an answer to $\mathsf{XRewrite}(P^*, Q^*)$ on B.

Due to Corollary 35, we also have:

Lemma 37. *For each base instance B, the following inductive property holds for each $1 \leq i \leq n+1$:*
- *ϕ. $S(\mathbf{t}) \in u(B)$, with u being a node of depth i in Γ iff $S^*(\mathbf{t}) \in \mathit{Ch}^i(P^*, B)$, where S^* is the predicate of the head atom of r^π_u.*

We now establish the correspondence between $v(B)$ and the answers to $\mathsf{rew}(v)$ on B. From Lemma 37, we have: $R^*(\mathbf{t}) \in \mathit{Ch}^n(P^*, B)$ iff $R(\mathbf{t}) \in v(B)$. Since $R^*(\mathbf{t}) \in \mathit{Ch}^n(P^*, B)$ iff $\mathbf{t}$ is an answer to $\mathsf{XRewrite}(P^*, Q^*)$ on B, it follows that $R(\mathbf{t}) \in v(B)$ iff $\mathbf{t}$ is an answer to $\mathsf{XRewrite}(P^*, Q^*)$ on B. Since $\mathbf{t}$ is an answer to $\mathsf{XRewrite}(P^*, Q^*)$ on B iff $\mathbf{t}$ is an answer to $\mathsf{rew}(v)$ on B and due to the above, it follows that: $\mathbf{t}$ is an answer to $\mathsf{rew}(v)$ on B iff $R(\mathbf{t}) \in v(B)$. The above completes the proof of Lemma 18. $\square$

Lemma 38. *For each EG G for a Datalog program P and each base instance B of P, $G(B) = \mathsf{minDatalog}(G)(B)$.*

Proof. Let G^i be the EG at the beginning of the i-th iteration of $\mathsf{minDatalog}(G)$ with $G^0 = G$. We show that the following property holds for each $i \leq 0$:
- ϕ. $G(B) = \mathsf{minDatalog}(G^i)(B)$.

For $i = 0$, ϕ trivially holds, since $G^0 = G$. For $i+1$ and assuming that ϕ holds for $i \leq 0$, we have. Let u, v be a pair of nodes in $\nu(G^i)$, such that (i) v's depth is not less than u's depth, (ii) the predicates of $\mathsf{head}(\mathsf{rule}(v))$ and of $\mathsf{head}(\mathsf{rule}(u))$ are the same and (iii) the EG-rewriting of v is contained in the EG-rewriting of u. In order to show that the inductive property for ϕ, it suffices to show that for each node $w \in \nu(G^i)$ for which $v \rightarrow_j w \in \epsilon(G^i)$ holds for some j, and each base instance B of P, $w(B)$ is the same both in $G^i(B)$ and in $G^{i+1}(B)$. However, this holds since (i) for each $v \rightarrow_j w \in \epsilon(G^i)$, we have $u \rightarrow_j w \in \epsilon(G^{i+1})$, (ii) $R(\mathbf{t}) \in v(B)$ implies $R(\mathbf{t}) \in u(B)$ and $G^i_{\preceq u} = G^{i+1}_{\preceq u}$. The above shows that ϕ holds for $i+1$ and concludes the proof of Lemma 38. $\square$

Theorem 20. *If G is a TG for a Datalog program P, then $\mathsf{minDatalog}(G)$ is a minimum size TG for P.*

Proof. *Part I.* The proof follows from Lemma 38.

Part II. First, we can see that the following holds due to Definition 19:

Corollary 39. *For a TG G of a Datalog program P, there exists no two nodes u, v in $\mathsf{minDatalog}(G)$ satisfying the following: (i) u and v define the same predicate*[3] *A and (ii) $\mathsf{rew}(u) \subseteq \mathsf{rew}(v)$.*

The proof works by contradiction. Let P be a Datalog program, G be a TG for P and $\Gamma = \mathsf{minDatalog}(G)$. Suppose by contradiction that there exists a TG Γ' for P with $\nu(\Gamma) > \nu(\Gamma')$. From Lemma 18 we know that for each set of nodes $u_1, \dots, u_m$ from $\nu(\Gamma)$ defining a predicate A, there exists a set of nodes $u'_1, \dots, u'_n$ from $\nu(\Gamma')$ defining also A, such that the following holds:

$$\mathsf{rew}(u_1) \cup \dots \cup \mathsf{rew}(u_m) \equiv \mathsf{rew}(u'_1) \cup \dots \cup \mathsf{rew}(u'_n) \tag{11}$$

Since $\nu(\Gamma) > \nu(\Gamma')$, we know that there exist a set $u_1, \dots, u_m$ and a set $u'_1, \dots, u'_n$ so that $m > n$. Since (11) holds, we know from [52] that the following hold:
- for each $\mathsf{rew}(u_i)$ with $1 \leq i \leq m$, there exists a $\mathsf{rew}(u'_j)$ with $1 \leq j \leq n$, such that $\mathsf{rew}(u_i) \subseteq \mathsf{rew}(u'_j)$;
- for each $\mathsf{rew}(u'_j)$ with $1 \leq j \leq n$, there exists a $\mathsf{rew}(u_\ell)$ with $1 \leq \ell \leq n$, such that $\mathsf{rew}(u'_j) \subseteq \mathsf{rew}(u_\ell)$.

Since $m > n$, it follows that there exist i_1, i_2 with $1 \leq i_1, i_2 \leq m$ and an ℓ with $1 \leq \ell \leq n$, such that $\mathsf{rew}(u_{i_1}) \subseteq \mathsf{rew}(u'_\ell)$ and $\mathsf{rew}(u_{i_2}) \subseteq \mathsf{rew}(u'_\ell)$ hold. Below, we show how we reach a contradiction. We consider the following cases:

[3]We say that a node u in a TG defines a predicate A if the predicate of $\mathsf{head}(u)$ is A.

- there exists an i_3 with $1 \leq i_3 \leq m$ and $i_3 \neq i_1, i_2$, such that $\mathsf{rew}(u'_\ell) \subseteq \mathsf{rew}(u_{i_3})$. From the above, it follows that $\mathsf{rew}(u_{i_1}) \subseteq \mathsf{rew}(u_{i_3})$ and $\mathsf{rew}(u_{i_2}) \subseteq \mathsf{rew}(u_{i_3})$ leading to a contradiction due to Corollary 39.
- $\mathsf{rew}(u_{i_1}) \subseteq \mathsf{rew}(u'_\ell)$. From the above, it follows that $\mathsf{rew}(u_{i_2}) \subseteq \mathsf{rew}(u_{i_1})$ leading again to a contradiction due to Corollary 39.

The above completes the proof of Theorem 20. □

Theorem 21. *For a Datalog program P and a TG G for P, deciding whether G is a TG of minimum size for P is co-NP-complete.*

Proof. *Membership.* We show that deciding wether G is a TG of P *not* of minimum size is in NP. By Definition 19 and Theorem 20, G is a TG of P not of minimum size iff there exists a pair of vertices u and v in G satisfying the conditions in Definition 19 for which $\mathsf{rew}(v) \subseteq \mathsf{rew}(u)$ (remember that the last condition holds iff there exists a homomorphism from $\mathsf{rew}(u)$ to $\mathsf{rew}(v)$). Hence, to *disprove* that G is a TG of P of minimum size, it is sufficient to guess such nodes u and v, guess the homomorphism from $\mathsf{rew}(u)$ to $\mathsf{rew}(v)$ (observe that the size of $\mathsf{rew}(u)$ and $\mathsf{rew}(v)$ is polynomial), then compute $\mathsf{rew}(u)$ and $\mathsf{rew}(v)$ (feasible in deterministic polynomial time), and then check that the guessed homomorphism is correct (feasible in deterministic polynomial time). This procedure is feasible in NP.

Hardness. We show the co-NP-hardness of the problem by showing the NP-hardness of its complement. The reduction is from the NP-complete problem of query containment in relational DBs: given two CQs $Q_1(\mathbf{X})$ and $Q_2(\mathbf{X})$ for a relational DB, decide whether $Q_1(\mathbf{X}) \subseteq Q_2(\mathbf{X})$. Let the queries be $Q_1(\mathbf{X}) \leftarrow a_{i_1}(\mathbf{X}_{i_1}), \dots, a_{i_n}(\mathbf{X}_{i_n})$ and $Q_2(\mathbf{X}) \leftarrow a_{j_1}(\mathbf{X}_{j_1}), \dots, a_{j_m}(\mathbf{X}_{j_m})$.

We now describe the reduction. Consider the following program P and TG G.

The rules of P are obtained as follows. Let $D = \{a_{k_1}, \dots, a_{k_\ell}\}$ be the set of all the *distinct* predicates from $\{a_{i_1}, \dots, a_{i_n}\}$. For each of the predicates a_{k_t} in D, there is a rule $a_{k_t}(\mathbf{X}_{k_t}) \to A_{k_t}(\mathbf{X}_{k_t})$ in P. In P there are also the rules $A_{i_1}(\mathbf{X}_{i_1}), \dots, A_{i_n}(\mathbf{X}_{i_n}) \to Q(\mathbf{X})$ and $a_{j_1}(\mathbf{X}_{j_1}), \dots, a_{j_m}(\mathbf{X}_{j_m}) \to Q(\mathbf{X})$.

The TG G is as follows. There is a node v_{k_t} associated with each of the rules $a_{k_t}(\mathbf{X}_{k_t}) \to A_{k_t}(\mathbf{X}_{k_t})$; there is a node v associated with the rule $A_{i_1}(\mathbf{X}_{i_1}), \dots, A_{i_n}(\mathbf{X}_{i_n}) \to Q(\mathbf{X})$; and there is a node u associated with the rule $a_{j_1}(\mathbf{X}_{j_1}), \dots, a_{j_m}(\mathbf{X}_{j_m}) \to Q(\mathbf{X})$. The edges of G are: for each $1 \leq s \leq n$, there is an edge labelled s to node v from the node v_{k_t} such that the predicate of $\mathsf{head}(\mathsf{rule}(v_{k_t}))$ is A_{i_s}.

We show the G is a TG of minimum size for P iff $Q_1(\mathbf{X}) \subseteq Q_2(\mathbf{X})$.

First, observe that the predicates of the rules associated with nodes v_{k_t} are all distinct, and they differ from the predicate of the heads of the rules associated with u and v. Hence none of the nodes v_{k_t} can be removed from G in the minimization process. Nodes u and v are the only nodes in G associated with rules with the same head predicate. The depth of u is 0, while the depth of v is 1. Hence, v is the only node that can be removed in the minimization process. Therefore, G is not of minimum size iff v can be removed. The node v can be removed iff $\mathsf{rew}(v) \subseteq \mathsf{rew}(u)$, and hence, by the definition of P, iff $Q_1(\mathbf{X}) \subseteq Q_2(\mathbf{X})$. □

Theorem 24. *For a Datalog program P and a base instance B, $\mathsf{TGmat}(P, B) = Ch(P, B)$.*

Proof. We first show that

Claim 40. *For each node $v \in \nu(G)$ and each instance I, we have*

$$v(B, I) = v(B) \setminus I \tag{12}$$

Proof. Let $A(\mathbf{X})$ be the head atom of $\mathsf{rule}(v)$ and let $Q(\mathbf{Y}) \leftarrow \bigwedge_{i=1}^{n} f_i$ be the EG-rewriting of v.

Recall from Lemma 18 that for each base instance of B of P we have: $v(B)$ includes exactly a fact $A(\mathbf{t})$ for each answer $\mathbf{t}$ to the EG-rewriting of v on B.

Now consider any $m \geq 1$ atoms $f_{i_1}, \dots, f_{i_m}$ from the body of Q whose variables include all variables in $\mathbf{Y}$. Consider also the query $Q'(\mathbf{Y}) \leftarrow f_{i_1} \wedge \dots \wedge f_{i_m}$. From [17], it follows that Q is contained in Q', i.e., for each base instance B, each answer $\mathbf{t}$ to Q on B is an answer to Q' on B. From the above, we have: each $\mathbf{t}$, for which $A(\mathbf{t}) \in v(B)$ holds, is also an answer to $Q'(\mathbf{Y}) \leftarrow f_{i_1} \wedge \dots \wedge f_{i_m}$ on B. We refer to this conclusion as $(*)$.

Since step (2) of Definition 23 considers each homomorphism h for which (i) $h(\mathbf{X})$ is an answer to Q' on B and (ii) $A(h(\mathbf{X})) \notin I$ and due to $(*)$, it follows that Claim 40 holds. □

Let I^k be the instance computed the beginning of the k-th iteration of the steps in lines 2–8 of Algorithm 2. Then, using Claim 40, Theorem 26 and Lemma 38, we can easily show that for each $k \geq 0$, the following property holds:

- ϕ. $I^k = Ch^k(P, B)$

The above concludes the proof of Theorem 24.

F. ADDITIONAL EXAMPLES

Example 41. *We show how reasoning over the TG G_1 from Figure 1 proceeds for the base instance $B = \{r(c_1, c_2)\}$.*

Reasoning starts from the root nodes u_1 and u_2, which are associated with the rules r_1 and r_4, respectively. Since there exists a homomorphism $h = \{X \mapsto c_1, Y \mapsto c_2\}$ from $\mathsf{body}(r_1)$ into B and from $\mathsf{body}(r_4)$ into B, we have

$$u_1(\{f_1\}) = \{R(c_1, c_2)\} \tag{13}$$

$$u_2(\{f_1\}) = \{T(c_2, c_1, \mathsf{n}_1)\} \tag{14}$$

where n_1 is a null. Then, since there exists an edge from u_1 to u_3 and since u_3 is associated with r_2, we compute all homomorphisms from $\mathsf{body}(r_2)$ into $u_1(B)$. Since there exists a homomorphism $h = \{X \mapsto c_1, Y \mapsto c_2\}$ from $\mathsf{body}(r_2)$ into $u_1(B)$, we have

$$u_3(\{f_1\}) = \{T(c_2, c_1, c_2)\} \tag{15}$$

Since there is no other node, reasoning stops.

Example 42. *We demonstrate the notion of preserving homomorphisms introduced in Definition 12.*

Consider the facts $f_1 = r(c_1, c_2)$ and $f_2 = r(c_3, c_3)$ from the set $\mathcal{H}(P_1)$. By applying Definition 5 for the base instance $\{f_1\}$, we have $u_1(\{f_1\})$, $u_2(\{f_1\})$ and $u_3(\{f_1\})$ as in (13), (14) *and* (15). *Similarly, by applying Definition 5 for the base instance $\{f_2\}$, we have*

$$u_1(\{f_2\}) = \{R(c_3, c_3)\} \tag{16}$$

$$u_2(\{f_2\}) = \{T(c_3, c_3, \mathsf{n}_2)\} \tag{17}$$

$$u_3(\{f_2\}) = \{T(c_3, c_3, c_3)\} \tag{18}$$

Above, n_2 is a null. We can see that there exists a preserving homomorphism from $u_2(\{f_1\})$ into $u_3(\{f_1\})$ mapping n_1 to c_2, since the null n_1 is not shared among the facts occurring in the instances associated with u_2 and u_2. For the same reason, there exists a preserving homomorphism from $u_2(\{f_2\})$ into $u_3(\{f_2\})$ mapping n_2 to c_3. Hence according to Lemma 13, there exists a preserving homomorphism from $u_2(B)$ into $u_3(B)$ for each base instance B.

Example 43. *We demonstrate the computation of EG-rewritings introduced in Definition 17.*

Consider the rules

$$r(X_1, Y_1, Z_1) \to T(X_1, X_1, Y_1) \tag{r_{10}}$$

$$T(X_2, Y_2, Z_2) \to R(Y_2, Z_2) \tag{r_{11}}$$

where r is the only extensional predicate. Consider now an EG having nodes u_1 and u_2, where u_i is associated with r_i for each $1 \leq i \leq 2$, and the edge $u_1 \to_1 u_2$.

To compute the EG-rewriting $\mathsf{rew}(u_2)$ of u_2 we first form the query

$$Q(Y_2, Z_2) \leftarrow R(Y_2, Z_2) \tag{19}$$

and associate the atom $R(Y_2, Z_2)$ with u_2. The following steps take place in the first iteration of the rewriting algorithm. First, since $R(Y_2, Z_2)$ is the only intensional atom in the query we have $\alpha = R(Y_2, Z_2)$. Then, according to step (ii) and since the node u_2 is associated with $R(Y_2, Z_2)$, we compute the MGU θ_1 of the set $\{\mathsf{head}(u_2), R(Y_2, Z_2)\}$. We have $\theta_1 = \{Y_2 \to Y_2, Z_2 \to Z_2\}$, since $\mathsf{head}(u_2) = R(Y_2, Z_2)$. By applying the step (iii), the query in (19) *becomes*

$$Q(Y_2, Z_2) \leftarrow T(X_2, Y_2, Z_2) \tag{20}$$

In step (iv) we associate the fact $T(X_2, Y_2, Z_2)$ with node u_1 due to the edge $u_1 \to_1 u_2$.

In the second iteration of the rewriting algorithm, we have $\alpha = T(X_2, Y_2, Z_2)$. Since the fact $T(X_2, Y_2, Z_2)$ is associated with node u_1, in step (ii) we compute the MGU θ_2 of the set $\{\mathsf{head}(u_1), T(X_2, Y_2, Z_2)\}$. We have $\theta_2 = \{X_1 \to Y_2, X_2 \to Y_2, Y_1 \to Z_2\}$. In step (iii) we replace $\alpha = T(X_2, Y_2, Z_2)$ in (20) *with $\mathsf{body}(u_1) = r(X_1, Y_1, Z_1)$ and apply θ_2 to the resulting query. The query in* (20) *becomes*

$$Q(Y_2, Z_2) \leftarrow r(Y_2, Z_2, Z_1) \tag{21}$$

Since there is no incoming edge to u_1, we associate no node to the fact $r(Y_2, Z_2, Z_1)$. The algorithm then stops, since there is no extensional fact in (21). *The EG-rewriting of u_2 is the query shown in* (21).

Example 44. *We demonstrate the notion of compatible nodes introduced in Definition 9, as well as the procedure for computing instance-dependent TGs from Section 4.*

$$\begin{tikzcd}
u_2\backslash r_{12} \arrow[r, "2"] & u_3\backslash r_{14} \arrow[r, "2"] & u_5\backslash r_{14} \\
u_1\backslash r_{13} \arrow[ur, "1"] \arrow[urr, "1"] \arrow[dr, bend left, "1"] \arrow[dr, bend right, "2"'] \arrow[drr, "1"] & & \\
& u_4\backslash r_{15} \arrow[r, "2"] & u_6\backslash r_{14}
\end{tikzcd}$$

Figure 3: Part of the graph from Example 44.

Consider the program P_3

$$a(X) \rightarrow A(X) \tag{r_{12}}$$
$$r(X,Y) \rightarrow R(X,Y) \tag{r_{13}}$$
$$R(X,Y) \wedge A(Y) \rightarrow A(X) \tag{r_{14}}$$
$$R(X,Y) \wedge R(Y,Z) \rightarrow A(X) \tag{r_{15}}$$

where a and r are extensional predicates. Figure 3 shows part of the graph computed up to level 3. Next to each node, we show the rule associated with it. For example, node u_1 is associated with rule r_{13} and node u_2 is associated with rule r_{12}.

When $k = 1$, G^1 includes two nodes, one associated with rule r_{12} (u_2) and one associated with rule r_{13} (u_1). When $k = 2$, r_{14} has only one 2-compatible combination of nodes. That is (u_1, u_2). Hence, the technique will add one fresh node u_3, associated with r_{14} and will add the edges $u_1 \rightarrow_1 u_3$ and $u_2 \rightarrow_2 u_3$. The 2-compatible combination of nodes for r_{15} is (u_2, u_2). Hence, the technique will add one fresh node u_4, associated with r_15 and will add the edges $u_2 \rightarrow_1 u_4$ and $u_2 \rightarrow_2 u_4$.

When $k = 3$, r_{14} has the following 3-compatible combinations of nodes: (u_1, u_3) and (u_1, u_4). For each such 3-compatible combinations of nodes, the algorithm adds a fresh node and associates it with r_{14}. For $k = 3$, the 3-compatible combinations of nodes for r_{15} are: (u_2, u_3), (u_2, u_4), (u_3, u_2), (u_4, u_2), (u_3, u_4), (u_4, u_3), (u_3, u_3), (u_4, u_4). Again, for each such combination of nodes, the algorithm adds a fresh node and associates it with r_{15}.