Materializing Knowledge Bases via Trigger Graphs

Efthymia Tsamoura
Samsung AI Research
Cambridge, United Kingdom
efi.tsamoura@samsung.com

David Carral
LIRMM, Inria, University of Montpellier, CNRS
Montpellier, France
david.carral@inria.fr

Enrico Malizia
University of Bologna
Bologna, Italy
enrico.malizia@unibo.it

Jacopo Urbani
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
jacopo@cs.vu.nl

ABSTRACT

The *chase* is a well-established family of algorithms used to materialize Knowledge Bases (KBs) for 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, called GLog, and our experiments show that it can be significantly more efficient than the chase enabling us to materialize Knowledge Graphs with 17B facts in less than 40 min using a single machine with commodity hardware.

PVLDB Reference Format:
Efthymia Tsamoura, David Carral, Enrico Malizia, and Jacopo Urbani. Materializing Knowledge Bases via Trigger Graphs. PVLDB, 14(6): XXX-XXX, 2021.
doi:10.14778/3447689.3447699

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/karmaresearch/glog.

1 INTRODUCTION

Motivation. Knowledge Bases (KBs) are becoming increasingly important with many industrial key players investing on this technology. For example, Knowledge Graphs (KGs) [29] have emerged as the main vehicle for representing factual knowledge on the Web, and enjoy a widespread adoption [44]. Moreover, several key industrial players, like Google and Microsoft, are building KGs to support their core business. For instance, the KG developed at Microsoft is used to support question answering, while Google uses KGs to enable various products to respond more appropriately to user requests. The use of 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 [21, 33].

A KB can be viewed as a classical database B with factual knowledge and a set of logical rules P, called the *program* of the KB, allowing the derivation of additional knowledge. One class of rules that is of particular interest both to academia and to industry is Datalog [1]. Datalog is a recursive language with declarative semantics that allows users to succinctly write recursive graph queries. Beyond expressing graph queries, e.g., graph reachability, Datalog allows richer fixed-point graph analytics via aggregate functions. LogicBlox and LinkedIn use Datalog to develop high-performance applications, or to compute analytics over its KG [2, 42]. Google developed their own Datalog engine called Yedalog [19]. Other industrial users include Facebook, BP [9], and Samsung [36].

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 [7, 8, 12, 51]. The last application is particularly useful in the presence of computational or memory restrictions. Beyond knowledge exploration, other applications of materialization are data wrangling [32], entity resolution [34], data exchange [24] and query answering over OWL [40] and RDFS [15] ontologies. Finally, materialization has been also used in probabilistic KBs [53].

Problem. The increasing sizes of modern KBs [44], and the fact that materialization is not a one-off operation when used for goal-driven query answering, urge the need for improving the performance of materialization. The *chase*, which was introduced in 1979 by Maier et al. [38], has been the most popular materialization technique and has been adopted by several commercial and open source engines such as VLog [55], RDFox [43], and Vadalog [9].

To improve the performance of materialization, different approaches have focused on different inefficiency aspects. One approach is to reduce the number of facts added to the KB. This is the take of some of the chase variants proposed by the database and AI communities [10, 22, 45]. A second approach is to parallelize the computation. For example, RDFox proposes a parallelization technique for Datalog rules [43], while WebPIE [56] and Inferray

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447699

[49] propose parallelization techniques for fixed RDFS rules. Orthogonal to those approaches are those employing compression and columnar storage layouts to reduce memory consumption [31, 55].

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) [1]. 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 each 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 the facts currently in the KB into smaller sub-instances. This partitioning also enables us to reduce the cost of executing the rules.

Furthermore, in specific cases, TGs allow us to reason either by completely avoiding certain steps involved in the execution of the rules, or by performing those steps at the end and collectively for all the rules. Our experiments show that we get good runtime improvements with both alternatives.

Contributions. We propose techniques for computing 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.

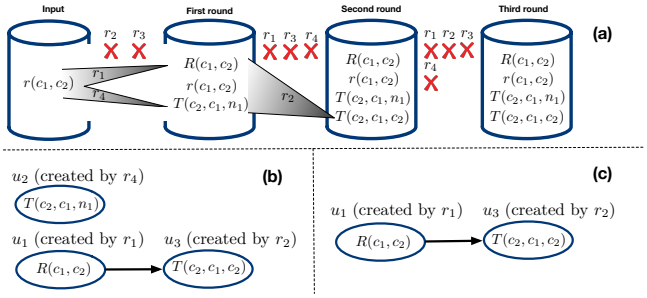

Figure 1: (a) Chase execution for Example 1, (b) the TG G_1, (c) the TG G_2. In (b) and (c), the facts shown inside the nodes are the results of reasoning over B using the TG.

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 [56] and Inferray [49], using well-established benchmarks, e.g., ChaseBench [10]. 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:

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

A version of this paper with more details and proofs is in [52].

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. 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 comprising the database* $B = \{r(c_1, c_2)\}$ *and the program* $P_1 = \{r_1, r_2, r_3, r_4\}$*:*

$$r(X, Y) \rightarrow R(X, Y) \quad (r_1)$$
$$R(X, Y) \rightarrow T(Y, X, Y) \quad (r_2)$$
$$T(Y, X, Y) \rightarrow R(X, Y) \quad (r_3)$$
$$r(X, Y) \rightarrow \exists Z.T(Y, X, Z) \quad (r_4)$$

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* $R(c_1, c_2)$*, which is added to the KB by the chase. 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 inefficiency is that of paying 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 the execution of* r_3 *always derives facts derived in previous rounds. Notice that this phenomenon 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 the fact* $T(c_2, c_1, c_2)$*, which is 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 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. Despite that the premise of* r_3 *cannot be instantiated using the derivations of* r_4*, the chase unnecessary pays the cost of executing* r_3 *over those facts.*

The root of these inefficiencies is that the chase considers in each round the entire KB as a source of potential derivations relying only to the SNE for avoiding redundant derivations. If we were able to "guide" the execution of the rules in a more clever way, then we could 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. Figure 1 (c) shows a graph for defining such a strategy. Informally, a *Trigger Graph* (TG) is precisely such a graph-based blueprint to compute the materialization.

In the remaining, we first provide a formal definition of TGs and study their properties. Next, we show that under certain cases we can compute TGs that support reasoning over any possible database and present techniques for computing such TGs in a static fashion, i.e., prior to reasoning. Next, we present techniques for computing TGs at reasoning time and show that such TGs support a wider class of rules than the ones statically computed. For both types of TGs we provide techniques for eliminating redundant computations.

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 $\text{arity}(p) \geq 0$, 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

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

where, P is an intensional predicate and for all $1 \leq i \leq n$, $\mathbf{X}_i \subseteq \mathbf{X}$ and $\mathbf{Y} \subseteq \mathbf{Y}$ ($\mathbf{X}_i$ and $\mathbf{Y}_i$ might be empty). We assume w.l.o.g. that the premise of a rule includes only extensional predicates or intensional predicates. We will denote extensional predicates with lowercase letters and intensional predicates with uppercase letters. Universal 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 $\text{body}(r)$ and $\text{head}(r)$. A rule is *Datalog* if it has no existentially quantified variables, *extensional* if $\text{body}(r)$ 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 from terms to terms; we write $\sigma = \{t_1 \mapsto s_1, \ldots, t_n \mapsto s_n\}$ 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 Consts $\cup$ 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 $\sigma : \mathcal{A}_1 \rightarrow \mathcal{A}_2$.

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 $\mathcal{A} = \{A_1, \ldots, A_n\}$ a *most general unifier* (MGU) μ for $\mathcal{A}$ is a substitution satisfying the following conditions: (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 $\sigma = \sigma' \circ \mu$ [4].

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

$$Q(X_1, \ldots, X_n) \leftarrow \bigwedge_{i=1}^{m} A_i \tag{2}$$

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 atom 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 $(\sigma(X_1), \ldots, \sigma(X_n))$. 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 [18].

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 facts from B. In particular, the result is a universal model of (P, B), which we can use for query answering [24]. 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 [10, 45]. 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 [9] 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 [22]. 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 [22]. The core chase [23] and the equivalent one do offer such guarantees.

For a chase variant, we use $Ch^i(K)$ or $Ch^i(P, B)$ to denote the instance computed during the i-th round and $Ch(P, B)$ to denote the (possibly infinite) result of the chase. Furthermore, we define the *chase graph* chaseGraph(P, B) for a KB (P, B) as the edge-labeled directed acyclic graph having as nodes the facts in $Ch(P, B)$ 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 satisfy this property.

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

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 the 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 rule(v). Observe that, by definition, nodes associated with extensional rules do not have incoming edges, and nodes v associated with an intensional rule have exactly *one* incoming edge associated with the j-th predicate of the body of v, i.e., there is exactly one node u such that $u \rightarrow_j v$. The latter might seem counter-intuitive 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 $v_1, \ldots, v_q$ associated with the same rule r can be linked with an edge labeled with j to different nodes $u_1, \ldots, u_q$ 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 a 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 for P is a TG for (P, B), if for each BCQ Q we have $(P, B) \models Q$ 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 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*, [22]), term-depth bounded (*TDB*, [35]) and first-order-rewritable (*FOR*, [16]).

Theorem 7. *For a program P, P is* FTG *iff it is* BDD*; and P is* TDB $\cap$ FOR *iff it is* BDD.

Below, we provide a sketch of the proof of Theorem 7. We start with the first part, namely that P is *FTG* iff it is *BDD*. In the forward direction, we show that if P is *FTG*, then it is *BDD* with bound the maximal depth of any instance-independent TG for P. In the backward direction if P is *BDD* with bound k, then there exists a (finite) EG G^k which is a TG for P. As we describe later, G^k can be computed by mimicking the chase. We now move to the second part of Theorem 7. If a program is *FOR*, then all facts that contain terms of depth at most k are produced in a fixed number of chase rounds. Therefore, if it is also *TDB*, then all relevant facts in the chase are also produced in a fixed number of steps.

We cannot determine if a program admits a finite TG.

Theorem 8. *The language of all programs that admit a finite TG is undecidable.*

The undecidability of *FTG* follows from the fact that *FOR* and *FTG* coincide for Datalog programs, which are always *TDB*.

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 9. *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 hold: (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 10. *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 $(u_1, \ldots, u_n)$ 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 $(u_1, \ldots, u_n)$ 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 $(u_1, \ldots, u_n)$ 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 is that, for each base instance B, the result of chasing B using a linear program P is

Algorithm 1 tglinear(P)

1: Let G be an empty EG
2: **for each** $f \in \mathcal{H}(P)$ **do**
3: Γ is an empty EG; μ is the empty mapping
4: **for each** $f_1 \rightarrow_r f_2 \in$ chaseGraph$(P, \{f\})$ **do**
5: **add** a fresh node u to $v(\Gamma)$ with rule$(u) := r$
6: $\mu(u) := f_1 \rightarrow_r f_2$
7: **for each** $v, u \in v(\Gamma)$ **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

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 combines these two ideas: it runs the chase for each fact in $\mathcal{H}(P)$ then tracks the rule executions and 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 11. *For any linear program P that is* FES, tglinear(P) *is a TG for P.*

Algorithm 1 has a double-exponential overhead.

Theorem 12. *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 $u' \in v(G)$ 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 13. *Let G be a TG for a program P, $u, v \in v(G)$ 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 14. *Let P be a linear program, G be an EG for P and $u, v \in v(G)$. 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 13 and Lemma 14, it follows that a node v of a TG can be "ignored" during 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 15. *For a TG G for a linear program P, the EG denoted by* minLinear(G) *is obtained by exhaustively applying the following steps: (i) choose a pair of nodes v, v' from G where v is dominated by v' and v' is not a successor of v, (ii) remove v from $v(G)$; 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 15 is correct: given a TG for a linear program P, the output of minLinear is still a TG for P.

Theorem 16. *For a TG G for a linear program P,* minLinear(G) *is a TG for P.*

We present an example demonstrating the TG computation and the minimization technique described above.

Example 17. *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 the following 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 $\text{rule}(v)$ 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; see 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 that we can use query containment tests to identify nodes that can be safely removed from TGs (and EGs). Intuitively, the naïve algorithm for computing TGs from Section 4 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 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 for eliminating nodes from TGs.

Definition 18. *Let v be a node in an EG G for a Datalog program. Let* $\text{rule}(v)$ *be* $\bigwedge_{i=1}^{n} A_i \rightarrow R(\mathbf{Y})$. *The EG-rewriting of v, denoted as* $\text{rew}(v)$, *is the CQ computed as follows (w.l.o.g. no pair of rules* $\text{rule}(u)$ *and* $\text{rule}(v)$ *with $u, v \in v(G)$ and $u \neq v$ shares variables):*

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

The rewriting algorithm described in Definition 18 is a variant of the rewriting algorithm in [26]. Our difference from [26] is that at each step of the rewriting process, we consider only the rule $\text{rule}(u)$ with u being the node with which α is associated with.

We demonstrate an example of Definition 18.

Example 19. *Consider the rules*

$$r(X_1, Y_1, Z_1) \rightarrow T(X_1, X_1, Y_1) \quad (r_8)$$

$$T(X_2, Y_2, Z_2) \rightarrow R(Y_2, Z_2) \quad (r_9)$$

where r is the only extensional predicate. Consider also an EG including the edge $u_1 \rightarrow_1 u_2$ where node u_1 is associated with r_8 and node u_2 is associated with r_9. To compute the EG-rewriting $\text{rew}(u_2)$, *we first form query $Q(Y_2, Z_2) \leftarrow R(Y_2, Z_2)$ and associate atom $R(Y_2, Z_2)$ with node u_2. Then, the next steps take place. First, since $R(Y_2, Z_2)$ is the only intensional atom in the body of the query we have $\alpha = R(Y_2, Z_2)$. Then, following step (ii) of Definition 18 and since node u_2 is associated with $R(Y_2, Z_2)$, we compute the MGU θ_1 of $\{\text{head}(u_2), R(Y_2, Z_2)\}$. We have $\theta_1 = \{Y_2 \mapsto Y_2, Z_2 \mapsto Z_2\}$. By applying the step (iii), the query becomes $Q(Y_2, Z_2) \leftarrow T(X_2, Y_2, Z_2)$. Due to the edge $u_1 \rightarrow_1 u_2$, in step (iv) we associate the fact $T(X_2, Y_2, Z_2)$ with node u_1 . In the second iteration, 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 $\{\text{head}(u_1), T(X_2, Y_2, Z_2)\}$. We have $\theta_2 = \{X_1 \mapsto Y_2, X_2 \mapsto Y_2, Y_1 \mapsto Z_2\}$. In step (iii), we replace $\alpha = T(X_2, Y_2, Z_2)$ with* $\text{body}(u_1)$, *i.e., $\{r(X_1, Y_1, Z_1)\}$, and apply θ_2 to the resulting query. The query becomes $Q(Y_2, Z_2) \leftarrow r(Y_2, Z_2, Z_1)$. Since there is no incoming edge to u_1, we associate no node with fact $r(Y_2, Z_2, Z_1)$. The algorithm then stops, since there is no intensional fact in the final query and returns $Q(Y_2, Z_2) \leftarrow r(Y_2, Z_2, Z_1)$ as the EG-rewriting of u_2.*

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

Lemma 20. *Let G be an EG for a Datalog program P and B be a base instance of P. Then for each $v \in v(G)$ we have: $v(B)$ includes exactly a fact $A(\mathbf{t})$ with A being the head predicate of* $\text{rule}(v)$, *for each answer $\mathbf{t}$ to the EG-rewriting of v on B.*

Our algorithm for removing nodes from EGs is stated below.

Definition 21. *The EG* $\text{minDatalog}(G)$ *is obtained from an EG G for a program P by exhaustively applying the following 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* $\text{head}(\text{rule}(v))$ *and of* $\text{head}(\text{rule}(u))$ *are the same and (iii) the EG-rewriting of v is contained in the EG-rewriting of u: (a) remove the node v from $v(G)$, and (b) add an edge $u \rightarrow_j w$, for each edge $v \rightarrow_j w$ occurring in G.*

The minimization technique of Definition 21 can be proven sound and to produce a TG with fewest nodes.

Theorem 22. *Let G be a TG for a Datalog program P. Then,* $\text{minDatalog}(G)$ *is also a TG for P. Furthermore, any other TG for P has at least as many nodes as* $\text{minDatalog}(G)$.

Deciding whether a TG of a Datalog program is of minimum size can be proven co-NP-complete. The problem's hardness lies in the necessity of performing query containment tests, carried out via homomorphism tests, which require exponential time on deterministic machines (unless $P = NP$) [18]. This hardness result supports the optimality of minDatalog in terms of complexity.

Theorem 23. *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.*

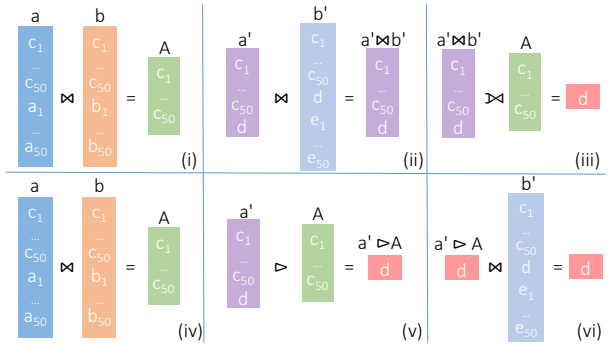

Figure 2: Different strategies for executing the rules from P_2.

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 24. *Consider the program P_2*

$$a(X) \wedge b(X) \rightarrow A(X) \qquad (r_{10})$$

$$a'(X) \wedge b'(X) \rightarrow A(X) \qquad (r_{11})$$

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 is shown in Figure 2.*

The upper part of Figure 2 shows the steps involved when executing r_{10} and r_{11} using the chase: (i) shows the joins involved when executing r_{10}; (ii)–(iii) show the joins involved when executing r_{11}: (ii) shows the join to compute body(r_{11}) *while (iii) shows the outer join involved when checking whether the conclusions of r_{11} 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_{10} stays the same (step (iv)), while for r_{11} we first compute all tuples that are in a′ *but not in* A *(step (v)) and use* a′ \ A *to restrict the tuples instantiating the body of r_{11} (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_{11}. The total cost of this strategy is: 100 (step (iv)) + 51 (step (v)) + 1 (step (vi))=152.*

Example 24 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 25. *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* 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 the query

$$Q'(\mathbf{Y}) \leftarrow f_{i_1} \wedge \cdots \wedge f_{i_m}$$

(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) $A(h(\mathbf{X})) \notin I$.

Algorithm 2 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: $G^k :=$ minDatalog(G^k)
6: **for each** node v of depth k **do**
7: **add** $v(B, I^{k-1})$ (cf. Definition 25) to I^k
8: **while** $I^k \neq I^{k-1}$
9: **return** I^∞

To help us understand Definition 25, let us apply it to Example 24. 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 25), but are not in I (step (2.ii) of Definition 25). In our implementation, we pick one extensional atom ($m = 1$) in step (1). To pick this atom, we consider each f_i in the body of rew(v), then compute the join as in step (v) of Example 24 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 26. *For a Datalog program P and a base instance B,* TGmat(P, B) = $Ch(P, B)$.

7 EVALUATION

We implemented Algorithm 1, TG-guided reasoning over a fixed TG (Definition 5) and Algorithm 2 in a new open-source reasoner called *GLog*. GLog is a fork of VLog [57] that shares the same code for handling the extensional relations while the code for reasoning is entirely novel. The source code is freely available at https://github.com/karmaresearch/glog. In the same repository, we also provide scripts to facilitate the replication of the experiments below, along with a link to the datasets.

To compare GLog against other reasoning engines, we consider three performance measures: the absolute reasoning time, the peak RAM consumption observed at reasoning time, and the total number of triggers. The last measure is considered 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 the best of our knowledge, is the most efficient system both time- and memory- wise [55, 57];
- the latest public release of RDFox from [50] as it outperforms all chase engines tested against ChaseBench [10]: ChaseFun, DEMo [46], LLunatic [25], PDQ [11] and Pegasus [39];
- the commercial state of the art chase engine COM (name is anonymized due to licensing restrictions);
- Inferray, an RDFS reasoner that outperforms RDFox [49] and that uses a columnar layout; and
- WebPIE, another high-performance RDFS reasoner that runs over Hadoop [56].

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

Scenario	#EDP's	#Rules			#IDP'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							
STB-128	0.15		167			1.9	
ONT-256	1		529			5.6	
RDFS (ρDF) scenarios							
LUBM	16.7		160			18	
YAGO	18.2		498016			27	

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 [10]. All engines ran using a single thread. We could not obtain access to the Vadalog [9] binaries. However, we perform an indirect comparison against Vadalog: we both compare against RDFox using the ChaseBench scenarios from [10].

Benchmarks. To asses the performance of GLog on linear and Datalog scenarios, we considered benchmarks previously used to evaluate reasoners, including VLog, RDFox, among others. LUBM [27] and UOBM [37] are synthetic benchmarks; DBpedia [13] (v2014, available online[1]) is a popular KB built from Wikipedia; Claros [47] and Reactome [20] are real-world ontologies[2]. For the linear scenarios, both VLog and GLog access the KBs stored as a collection of CSV files. For the Datalog scenarios, the KBs are stored instead with the RDF engine Trident [54].

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 [58], which was the technique used for evaluating our competitors [41, 55]. 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) is 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 [41]. 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 [57]. 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 [58] as described above. These programs are suffixed by "L". For Claros and LUBM, we used two additional programs, suffixed by "LE", created by [41] 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 [10]. 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 [3], 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 [10], we can use these two scenarios to indirectly compare against all the engines considered in [10].

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) [15] 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 [28] 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 15. 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

[1] https://www.cs.ox.ac.uk/isg/tools/RDFox/2014/AAAI/input/DBpedia/ttl/
[2] Both datasets are available in our data repository.

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

	VLog		RDFox		COM		GLog				
Scenario	Runtime	Memory	Runtime	Memory	Runtime	Memory	Comp.	Reason	w/o cleaning	w/cleaning	Memory
LUBM-LI	1.3	1617	22	2353	18.4	5122	0.007	0.2	0.207	1.1	1674
UOBM-LI	0.3	221	3.9	726	3.3	3570	0.01	0.015	0.025	0.2	219
DBpedia-LI	6.9	2579	44.1	3197	36.3	3767	0.448	0.776	1.224	4.5	2647
Claros-LI	5.6	2870	78.4	3918	72.3	5122	0.006	0.407	0.413	4.8	2586
React.-LI	1.8	1312	12.7	1448	9.9	4479	0.002	0.329	0.329	0.9	1312

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 *Memory* 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. Table 3 reports details about the TGs computed with such inputs. In this table, columns #N, #E, and D show the number of nodes, edges, and the depth (i.e., length of the longest shortest path), respectively, of the resulting TGs.

Table 3: Sizes of the TGs, linear and Datalog scenarios.

	Scenario	#N	#E	D
Linear	LUBM-LI	155	101	6
	UOBM-LI	313	206	9
	DBpedia-LI	12 600	8970	17
	Claros-LI	792	621	23
	React.-LI	386	263	8
Datalog	LUBM-L	56	33	4
	LUBM-LE	63	43	5
	UOBM-L	527	859	6
	DBpedia-L	4144	3062	8
	Claros-L	438	404	9
	Claros-LE	1461	3288	9

We summarize the 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 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 from the obvious benefit 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 it uses a single n-way join rather than multiple binary joins.

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 data changes.

7.3 Results for Datalog and ChaseBench

Table 4 summarizes our results on linear and non-linear Datalog rules. The last six columns show results for TGmat (Algorithm 2). To assess the impact of minDatalog and ruleExec, the rule execution strategy from Definition 25, 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. Table 3 reports the number of nodes, edges, and depth of the computed TGs when both minDatalog or ruleExec are employed. Table 5 shows results for ChaseBench, while Table 6 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 on GLog outperforms its competitors with all the considered inputs.

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

Table 4: Datalog scenarios. Runtime is in sec and memory in MB. $*$ denotes timeout after 1h.

	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	2.4	2.2	1.0	446	424	264
LUBM-LE	170.5	2725	116.6	3140	115.9	3610	17.3	17.2	16.1	1340	1310	1338
UOBM-L	7.3	1021	10	784	10	4215	2.6	2.4	2.6	335	335	342
DBpedia-L	41.6	827	64.4	3290	198.4	3878	20	19	19	1341	1352	1339
Claros-L	431	3170	2512	5491	2373.0	6453	122	118.3	119	6076	6077	6078
Claros-LE	2771.8	11 895	*	*	*	*	1040.8	1012.2	1053.9	48 464	48 474	48 455

Table 5: ChaseBench scenarios. Runtime in sec and memory in MB.

	VLog		RDFox		COM		GLog		TG Sizes		
Scenario	Runtime	Memory	Runtime	Memory	Runtime	Memory	Runtime	Memory	#N	#E	D
STB-128	0.5	1350	13.4	1747	10	5217	0.2	1266	192	0	0
ONT-256	2.3	4930	49	3997	35	6340	1	4930	577	65	3

Table 6: #Triggers (millions), Datalog scenarios.

	VLog	GLog		
Scenario		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 7: RDFS scenarios (L=LUBM, Y=YAGO). Runtime in sec and memory in MB.

	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

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 [52]). 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 [30] 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% respectively). In any case, even without this technique, GLog remains faster than its competitors in all cases. Detailed results can be found in [52].

Last, the ChaseBench experiments allow us to compare against Vadalog. According to [9], 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 6 shows that the total number of triggers produced with the Datalog rules. Recall that the number of triggers is a good indicator for estimating the amount of redundant computations. From the table, we see that this number 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 [55], 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 7 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 case.

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 8

Table 8: Scalability results. Runtime in sec, memory in GB.

	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

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 memory consumption, and the number of derived facts for each input database. We can see that GLog can reason on a KB 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 scales up to such an extent.

8 RELATED WORK

We briefly discuss different approaches adopted in the literature for improving the performance of the chase, and we compare them with our approach.

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 to improve the reasoning performance is to reduce the number of logically redundant facts by appropriately ordering the rules. In [56], 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 [14] 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 [30] 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 [25], PDQ and ChaseFun run on top of PostgreSQL; RDFox and VLog implement their own in-memory rule execution engines. 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 [9], which introduces a new chase variant for materializing KBs of warded Datalog programs. Warded Datalog is a class of programs not admiting 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 [9], 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* [7, 8, 48, 51]. The advantage of these techniques over the *query rewriting* ones, which are not based on materialization, e.g., [5, 17, 26], 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 [12], as well as for probabilistic KBs [53], 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 [6]: 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, and implemented it into a new reasoner called GLog. The primary goal of TGs is to reduce the amount of redundant computations performed during the execution of a chase algorithm. Our theoretical analysis and 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. In particular, our experiments report that GLog often outperforms existing state-of-the-art engines in terms of runtime and memory consumption. Moreover, in our largest experiment GLog was able to materialize a KB with 17B facts in less than an hour using commodity hardware. In terms of scalability, these are unprecedented results, as far as we know.

There are multiple research directions that are worth pursuing, either to further improve the performance or to extend the applicability of TGs to more scenarios. First, studying how TGs can be efficiently updated in response to KB updates is a natural continuation of our work. Further work is also needed to understand whether TGs can be computed using distributed computing architectures. Finally, it is interesting to study whether TGs can be combined with query rewriting techniques to further improve the performance of query answering.

Acknowledgments. David Carral conducted this research at TU Dresden, where he was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in projects number 389792660 (TRR 248, Center for Perspicuous Systems) and KR 4381/1-1 (Emmy Noether grant DIAMOND), and by the Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI).

REFERENCES

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

[2] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*. ACM, Melbourne, VIC, Australia, 1371–1382.

[3] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. 2015. The iBench Integration Metadata Generator. In *VLDB*. Springer-Verlag, Kohala Coast, HI, USA, 108–119.

[4] Franz Baader and Tobias Nipkow. 1999. *Term Rewriting and All That.* Cambridge University Press, USA.

[5] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A Toolkit for Query Answering with Existential Rules. In *RuleML*. Springer, Berlin, Germany, 328–344.

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

[7] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS*. ACM, Cambridge, MA, USA, 1–15.

[8] Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *Journal of Logic Programming* 10, 3,4 (1991), 255–299.

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

[10] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *PODS*. ACM, Raleigh, NC, USA, 37–52.

[11] M. Benedikt, J. Leblay, and E. Tsamoura. 2014. PDQ: Proof-driven Query Answering over Web-based Data. In *VLDB*. Springer-Verlag, Hangzhou, China, 1553–1556.

[12] Michael Benedikt, Boris Motik, and Efthymia Tsamoura. 2018. Goal-Driven Query Answering for Existential Rules With Equality. In *AAAI*. AAAI Press, New Orleans, LA, USA, 1761–1770.

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

[14] A. Bonifati, I. Ileana, and M. Linardi. 2016. Functional Dependencies Unleashed for Scalable Data Exchange. In *SSDBM*. ACM, Budapest, Hungary, 2:1–2:12.

[15] Dan Brickley, Ramanathan V Guha, and Brian McBride. 2014. RDF Schema 1.1. *W3C recommendation* 25 (2014), 2004–2014.

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

[17] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8, 3 (2017), 471–487.

[18] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC*. ACM, Boulder, CO, USA, 77–90.

[19] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Och, Chris Olston, and Fernando Pereira. 2015. Yedalog: Exploring Knowledge at Scale. In *SNAPL*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Asilomar, CA, USA, 63–78.

[20] David Croft, Antonio Fabregat Mundo, Robin Haw, Marija Milacic, Joel Weiser, Guanming Wu, Michael Caudy, Phani Garapati, Marc Gillespie, Maulik R Kamdar, et al. 2013. The Reactome pathway knowledge base. *Nucleic acids research* 42, D1 (2013), D472–D477.

[21] Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. 2002. *Neural-symbolic learning systems: foundations and applications.* Springer, Berlin, Germany.

[22] Stathis Delivorias, Michel Leclère, Marie-Laure Mugnier, and Federico Ulliana. 2018. On the k-Boundedness for Existential Rules. In *RuleML+RR*. Springer, Luxembourg, Luxembourg, 48–64.

[23] A. Deutsch, A. Nash, and J. B. Remmel. 2008. The chase revisited. In *PODS*. ACM, Vancouver, BC, Canada, 149–158.

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

[25] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. 2014. That's All Folks! LLUNATIC Goes Open Source. In *VLDB*. Springer-Verlag, Hangzhou, China, 1565–1568.

[26] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. 2014. Query Rewriting and Optimization for Ontological Databases. *ACM TODS* 39, 3 (2014), 25:1–25:46.

[27] Y. Guo, Z. Pan, and J. Heflin. 2011. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics* 3, 2-3 (2011), 158–182.

[28] J. Hoffart, F. Suchanek, K. Berberich, and G. Weikum. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence* 194 (2013), 28–61.

[29] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, JosÃĺ Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2020. Knowledge Graphs. *arXiv:2003.02320 [cs]* (2020). http://arxiv.org/abs/2003.02320 arXiv: 2003.02320.

[30] Pan Hu, Boris Motik, and Ian Horrocks. 2019. Modular Materialisation of Datalog Programs. In *AAAI*. AAAI Press, Honolulu, HI, USA, 2859–2866.

[31] Pan Hu, Jacopo Urbani, Boris Motik, and Ian Horrocks. 2019. Datalog Reasoning over Compressed RDF Knowledge Bases. In *CIKM*. ACM, Beijing, China, 2065–2068.

[32] Nikolaos Konstantinou, Martin Koehler, Edward Abel, Cristina Civili, Bernd Neumayr, Emanuel Sallinger, Alvaro A.A. Fernandes, Georg Gottlob, John A. Keane, Leonid Libkin, and Norman W. Paton. 2017. The VADA Architecture for Cost-Effective Data Wrangling. In *SIGMOD*. ACM, Raleigh, NC, USA, 1599–1602.

[33] Benno Kruit, Peter A. Boncz, and Jacopo Urbani. 2019. Extracting Novel Facts from Tables for Knowledge Graph Completion. In *ISWC*. Springer, Virtual Conference, 364–381.

[34] Benno Kruit, Hongyu He, and Jacopo Urbani. 2020. Tab2Know: Building a Knowledge Base from Tables in Scientific Papers. In *ISWC*. Springer, Springer, Virtual Conference, 349–365.

[35] Michel Leclère, Marie-Laure Mugnier, and Federico Ulliana. 2016. On Bounded Positive Existential Rules. In *DL*, Vol. 1577. CEUR-WS.org, Cape Town, South Africa.

[36] Jaehun Lee, Taeho Hwang, Jungho Park, Yunsu Lee, Boris Motik, and Ian Horrocks. 2020. A Context-Aware Recommendation System for Mobile Devices. In *ISWC*. CEUR-WS.org, Virtual Conference, 380–382.

[37] Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan, and Shengping Liu. 2006. Towards a complete OWL Ontology Benchmark. In *ESWC*. Springer, Budva, Montenegro, 125–139.

[38] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems* 4, 4 (1979), 455–469.

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

[40] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, Carsten Lutz, et al. 2009. OWL 2 web ontology language profiles. *W3C recommendation* 27 (2009), 61.

[41] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. 2014. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *AAAI*. AAAI Press, Quebec City, QUE, Canada, 129–137.

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

[43] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. In *ISWC*. Springer, Bethlehem, PA, USA, 3–20.

[44] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale Knowledge Graphs: Lessons and Challenges. *Commun. ACM* 62, 8 (2019), 36–43. https://doi.org/10.1145/3331166

[45] A. Onet. 2013. The Chase Procedure and its Applications in Data Exchange. In *DEIS*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Quebec City, QUE, Canada, 1–37.

[46] R. Pichler and V. Savenkov. 2009. DEMo: Data Exchange Modeling Tool. In *VLDB*. Springer-Verlag, Lyon, France, 1606–1609.

[47] Sebastian Rahtz, Alexander Dutton, Donna Kurtz, Graham Klyne, Andrew Zisserman, and Relja Arandjelovic. 2011. CLAROS—Collaborating on Delivering the Future of the Past. In *DH*. Stanford University Library, Stanford, CA, USA, 355–357.

[48] Damien Sereni, Pavel Avgustinov, and Oege de Moor. 2008. Adding Magic to an Optimising Datalog Compiler. In *SIGMOD*. ACM, Vancouver, BC, Canada, 553–566.

[49] Julien Subercaze, Christophe Gravier, Jules Chevalier, and Frederique Laforest. 2016. Inferray: Fast in-Memory RDF Inference. *Proceedings of the VLDB Endowment* 9, 6 (2016), 468–479.

[50] RDFox's team. 2020. RDFox public release. https://github.com/dbunibas/chasebench/tree/master/tools/rdfox. Accessed: 2020-11-10.

[51] K. Tuncay Tekle and Yanhong A. Liu. 2011. More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets. In *SIGMOD*. ACM, Athens, Greece, 661–672.

[52] Efthymia Tsamoura, David Carral, Enrico Malizia, and Jacopo Urbani. 2021. Materializing Knowledge Bases via Trigger Graphs. arXiv:2102.02753 [cs.DB]

[53] Efthymia Tsamoura, Víctor Gutiérrez-Basulto, and Angelika Kimmig. 2020. Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs. In *AAAI*. AAAI Press, New York, NY, USA, 10284–10291.

[54] Jacopo Urbani and Ceriel Jacobs. 2020. Adaptive Low-level Storage of Very Large Knowledge Graphs. In *WWW*. ACM / IW3C2, Virtual Conference, 1761–1772.

[55] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. 2016. Column-Oriented Datalog Materialization for Large Knowledge Graphs. In *AAAI*. AAAI Press, Phoenix, AZ, USA, 258–264.

[56] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri Bal. 2010. OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In *ESWC*. Springer, Heraklion, Greece, 213–227.

[57] Jacopo Urbani, Markus KrÃűtzsch, Ceriel Jacobs, Irina Dragoste, and David Carral. 2018. Efficient Model Construction for Horn Logic with VLog. In *IJCAR*. Springer, Oxford, UK, 680–688.

[58] Yujiao Zhou, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2013. Making the Most of your Triple Store: Query Answering in OWL 2 using an RL Reasoner. In *WWW*. International World Wide Web Conferences Steering Committee / ACM, Rio de Janeiro, Brazil, 1569–1580.